Mark Summerfield

Python 3

Výukový kurz

Procedurální i objektově orientované programování

Ladění, testování, distribuce zátěže do více vláken

Síťová komunikace, aplikace typu klient-server, programování databází

Využití regulárních výrazů, tvorba grafického uživatelského rozhraní

WWW

Ke stažení zdrojové kódy příkladů z knihy

CPRESS

Addison Wesley

Mark Summerfield

Python 3
Výukový kurz

Computer Press
Brno
2013

Python 3
Výukový kurz

Mark Summerfield

Překlad: Lukáš Krejčí
Obálka: Martin Sodomka
Odpovědný redaktor: Martin Herodek
Technický redaktor: Jiří Matoušek

Objednávky knih:
http://knihy.cpress.cz
www.albatrosmedia.cz
eshop@albatrosmedia.cz
bezplatná linka 800 555 513
ISBN 978-80-251-2737-7

Vydalo nakladatelství Computer Press v Brně roku 2013 ve společnosti Albatros Media a. s. se sídlem Na Pankráci 30, Praha 4. Číslo publikace 17 938.

Dotisk 1. vydání

ALBATROS MEDIA a.s.

Stručný obsah

Obsah

Úvod

Python je pravděpodobně nejsnadněji osvojitelný programovací jazyk, který se nejkrásněji používá. Kód jazyka Python je srozumitelný pro čtení i zápis a k tomu je stručný bez jakéhokoli nádechu tajemna. Python je velmi expresivní jazyk, což znamená, že obvykle stačí napsat daleko méně řádků kódu jazyka Python, než kolik by jich bylo zapotřebí pro ekvivalentní aplikace napsanou třeba v jazyku C++ nebo Java.

Python je multiplatformní jazyk. Obecně lze tedy říci, že program napsaný v jazyku Python lze spustit ve Windows i v unixových systémech, jako je Linux, BSD a Mac OS X, pouhým zkopírováním souboru či souborů, které tvoří daný program, na cílový stroj, aniž by jej bylo nutné „sestavovat" nebo kompilovat. Je možné vytvářet programy napsané v Pythonu, které používají funkčnost specifickou pro určitou platformu. To ale jen zřídkakdy nezbytné, protože téměř celá standardní knihovna Pythonu a většina knihoven třetích stran jsou plně a transparentně multiplatformní.

Jednou z opravdu silných stránek Pythonu je, že se dodává se skutečně kompletní standardní knihovnou, díky čemuž můžeme provádět třeba stahování souboru z Internetu, rozbalování zkomprimovaného archivního souboru nebo vytváření webového serveru jen pomocí jediného nebo několika málo řádků kódu. A kromě standardní knihovny je k dispozici tisíce knihoven třetích stran, z nichž některé poskytují ve srovnání se standardní knihovnou výkonnější a sofistikovanější možnosti (např. síťová knihovna Twisted nebo numerická knihovna NumPy), zatímco jiné poskytují funkčnost, která je příliš specializovaná na to, aby byla zahrnuta do standardní knihovny (např. simulační balíček SimPy). Většina knihoven třetích stran je k dispozici v seznamu balíčků pro jazyk Python (*pypi.python.org/pypi*).

V jazyku Python lze programovat v procedurálním, objektově orientovaném a v menší míře též funkcionálním stylu, i když v jádru je Pythonu objektově orientovaným jazykem. V této knize si ukážeme, jak psát procedurální a objektově orientované programy, a osvojíme si též prvky funkcionálního programování v jazyku Python.

Účelem této knihy je prezentovat způsob, jakým psát programy ve správném stylu Pythonu 3, a po přečtení se stát užitečnou příručkou pro jazyk Python 3. Přestože Python 3 je spíše evolučním nežli revolučním pokračováním Pythonu 2, nejsou u něj starší postupy již vhodné nebo nezbytné, přičemž se objevilo několik nových, využívajících přednosti Pythonu 3. Python 3 je lepší jazyk než Python 2 – je totiž postaven na mnohaleté zkušenosti s Pythonem 2 a přidává spoustu nových možností (a současně vypouští ty, které se v Pythonu 2 neosvědčily), díky nimž je programování ještě příjemnější, pohodlnější, snazší a konzistentnější.

Cílem knihy je naučit *jazyk* Python, a přestože se v ní seznámíte s množstvím standardních knihoven Pythonu, nesetkáte se se všemi. To ale není žádný problém, protože po přečtení knihy budete mít o Pythonu dost znalostí na to, abyste použili jakoukoli ze standardních knihoven nebo z knihoven třetích stran, a také na to, abyste byly schopni vytvářet své vlastní knihovní moduly.

Kniha je navržena tak, aby byla užitečná pro různé skupiny čtenářů, mezi něž patří samouci a amatérští programátoři, studenti, vědci, inženýři a všichni ostatní, kteří potřebují v rámci své práce něco naprogramovat, a samozřejmě také profesionální vývojáři a počítačový odborníci. Ovšem k tomu, aby byla kniha použitelná pro tak široké spektrum čtenářů, aniž by přitom znalé nudila nebo méně zkušené ztrácela, musí předpokládat alespoň nějaké zkušenosti s programováním (v libovolném jazyku). Především předpokládá základní znalosti v oblasti datových typů (jako jsou čísla a řetězce), datových typů představujících kolekce (jako jsou množiny a seznamy), řídících struktur (jako jsou příkazy `if` a `while`) a funkcí. Kromě toho některé příklady a cvičení předpokládají základní znalost značkovacího jazyka HTML a některé ze specializovanějších lekcí na konci vyžadují alespoň základní orientaci v probíraném tématu. Například Lekce o databázích předpokládá základní znalost jazyka SQL.

Kniha je uspořádána s ohledem na maximální možnou produktivitu a rychlost. Na konci první lekce budete schopni psát v jazyku Python malé, ale užitečné programy. V každé další lekci se seznámíte s novými tématy a zároveň témata probíraná v předchozích lekcích budete často rozšiřovat a prohlubovat. To znamená, že při postupném pročítání jednotlivých lekcí můžeme kdykoliv přestat– a s dosud získanými znalostmi budete schopni psát ucelené programy. Potom se můžete samozřejmě pustit do dalšího čtení a naučit se pokročilejší a sofistikovanější techniky. Z tohoto důvodu se s některými tématy seznámíte v jedné lekci a pak je blíže prozkoumáte v další či v několika pozdějších lekcích.

Při výuce nového programovacího jazyka se objevují dva hlavní problémy. Prvním je, že někdy, když je nutné se naučit nějaký nový princip, tento princip závisí na jiném, který zase přímo či nepřímo závisí na tom prvním. Druhý problém tkví v tom, že na začátku může čtenář o jazyku vědět jen něco málo neb vůbec nic, takže je velice obtížné prezentovat zajímavé nebo užitečné příklady či cvičení. V této knize se budeme snažit vyřešit oba problémy. První předpokládáním nějakým předchozích zkušeností s programováním a druhý představením „nádherného srdce“ jazyka Python v lekci 1, což je osm klíčových oblastí jazyka Python, které jsou samy o sobě dostatečné pro tvorbu ucházejících programů. Důsledkem tohoto přístupu je, že v prvních lekcích jsou některé příklady v trošičku umělém stylu, poněvadž používají pouze to, co jsme se do místa jejich prezentace naučili. Tento vliv se s každou další lekcí zmenšuje, a to až do konce lekce 7, kde jsou všechny příklady zapsány stylem, který je pro Python 3 naprosto přirozený.

Přístup knihy je veskrze praktický, takže budete vyzýváni, abyste si příklady a cvičení sami vyzkoušeli a získali tak určitou praxi. Kdykoliv to bude možné, použijeme pro příklady kompletní programy a moduly představující realistické případy užití. Příklady, řešení pro cvičení a errata ke knize jsou k dispozici na stránce *http://knihy.cpress.cz/K1747*.

I když je nejlepší používat nejnovější verzi Pythonu 3, nemusí to být vždy možné, pokud uživatelé nemohou nebo nechtějí svoji verzi Pythonu modernizovat. Každý příklad v této knize funguje s Pythonem 3.0, přičemž příklady a funkční prvky specifické pro Python 3.1 jsou výslovně uvedeny.

Přestože je možné tuto knihu použít pro vývoj softwaru, který používá pouze Python 3.0, měli by všichni, kteří chtějí vytvářet software, který se bude používat řadu let a který by měl být kompatibilní s pozdějšími vydáními Pythonu 3.x, používat Python ve verzi 3.1 a podporovat tuto verzi jako nejstarší verzi Pythonu 3. To je dáno zčásti tím, že Python 3.1 nabízí několik velice pěkných nových možností, ale především tím, že vývojáři Pythonu důrazně doporučují používat Python 3.1 (nebo

novější). Vývojáři se rozhodli, že Python 3.0.1 bude posledním vydáním v řadě 3.0.y a že již žádná další vydání v této řadě nebudou, a to ani tehdy, pokud se objeví nějaké chyby či bezpečnostní problémy. Chtějí totiž, aby všichni uživatelé Pythonu 3 přešli k Pythonu 3.1 (nebo k novější verzi), který bude mít běžná vydání s opravami chyb a bezpečnostních problémů.

Uspořádání knihy

Lekce 1 prezentuje osm klíčových oblastí jazyka Python, které jsou dostatečné pro psaní kompletních programů. Dále popisuje některá z dostupných programovacích prostředí Pythonu a prezentuje dva malinké programy sestavené s využitím osmi klíčových oblastí jazyka Python probíraných v dřívější části lekce.

Lekce 2 až 5 představují prvky procedurálního programování jazyka Python, včetně jeho základních datových typů, datových typů představujících kolekce a řady užitečných vestavěných funkcí a řídících struktur společně s velmi jednoduchou prací se soubory. Lekce 5 ukazuje, jak vytvářet vlastní moduly a balíčky, a poskytuje přehled standardní knihovny Pythonu, abyste měli dobrou představu o funkcích, které jsou v Pythonu ihned k dispozici – a díky kterým nemusíte znovu objevovat kolo.

Lekce 6 poskytuje důkladné seznámení s objektově orientovaným programováním v jazyku Python. Veškerá látka týkající se procedurálního programování, kterou jste se naučili v předchozích lekcích, i nadále platí, protože objektově orientované programování je postaveno na procedurálních základech. Využívá tak například stejné datové typy, datové typy představující kolekce a řídící struktury.

Lekce 7 se věnuje zápisu a čtení souborů. V případě binárních souborů se navíc jedná o kompresi a náhodný přístup a u textových souborů o syntaktickou analýzu prováděnou ručně a pomocí regulárních výrazů. Tato Lekce dále ukazuje, jak zapisovat a číst soubory XML, včetně použití stromů elementů, modelu DOM (Document Object Model – objektový model dokumentu) a rozhraní SAX (Simple API for XML – jednoduché aplikační rozhraní pro XML).

Lekce 8 reviduje látku probíranou v několika předchozích lekcích a prozkoumává řadu pokročilejších prvků jazyka Python v oblasti datových typů a datových typů představujících kolekce, řídících struktur, funkcí a objektově orientovaného programování. Tato Lekce dále představuje spoustu nových funkcí, tříd a pokročilých technologií, včetně funkcionálního stylu programování a použití korutin. Probíraná témata jsou sice náročná, ale zato velice užitečná.

Lekce 9 se od všech předchozích lekcí liší v tom, že místo představování nových prvků jazyka Python probírá techniky a knihovny pro ladění, testování a profilování programů.

Zbývající lekce se věnují nejrůznějším pokročilým tématům. Lekce 10 ukazuje techniky pro rozložení pracovní zátěže programu do více procesů nebo vláken. Lekce 11 ukazuje, jak pomocí standardní podpory Pythonu pro komunikace přes síť vytvářet aplikace s architekturou klient-server. Lekce 12 se věnuje databázovému programování (jednoduché soubory DBM s daty ve tvaru klíč-hodnota i databáze SQL).

Lekce 13 vysvětluje a demonstruje minijazyk regulárních výrazů v Pythonu a věnuje se modulu pro regulární výrazy. Lekce 14 pokračuje dále a ukazuje základní techniky syntaktické analýzy pomocí regulárních výrazů a také použití dvou modulů třetích stran, PyParsing a PLY. Nakonec Lekce 15 představuje programování grafického uživatelského rozhraní (Graphical User Interface neboli GUI)

pomocí modulu `tkinter`, který je součástí standardní knihovny Pythonu. Kniha má dále velmi stručný závěr a samozřejmě rejstřík.

Mnohé lekce jsou pro udržení související látky na jednom místě docela dlouhé. Nicméně lekce jsou rozděleny na části, oddíly a někdy i pododdíly, takže je lze číst takovým tempem, které vám nejlépe vyhovuje – třeba přečtením jedné části nebo jednoho oddílu najednou.

Získání a instalace Pythonu 3

Máte-li moderní a aktualizovaný unixový systém nebo Mac, pak již máte Python 3 nejspíše nainstalovaný, což ověříte zapsáním příkazu `python -V` (jedná se o velké písmeno V) do konzoly (`Terminal.app` v systému Mac OS X). Jedná-li se o verzi 3.x, pak je Python 3 již přítomen, takže nemusíte nic instalovat. Pokud Python nebyl vůbec nalezen, může to být tím, že má název, který obsahuje číslo verze. Zkuste napsat `python3 -V`, a pokud ani to nefunguje, tak `python3.0 -V` nebo `python3.1 -V`. Pokud některá z těchto možností funguje, pak víte, že již máte Python nainstalovaný, a znáte jeho verzi i název. (V této knize používáme název `python3`, můžeme ale používat takový název, který u vás funguje, například `python3.1`.) Pokud nemáte nainstalovanou žádnou verzi Pythonu 3, čtěte dále.

Pro systémy Windows a Mac OS X jsou k dispozici snadno použitelné grafické instalační balíčky, které vás provedou instalačním procesem krok za krokem. Můžete je stáhnout na adrese *www.python.org/download*. Pro Windows stáhněte balíček „Windows x86 MSI Installer“, pokud si ovšem nejste jisti, že váš stroj má jiný procesor, pro který je dodáván jiný instalátor. Máte-li například AMD64, sáhněte po balíčku „Windows X86-64 MSI Installer“. Jakmile instalační balíček získáte, stačí jej už jen spustit a řídit se pokyny na obrazovce.

Pro Linux, BSD a další unixové systémy (kromě systému Mac OS X, pro nějž je k dispozici instalační soubor `.dmg`) spočívá nejjednodušší způsob instalace Pythonu v použití systému pro správu balíčků vašeho operačním systému. Ve většině případů je Python k dispozici v několika samostatných balíčcích. Například v systému Ubuntu (od verze 8) existuje `python3.0` pro Python, `idle-python3.0` pro editor IDLE (jednoduché vývojové prostředí) a `python3.0-doc` pro dokumentaci – společně se spoustou dalších balíčků, které vedle standardní knihovny poskytují doplňky s dalšími funkčními prvky. (Pro Python ve verzi 3.1 budou názvy balíčků samozřejmě začínat `python-3.1`.)

Pokud na vašem systému nejsou k dispozici žádné balíčky s Pythonem 3, pak musíte stáhnout zdrojový kód z adresy *www.python.org/download* a sestavit Python úplně od začátku. Stáhněte jeden z archivů tarball se zdroji a v případě komprese gzip jej rozbalte příkazem `tar xvfz Python-3.1.tgz` nebo v případě komprese bzip2 příkazem tar `xvfj Python-3.1.tar.bz2`. (Číslo verze se může lišit, například `Python-3.1.1.tgz` nebo `Python-3.1.2.tar.bz2`, ale stačí jednoduše nahradit 3.1 skutečným číslem verze.) Konfigurace sestavení probíhá standardním způsobem. Nejdříve se přesuňte do nově vytvořeného adresáře Python-3.1 a spusťte `./configure`. (Pro lokální instalaci můžete použít volbu `--prefix`.) Dále spusťte `make`.

Je možné, že na konci obdržíte několik zpráv oznamujících, že ne všechny moduly bylo možné sestavit. To obvykle znamená, že na svém počítači nemáte některé z požadovaných knihoven nebo hlaviček. Pokud například nelze sestavit modul `readline`, použijte systém pro správu balíčků pro nainstalování odpovídající vývojové knihovny – například `readline-devel` na systémech na bázi distribuci Fedora nebo `readline-dev` na systémech na bázi distribuce Debian, jako je například Ubuntu. Další

modul, který se nemusí ihned sestavit, je modul `tkinter`, který závisí na vývojových knihovnách Tcl a Tk, což jsou moduly `tcl-devel` a `tk-devel` na systémech na bázi distribuce Fedora a moduly `tcl8.5-dev` a `tk8.5-dev` na systémech na bázi distribuce Debian (s tím, že vedlejší verze nemusí být 5). Naneštěstí nejsou názvy příslušných balíčků na první pohled zřejmé, a proto může být nutné obrátit se s žádostí o pomoc na diskuzní fórum Pythonu. Po nainstalování chybějících balíčků spusťte znovu `./configure` a `make`.

Po úspěšně provedeném příkazu `make` se můžete spuštěním příkazu `make test` přesvědčit, zda je všechno v pořádku. Není to ale nezbytné a navíc může dokončení tohoto příkazu trvat spoustu minut.

Pokud použijete volbu `--prefix` pro lokální instalaci, pak stačí spustit `make install`. Pokud v případě Pythonu 3.1 instalujete třeba do adresáře `~/local/python31`, pak přidáním adresáře `~/local/python31/bin` do své proměnné prostředí `PATH` budete schopni spouštět Python příkazem `python3` a editor IDLE příkazem `idle3`. Pokud již máte lokální adresář pro spustitelné soubory, který se nachází v proměnné prostředí `PATH` (např. `~/bin`), pak můžete místo změny proměnné `PATH` přidat symbolické odkazy. Máte-li spustitelné soubory například v adresáři `~/bin` a Python jste nainstalovali do adresáře `~/local/python31`, pak můžete vytvořit vhodné odkazy spuštěním příkazů `ln -s ~/local/python31/bin/python3 ~/bin/python3` a `~/local/python31/bin/idle3 ~/bin/idle3`. Pro účely této knihy jsme v systémech Linux a Mac OS X přesně takto provedli lokální instalaci a přidali symbolické odkazy, přičemž ve Windows jsme použili binární instalátor.

Pokud nepoužijete volbu `--prefix` a máte přístup uživatele „root“, přihlaste se jako „root“ a proveďte příkaz `make install`. Na systémech podporujících příkaz `sudo`, jako je například Ubuntu, spusťte příkaz `sudo make install`. Je-li v systému Python 2, adresář `/usr/bin/python` se nezmění a Python 3 bude dostupný jako `python3.0` (nebo `python3.1` podle nainstalované verze) a od verze Python 3.1 také jako `python3`. Editor IDLE pro Python 3.0 se nainstaluje jako `idle`, takže pokud potřebujete i nadále přístup k editoru IDLE pro Python 2, musíte před provedením instalace starý editor IDLE přejmenovat (např. na `/usr/bin/idle2`). Python 3.1 nainstaluje editor IDLE jako `idle3`, takže k žádnému konfliktu s editorem IDLE pro Python 2 nedochází.

Poděkování

Nejdříve bych chtěl poděkovat za odezvu, kterou jsem obdržel od čtenářů první edice, kteří mi poskytli připomínky ohledně oprav, návrhů nebo obojího.

Mé další poděkování míří k odborným recenzentům knihy, počínaje Jasminem Blanchettem, který je počítačovým odborníkem, programátorem a spisovatelem, s nímž jsem spolupracoval na dvou knihách o C++ a knihovně Qt. Jeho zapojení do plánování lekcí, jeho rady, kritika všech příkladů i jeho pečlivé čtení významným způsobem zlepšily kvalitu této knihy.

Georg Brandl je přední vývojář a dokumentátor v oblasti Pythonu odpovědný za vytvoření nové sady dokumentačních nástrojů. Všiml si spousty zákeřných chyb a velice trpělivě a neústupně je vysvětloval, dokud nebyly pochopeny a opraveny. Dále provedl řadu zlepšení v rámci příkladů.

Phil Thompson je expertem na jazyk Python a tvůrcem knihovny PyQt, což je pravděpodobně nejlepší knihovna GUI pro Python. Jeho bystrozraká a podnětná odezva vedla k řadě vyjasnění a korekcí.

Trenton Schulz je hlavní softwarový inženýr ve společnosti Qt Software (před odkoupením společností Nokia známé jako Trolltech), který byl cenným recenzentem všech mých předchozích knih a který mi opět přišel na pomoc. Pozorně přečetl a množství jeho připomínek napomohlo k ujasnění řady problémů a vedlo k značným zlepšením v textu.

Kromě výše zmíněných recenzentů, z nichž každý přečetl celou knihu, nesmím zapomenout na Davida Boddieho, předního autora odborných titulů ve společnosti Qt Software, zkušeného odborníka na jazyk Python a vývojáře softwaru s otevřeným zdrojovým kódem, který přečetl a poskytl cennou odezvu na několik částí této knihy.

Pro tuto druhou edici bych také rád poděkoval Paulu McGuireovi (autorovi modulu PyParsing), který byl tak laskav a zkontroloval příklady využívající modul PyParsing, které se objevily v nové lekci věnované syntaktické analýze, a který mi poskytl spoustu uvážených a užitečných rad. A pro stejnou lekci zkontroloval David Beazley (autor modulu PLY) příklady využívající modul PLY a postaral se o cennou odezvu. Kromě toho Jasmin Blauchebt, Treon Schulz, Georg Braudla Phil Thompson přečetli většinu z nového materiálu této druhé edice a poskytli mi velice hodnotnou zpětnou vazbu.

Díky patří také Guidovi van Rossumovi, tvůrci jazyka Python, jakož i širší komunitě kolem Pythonu, která se významným způsobem podílela na tvorbě Pythonu a zvláště jeho knihoven, které jsou nesmírně užitečné a které je radost používat.

A jako vždy děkuji Jeffu Kingstonovi, tvůrci jazyka Lout pro sazbu písma, který používám již více než deset let.

Zvláštní díky patří mé redaktorce Debře Williams Cauley za její podporu a také za to, že se opět postarala, aby měl celý proces co nejhladší průběh. Děkuji též Anně Popick, která se tak dobře starala o produkční proces, a korektorovi Audrey Doyle, který opět odvedl naprosto skvělou práci. A v souvislosti s touto druhou edicí chci též poděkovat Jennifer Lindnerové za pomoc při udržování nového materiálu na srozumitelné úrovni a japonskému překladateli první edice Takahiro Nagaovi za odhalení zákeřných chyb, které jsem měl možnost v této edici opravit.

V neposlední řadě bych chtěl poděkovat své ženě Andree za to, že zvládla mé buzení ve čtyři hodiny ráno, kdy často přicházely nápady a opravy kódu, které se tu a tam dožadovaly poznamenání nebo otestování, a za její lásku, věrnost a podporu.

LEKCE 1

Rychlý úvod do procedurálního programování

V této lekci:

- Tvorba a spouštění programů napsaných v jazyku Python
- Nádherné srdce jazyka Python

Tato Lekce vás vybaví všemi informacemi, které jsou nezbytné k tomu, abyste mohli v jazyku Python začít psát své programy. Důrazně doporučujeme nainstalovat Python, pokud jste tak již neučinili, abyste si mohli vše, co se zde naučíte, ihned vyzkoušet (vysvětlení způsobu získání a instalace Pythonu na všechny přední platformy najdete v Úvodu).

V první části této lekce si ukážeme, jak vytvářet a spouštět programy napsané v jazyku Python. K psaní kódu jazyka Python můžete používat svůj oblíbený textový editor. Na druhou stranu programovací prostředí IDLE probírané v této části nabízí kromě editoru kódu také doplňkové funkce, mezi něž patří prvky pro experimentování s kódem jazyka Python a pro ladění programů napsaných v jazyku Python.

Ve druhé části se seznámíte s osmi klíčovými částmi jazyka Python, které jsou samy o sobě dostatečné pro vytváření užitečných programů. Všem těmto částem se budeme podrobně věnovat v dalších lekcích, přičemž v průběhu knihy budeme doplňovat zbývající prvky jazyka Python, takže na jejím konci budete znát celý jazyk a budete schopni použít vše, co nabízí, ve svých vlastních programech.

V poslední části této lekce si ukážeme dva krátké programy, které používají jistou podmnožinu prvků jazyka Python, které jsme si představili ve druhé části, takže si ihned vyzkoušíte, jak se v jazyku Python programuje.

Tvorba a spouštění programů napsaných v jazyku Python

Kódování znaků ➤ 95

Kód jazyka Python lze psát pomocí libovolného textového editoru, který dokáže načítat a ukládat text v kódování ASCII nebo UTF-8 znakové sady Unicode. U souborů s kódem jazyka Python se standardně předpokládá, že používají kódování UTF-8, což je nadmnožina kódování ASCII, která dokáže docela dobře reprezentovat libovolný znak libovolného jazyka. Soubory s kódem jazyka Python mají obvykle příponu `.py`, i když na některých systémech na bázi Unixu (např. Linux a Mac OS X) jsou některé aplikace napsané v jazyku Python bez přípony. Programy napsané v jazyku Python využívající grafické uživatelské rozhraní (GUI) mají většinou příponu `.pyw`, především na systémech Windows a Mac OS X. V této knize budeme pro konzolové programy Pythonu a pro moduly Pythonu používat vždy příponu `.py` a pro programy GUI příponu `.pyw`. Všechny příklady uvedené v této knize lze spustit beze změny na všech platformách, na nichž je dostupný Python 3.

Pro jistotu, že je vše správně připraveno, a také pro demonstraci klasického prvního příkladu, vytvořte v obyčejném textovém editoru (např. Poznámkový blok – vzápětí si ukážeme lepší) soubor s názvem `hello.py` a s následujícím obsahem:

```
#!/usr/bin/env python3

print("Ahoj", "světe!")
```

Na prvním řádku je komentář. Komentář v jazyku Python začíná znakem # a pokračuje až na konec řádku (smysl výše uvedeného komentáře si vysvětlíme vzápětí). Druhý řádek je prázdný – Python sice prázdné řádky ignoruje, ale lidskému oku se větší bloky čtou lépe, jsou-li rozdělené. Na třetím řádku je kód jazyka Python. Zde voláme funkci `print()` se dvěma argumenty, z nichž každý je typu

`str` (řetězec, tj. posloupnost znaků). Všechny příkazy uvedené v souboru `.py` se provádějí postupně, přičemž se začíná prvním příkazem a pokračuje se po jednotlivých řádcích. To je odlišné od některých jiných jazyků, jako je například C++ nebo Java, které musejí obsahovat určitou zahajovací funkci či metodu se zvláštním názvem. Tok programu lze samozřejmě korigovat, o čemž se přesvědčíme v následující části při probírání řídících struktur jazyka Python.

Budeme předpokládat, že uživatelé systému Windows mají svůj kód jazyka Python v adresáři `C:\py3eg` a uživatelé systému na bázi Unixu (např. Unix, Linux nebo Mac OS X) jej mají umístěný v adresáři `$HOME/py3eg`. Soubor `hello.py` tedy uložte do adresáře `py3eg` a zavřete textový editor.

Program máme hotový, takže jej můžeme spustit. Programy napsané v Pythonu se spouštějí pomocí interpretu jazyka Python, což se obvykle provádí uvnitř okna příkazového řádku. Tomu se ve Windows říká „Konzola" nebo „Příkazový řádek" a většinou je k dispozici v nabídce **Start → Všechny programy → Příslušenství**. V systému Mac OS X nabízí konzoli program `Terminal.app` (umístěný standardně ve skupině Applications/Utilities), k němuž se dostanete přes Finder, přičemž na ostatních systémech na bázi Unixu můžete použít `xterm` nebo konzolu poskytovanou okénkovým systémem (např. `konsole` nebo `gnome-terminal`).

Spusťte konzolu a ve Windows napište následující povely (u nichž předpokládáme, že je Python nainstalován ve výchozí lokaci) – výstup konzoly je vytištěn `normálním písmem`, zatímco vámi zadávaný vstup `zvýrazněn`:

```
C:\>cd c:\py3eg
C:\py3eg\>c:\python31\python.exe hello.py
```

Povel `cd` (změna adresáře) obsahuje absolutní cestu, a proto nezáleží na tom, z jakého adresáře začínáte.

Uživatelé Unixu zadají níže uvedené povely (předpokládáme, že Python 3 je v systémové proměnné `PATH`):*

```
$ cd $HOME/py3eg
$ python3 hello.py
```

V obou případech by měl být výstup stejný:

```
Ahoj, světe!
```

Všimněte si, že není-li uvedeno jinak, má Python chování v systému Mac OS X úplně stejně jako v kterémkoli jiném systému na bázi Unixu. Kdykoli se tedy budeme odkazovat na Unix, budeme mít na mysli Linux, BSD, Mac OS X a většinu ostatních Unixů a Unixu podobných systémů.

Ačkoliv má náš program jen jeden prováděný příkaz, jeho spuštěním si můžeme odvodit několik informací o funkci `print()`. Funkce `print()` je vestavěnou součástí jazyka Python, takže ji nemusíme „importovat" nebo „začleňovat" z nějaké knihovny. Dále vidíme, že každý vypisovaný prvek odděluje jednou mezerou a za posledním prvkem vypíše znak nového řádku. Jedná se o výchozí chování, které lze změnit, jak uvidíme později. Další věcí, která stojí za povšimnutí, je skutečnost, že funkce `print()` může přijímat libovolný počet argumentů.

`print()` ➤ 179

* Výzva příkazového řádku v systému Unix se může od níže uvedeného znaku $ klidně lišit, což není vůbec podstatné.

Psaní výše uvedených povelů ke spouštění programů napsaných v jazyku Python začne být brzy velmi otravné a pracné. Naštěstí lze ve Windows i v Unixu použít pohodlnější postup. Za předpokladu, že se nacházíme v adresáři `py3eg`, můžeme ve Windows jednoduše napsat:

```
C:\py3eg\>hello.py
```

Jakmile se v konzolu objeví přípona `.py`, zavolá systém Windows pomocí svého registru souborových asociací interpret jazyka Python.

Tento postup však nefunguje za všech okolností, poněvadž některé verze Windows obsahují chybu, která má v některých situacích vliv na provádění interpretovaných programů, které se spouštějí v důsledku asociace s jistou příponou souboru. To se netýká jen Pythonu, ale i další interpretů, a dokonce i některých souborů s příponou `.bat`. Pokud k tomuto problému dojde, tak prostě spusťte Python přímo.

Pokud v systému Windows vypadá váš výstup takto:

```
('Hello', 'World!')
```

znamená to, že se v systému nachází Python 2 a spouští se místo Pythonu 3. Jedno z řešení spočívá ve změně souborové asociace `.py` z Pythonu 2 na Python 3. Dalším řešením (méně pohodlným, ale bezpečnějším) je umístit interpret Pythonu 3 do cesty (předpokládáme, že je nainstalován ve výchozí lokaci) a pokaždé jej explicitně spouštět (tím se vyhnete výše zmíněné chybě systému Windows s asociacemi souborů):

```
C:\py3eg\>path=c:\python31;%path%
C:\py3eg\>python hello.py
```

Mnohem pohodlnější je vytvořit si soubor py3.bat s jediným řádkem `path=c:\python31;%path%` a uložit jej do adresáře `C:\Windows`. Kdykoliv pak spustíte konzolu s úmyslem provádět programy napsané v jazyku Python 3, tak nejdříve spustíte soubor `py3.bat`. Další možností je nechat si soubor `py3.bat` spouštět automaticky. K tomu stačí otevřít vlastnosti konzoly (v nabídce **Start** vyhledejte konzolu, klepněte na ni pravým tlačítkem a zvolte příkaz **Vlastnosti**) a v záložce **Zástupce** (**Shortcut**) přidejte do pole **Cíl** (**Target**) text „`/u /k c:\windows\py3.bat`“ (všimněte si mezery před, mezi a za volbami /u a /k a ujistěte se, že je tento text na konci za textem „`cmd.exe`“).

V Unixu je nutné nejdříve udělat ze souboru spustitelný soubor, který pak můžeme spustit:

```
$ chmod +x hello.py
$ ./hello.py
```

Příkaz `chmod` stačí pochopitelně spustit pouze jednou. Pak již můžeme napsat `./hello.py` a program se spustí.

Když se v Unixu spustí z konzoly nějaký program, nejdříve se přečtou první dva bajty souboru.* Jsou-li těmito bajty ASCII znaky `#!`, tak shell předpokládá, že soubor spustí interpret, který je specifiko-

* O interakci mezi uživatelem a konzolou se stará program označovaný jako „shell“. Rozdíl mezi konzolou a programem shell pro nás není podstatný, takže budeme používat oba výrazy pro označení téhož prostředí.

ván na prvním řádku. Tento řádek se označuje jako *shebang* (shell execute), a pokud je uveden, musí být vždy na prvním řádku souboru.

Řádek shebang má obvykle jednu z následujících podob:

```
#!/usr/bin/python3
```

nebo:

```
#!/usr/bin/env python3
```

Při použití prvního způsobu se použije uvedený interpret. Tento způsob může být nezbytný pro programy napsané v Pythonu, které budou spouštěny webovým serverem, ačkoliv zadaná cesta se samozřejmě může lišit. Při použití druhého způsobu se použije první interpret `python3` nalezený v aktuální prostředí shellu. Druhý způsob je všestrannější, protože interpret Pythonu 3 nemusí být umístěn v adresáři `/usr/bin` (může být například v adresáři `/usr/local/bin` nebo `$HOME`). Řádek shebang není ve Windows nutný (je však naprosto neškodný). Všechny příklady v této knize mají řádek shebang ve druhé z uvedených podob, ačkoliv si jej zde ukazovat nebudeme.

Všimněte si, že u systému na bázi Unixu předpokládáme, že název spustitelného souboru (nebo symbolického odkazu na něj) interpretu jazyka Python 3 v proměnné `PATH` je `python3`. Pokud to neplatí, pak musíte v příkladech upravit řádek shebang tak, aby používal správný název (nebo opravit název a cestu, používáte-li první způsob), nebo vytvořit symbolický odkaz ze spustitelného souboru interpretu Pythonu 3 na název `python3` někde v cestě `PATH`.

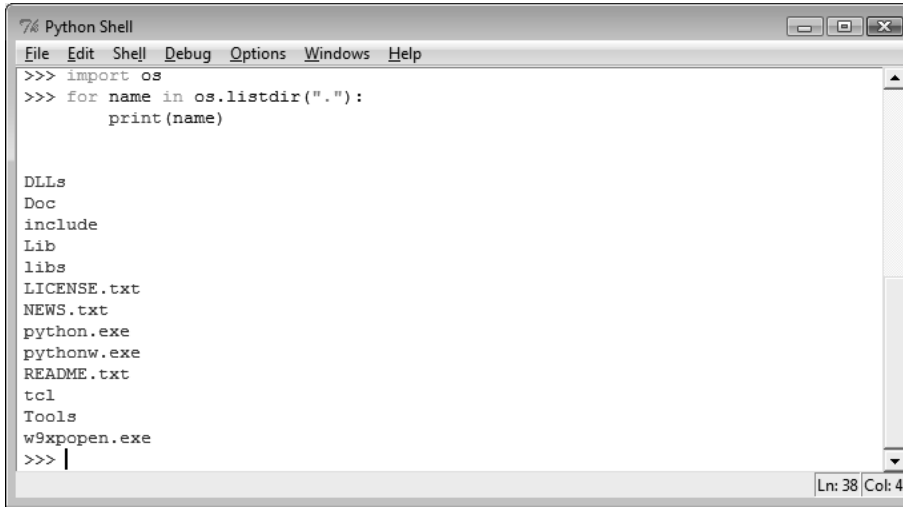
Řada výkonných editorů holého textu, jako je Vim nebo Emacs, obsahuje vestavěnou podporu pro úpravu programů psaných v jazyku Python. Tato podpora obvykle zahrnuje barevné zvýrazňování syntaxe a správné odsazování řádků. Alternativou je pak programovací prostředí Pythonu s názvem IDLE. V systémech Windows a Mac OS X je toto prostředí standardní součástí instalace Pythonu. V systémech na bázi Unixu se při sestavování z archivu tarball sestavuje prostředí IDLE společně s interpretem jazyka Python, pokud ale používáte správce balíčků, pak můžete prostředí IDLE nainstalovat jako samostatný balíček (viz popis v Úvodu).

Získání a instalace Pythonu 3 ➤ 16

Jak je z obrazovky na obrázku 1.1 patrné, působí IDLE na první pohled poněkud archaickým způsobem připomínajícím doby Motifu na Unixu a Windows 95. To je dáno tím, že namísto moderních knihoven GUI, jako jsou PyGtk, PyQt nebo wxPython, používá knihovnu Tkinter postavenou na bázi Tk (viz Lekce 15). Důvodem pro použití knihovny Tkinter je směsice historie, liberálních licenčních podmínek a skutečnosti, že Tkinter je v porovnání s ostatními knihovnami GUI mnohem menší. Výhodou je navíc to, že prostředí IDLE se standardně dodává s Pythonem a velmi snadno se osvojuje a používá.

Prostředí IDLE poskytuje tři klíčové funkce: možnost zadávat výrazy a kód jazyka Python a sledovat výsledky přímo v okně **Python Shell**, editor kódu, který nabízí barevné zvýrazňování syntaxe a odsazování kódu jazyka Python, a ladicí nástroj, pomocí něhož lze krokovat kód pro snazší identifikaci a odstranění chyb. Okno Python Shell je zvláště užitečné pro zkoušení jednoduchých algoritmů, úryvků kódu a regulárních výrazů a lze jej použít také jako velmi výkonnou a flexibilní kalkulačku.

K dispozici je několik dalších vývojových prostředí pro Python, raději ale používejte IDLE, tedy alespoň zpočátku. Své programy můžete také vytvářet v obyčejném textovém editoru dle vlastního výběru a ladit je pomocí volání funkce `print()`. Interpret jazyka lze vyvolat i bez uvedení programu, který chceme spustit. V takovém případě se interpret spustí v interaktivním režimu, ve kterém je možné zadávat příkazy jazyka Python a sledovat výsledky úplně stejně jako v okně Python Shell v prostředí IDLE, a to včetně téže výzvy příkazového řádku (`>>>`). Avšak s prostředím IDLE se mnohem snadněji pracuje, a proto byste měli s úryvky kódu experimentovat právě v tomto prostředí. U zde předkládaných krátkých interaktivních příkladů tedy předpokládáme, že se zadávají do interaktivního interpretu jazyka Python nebo do okna Python Shell v prostředí IDLE.



Obrázek 1.1: Okno Python Shell v prostředí IDLE

Nyní již tedy víme, jak se programy napsané v jazyku Python vytvářejí a spouštějí, z jazyka Python jsme se však zatím seznámili pouze s funkcí `print()`. V následující části své znalosti jazyka Python značně rozšíříme, takže budeme schopni vytvářet krátké, ale užitečné programy, což si také v poslední části této lekce vyzkoušíme.

Nádherné srdce jazyka Python

V této části se seznámíme s osmi klíčovými oblastmi jazyka Python a v další části si ukážeme, jak lze pomocí nich napsat několik malých, ale praktických programů. Budete-li mít při pročítání této části pocit, že něco chybí nebo že je výklad příliš rozvláčný, nakoukněte pomocí odkazů, obsahu nebo rejstříku na další stránky knihy. S největší pravděpodobností zjistíte, že prvek, který potřebujete, jazyk Python nejen obsahuje, ale že často nabízí poněkud zhuštěnější formy výrazu, který jsme si zde ukázali, a k tomu ještě mnohem více.

Oblast č. 1: Datové typy

Jednou ze základních věcí, které musí být schopen každý programovací jazyk, je reprezentovat prvky dat. Jazyk Python nabízí hned několik vestavěných datových typů, my se ale prozatím soustředíme pouze na dva z nich. Celočíselné hodnoty (kladná a záporná celá čísla) jsou v Pythonu reprezentovány pomocí typu `int` a řetězce (posloupnosti znaků ze znakové sady Unicode) pomocí typu `str`. Zde je několik příkladů literálů představujících celá čísla a řetězce:

```
-973
210624583337114373395836055367340864637790190801098222508621955072
0
"Někonečně náročné"
'Jakub Krtek'
'suprově αβγ€÷©'
''
```

Druhé výše uvedené číslo je 2^{217}. Velikost celých čísel v jazyku Python totiž není omezena pevně daným počtem bajtů, ale pouze pamětí počítače. Řetězce lze ohraničit dvojitými nebo jednoduchými uvozovkami, podstatné je, aby na obou koncích řetězce byl stejný druh uvozovek. Python používá pro řetězce znakovou sadu Unicode. Řetězce tedy nejsou omezeny jen na znaky ASCII, což je patrné z předposledního řetězce. Prázdný řetězec je prostě takový, který mezi ohraničujícími znaky neobsahuje vůbec nic.

Jazyk Python používá pro přístup k prvkům posloupnosti, jako je například řetězec, hranaté závorky (`[]`). Pokud se například nacházíme v okně Python Shell (ať už v interaktivním interpretu nebo v prostředí IDLE), můžeme zadat následující (výstup v okně Python Shell je vytištěn `normálním písmem`, zatímco vámi zadávaný vstup `zvýrazněn`):

```
>>> "Těžké časy"[6]
'č'
>>> "žirafa"[0]
'ž'
```

Okno Python Shell používá standardně pro výzvu svého příkazového řádku znaky `>>>`. Toto nastavení lze však jednoduše změnit. Syntaxi s hranatými závorkami lze použít u datových prvků libovolného datového typu, který představuje nějakou posloupnost, jako jsou řetězce a seznamy. Tato konzistence syntaxe je jedním z důvodů, proč je Python tak výjimečný. Všimněte si, že index v jazyku Python začíná vždy na hodnotě 0.

V jazyku Python jsou typ `str` a základní číselné typy *neměnitelné* (immutable), což znamená, že po nastavení již nelze jejich hodnotu změnit. Na první pohled to může vypadat jako poněkud zvláštní omezení, avšak pro syntaxe jazyka Python to není v praxi žádný problém. Jediným důvodem, proč se o tom zmiňujeme, je fakt, že znak na zadané pozici řetězce sice můžeme získat pomocí hranatých závorek, pro nastavení nového znaku je ale použít nemůžeme. (Je třeba poznamenat, že znak je v jazyku Python jednoduše řetězec s délkou 1.) Pro převod datového prvku jednoho typu na jiný můžeme použít syntaxi `datový_typ(prvek)`:

```
>>> int("45")
45
>>> str(912)
'912'
```

Převod pomocí `int()` je tolerantní vůči úvodnímu a koncovému prázdnému prostoru, takže stejný výsledek získáme také po vyhodnocení výrazu `int(" 45 ")`. Převod pomocí `str()` lze aplikovat na téměř jakýkoliv datový prvek. Podporu převodu pomocí `str()`, `int()` nebo dalších typů lze snadno zařídit také u našich vlastních datových typů, pokud takový převod dává smysl, což si vyzkoušíme v lekci 6. Pokud se převod nezdaří, vyvolá se výjimka. S ošetřováním výjimek se ve stručnosti seznámíme v části „Oblast č. 6“, přičemž výjimkám se budeme podrobně věnovat v lekci 4.

Řetězce a celá čísla jsou společně s ostatními vestavěnými datovými typy a některými datovými typy ze standardní knihovny jazyka Python obsahem lekce 2. V této lekci se podíváme také na operace, jež lze aplikovat na neměnitelné posloupnosti, jako jsou řetězce.

Oblast č. 2: Odkazy na objekty

Mělké a hloubkové kopírování ➢ 146

Jakmile máme nějaké datové typy, tak další věcí, kterou potřebujeme, jsou proměnné, v nichž je budeme uchovávat. Jazyk Python nezná proměnné jako takové, nabízí však tzv. *odkazy na objekty*. Co se týče neměnitelných objektů, jako jsou `int` a `str`, pak neexistuje žádný rozeznatelný rozdíl mezi proměnnou a odkazem na objekt. U proměnlivých objektů již rozdíl existuje, v praxi však nemá téměř žádný význam. Z tohoto důvodu budou pro nás oba termíny, proměnná a odkaz na objekt, znamenat totéž.

Nyní se podíváme na několik kratičkých příkladů, které si poté podrobně rozebereme.

```
x = "modrá"
y = "zelená"
z = x
```

Syntaxe má prostý tvar *`odkazNaObjekt = hodnota`*. Nic není třeba deklarovat předem, přičemž není nutné uvádět ani typ hodnoty. Jakmile totiž Python začne provádět první příkaz, vytvoří objekt `str` s textem „modrá“ a poté vytvoří odkaz na objekt s názvem `x`, který odkazuje na objekt `str`. Z praktického hlediska tedy můžeme říct, že „proměnné x byl přiřazen řetězec ‘modrá’“. Druhý příkaz je podobný. Třetí příkaz vytváří nový odkaz na objekt s názvem `z` a nastavuje jej tak, aby odkazoval na tentýž objekt, na který odkazuje objekt `x` (v tomto případě se jedná o objekt `str` obsahující text „modrá“).

Operátor = není stejný jako operátor přiřazení proměnné v některých jiných jazycích. Operátor = totiž sváže odkaz na objekt s objektem v paměti. Pokud odkaz na objekt již existuje, pak je jednoduše svázán znovu, tentokráte ale s objektem na pravé straně operátoru =. Pokud odkaz na objekt neexistuje, tak jej operátor = vytvoří.

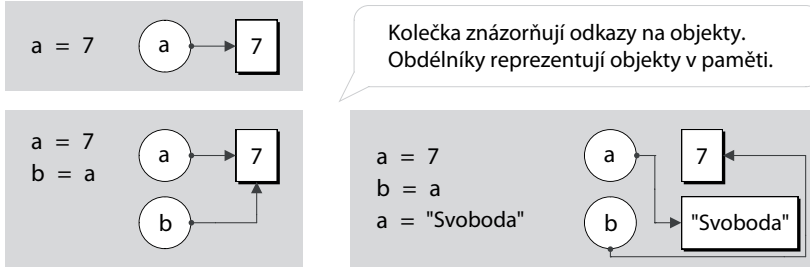
Nyní budeme pokračovat s naším příkladem a provedeme několik opětovných svázání. Jak jsme si řekli již dříve, komentáře začínají znakem # a pokračují až do konce řádku:

```
print(x, y, z) # vypíše: modrá zelená modrá
z = y
```

```
print(x, y, z) # vypíše: modrá zelená zelená
x = z
print(x, y, z) # vypíše: zelená zelená zelená
```

Po čtvrtém příkazu (`x = z`) se všechny tři odkazy na objekt odkazují na tentýž objekt `str`. Vzhledem k tomu, že na řetězec „modrá“ se již žádné objekty neodkazují, může jej Python uklidit z paměti.

Obrázek 1.2 schematicky znázorňuje vztah mezi objekty a odkazy na objekty.



Obrázek 1.2: Odkazy na objekty a objekty

Na názvy používané pro odkazy na objekty (říká se jim *identifikátory*) se vztahuje několik omezení. Nesmějí být totožné s žádným klíčovým slovem jazyka Python a musejí začínat písmenem nebo podtržítkem, za nímž následuje nula nebo více písmen, podtržítek nebo číslic, přičemž nesmí jít o znak představující prázdné místo. Délka není nijak omezena, přičemž písmena a číslice jsou definovány znakovou sadou Unicode, což mimo jiné zahrnuje též znaky a číslice ASCII („a“, „b“, ..., „z“, „A“, „B“, ..., „Z“, „0“, „1“, ..., „9“). U identifikátorů jazyka Python se rozlišuje velikost písmen, takže například `LIMIT`, `Limit` a `limit` jsou tři různé identifikátory. Další podrobnosti a několik malinko exotických ukázek naleznete v lekci 2.

Identifikátory a klíčová slova ➤ 58

Jazyk Python používá *dynamickou práci s typy*, což znamená, že odkaz na objekt lze kdykoliv opětovně svázat s jiným objektem (který může být odlišného datového typu). Jazyky, které jsou silně typované (jako například C++ nebo Java), povolují provádění pouze takových operací, které jsou pro dané datové typy definované. Jazyk Python toto omezení také aplikuje, nenazývá se ale silně typovaný jazyk, protože platné operace se mohou změnit. K tomu může dojít třeba v okamžiku, kdy se odkaz na objekt opětovně sváže s objektem jiného datového typu:

```
route = 866
print(route, type(route)) # vypíše: 866 <class 'int'>
route = "Sever"
print(route, type(route)) # vypíše: Sever <class 'str'>
```

Zde vytváříme nový odkaz na objekt s názvem `route` a nastavujeme jej tak, aby odkazoval na nový objekt `int` s hodnotou 866. V tomto okamžiku bychom mohli použít operátor `/`, poněvadž pro celá čísla je dělení platná operace. Odkaz na objekt s názvem `route` pak použijeme znovu tak, aby odkazoval na nový objekt `str` s hodnotou „Server“, přičemž objekt `int` je naplánován pro uklizení z paměti, protože se na něj již nic neodkazuje. V tomto okamžiku by použití operátoru `/` způsobilo vyvolání výjimky `TypeError`, neboť operátor `/` není platnou operací pro řetězec.

isinstance() ➤ 238

Funkce `type()` vrací datový typ (označovaný též jako „class“ – „třída“) zadaného datového prvku. Tato funkce je velice užitečná pro testování a ladění, v ostrém kódu by se však již objevit neměla, protože se dá nahradit lepší alternativou, o čemž se přesvědčíme v lekci 6.

Při experimentování s kódem jazyka Python uvnitř interaktivního interpretu nebo v okně Python Shell (např. v prostředí IDLE) pak prostý zápis názvu odkazu na objekt způsobí vypsání jeho hodnoty:

```
>>> x = "modrá"
>>> y = "zelená"
>>> z = x
>>> x
'modrá'
>>> x, y, z
('modrá', 'zelená', 'modrá')
```

To je mnohem pohodlnější řešení než neustálé volání funkce `print()`, funguje ale jen při interaktivní práci s Pythonem. Všechny programy a moduly, které napíšeme, musejí pro výstup nějakých hodnot používat `print()` nebo podobnou funkci. Všimněte si, že Python zobrazil poslední výstup v závorkách a oddělený čárkami. To označuje n-tici, což je uspořádaná, neměnitelná posloupnost objektů. K n-ticím se ještě dostaneme v následujících oblastech.

Oblast č. 3: Datové typy pro kolekce

Často je užitečné uchovávat celou kolekci datových prvků. Jazyk Python nabízí pro kolekce několik datových typů, které mohou uchovávat prvky. Mezi tyto datové typy patří mimo jiné také asociativní pole a množiny. My si zde ale představíme jen dva: `tuple` (n-tice) a `list` (seznam). Pomocí n-tic a seznamů jazyka Python lze uchovávat libovolný počet datových prvků libovolného datového typu. N-tice jsou neměnitelné, takže je po vytvoření již nelze změnit. Seznamy jsou měnitelné, takže lze snadno vkládat a odebírat prvky, kdykoli chceme.

N-tice se vytvářejí pomocí čárek (`,`), jak ukazují tyto příklady (od této chvíle již nebudeme zvýrazňovat vámi zadávanou část kódu):

```
>>> "Dánsko", "Finsko", "Norsko", "Švédsko"
('Dánsko', 'Finsko', 'Norsko', 'Švédsko')
>>> "one",
('one',)
```

Typ tuple ➤ 110

Python vypisuje n-tice uzavřené do závorek. Řada programátorů to napodobuje a při psaní kódu uzavírá n-ticové literály také vždy do závorek. Pokud máme jednoprvkovou n-tici a chceme použít závorky, musíme i tak uvést čárku – například `(1,)`. Prázdnou n-tici vytvoříme pomocí prázdných závorek, tedy `()`. Čárka se používá také k oddělení argumentů při volání funkce, takže pokud chceme jako argument předat n-ticový literál, musíme jej pro jednoznačnost uzavřít do závorek.

Tvorba a volání funkcí ➤ 44

Zde je několik ukázkových seznamů:

```
[1, 4, 9, 16, 25, 36, 49]
['alfa', 'bravo', 'charlie', 'delta', 'echo']
```

```
['zebra', 49, -879, 'hrabáč', 200]
[]
```

Jedna z možností pro vytvoření seznamu, kterou jsme si již ukázali, spočívá v použití hranatých závorek ([]). Později se podíváme na další možnosti. Čtvrtý z výše uvedených seznamů je prázdný seznam.

Typ list ➤ 115

Ve skutečnosti to funguje tak, že seznamy ani n-tice neuchovávají datové prvky, ale odkazy na objekty. Při vytváření seznamů a n-tic (a také při vkládání prvků v případě seznamů) přijímají tyto kolekce kopie zadávaných odkazů na objekty. V případě literálových prvků, jako jsou čísla a řetězce, se v paměti vytvoří a vhodným způsobem inicializuje objekt příslušného datového typu a poté se vytvoří odkaz na objekt odkazující na daný objekt, přičemž do seznamu nebo n-tice se uloží právě tento odkaz na objekt.

Jako cokoliv v jazyku Python také datové typy pro kolekce jsou objekty, takže datové typy pro kolekce lze vnořovat do dalších datových typů pro kolekce, čímž lze například vytvořit seznamy seznamů. V některých situacích je skutečnost, že seznamy, n-tice a většina ostatních datových typů pro kolekce jazyka Python uchovávají místo objektů odkazy na objekty, velice podstatná. Více se tomuto tématu budeme věnovat v lekci 3.

Mělké a hloubkové kopírování ➤ 146

Při procedurálním programování se volají funkce, kterým se často předávají datové prvky jako argumenty. Například s funkcí print() jsme se již setkali. Další v Pythonu často používanou funkcí je funkce len(), která jako svůj argument přijímá jediný datový prvek a vrací jeho „délku“ jako objekt int. Zde je několik příkladů volání funkce len():

```
>>> len(("jedna",))
1
>>> len([3, 5, 1, 2, "pauza", 5])
6
>>> len("automaticky")
11
```

N-tice, seznamy a řetězce jsou datové typy, u nichž má smysl mluvit o velikost (Sized), a proto lze datové prvky libovolného z těchto datových typů předat funkci len(). (Při předání funkci len() datového prvku, který nezná pojem velikost, dojde k vyvolání výjimky.)

třída Sized ➤ 369

Všechny datové prvky jazyka Python jsou *objekty* (nazývané též *instance*) určitého datového typu (označovaného též jako *třída*). Pro nás budou oba termíny *datový typ* a *třída* znamenat totéž. Jediným podstatným rozdílem mezi objektem a holými prvky s daty nabízenými v některých jiných jazycích (např. vestavěné číselné typy jazyka C++ nebo Java) spočívá v tom, že objekt může mít *metody*. Metoda je v podstatě funkce, která se volá pro určitý objekt. Například typ list má metodu append(), která připojí zadaný objekt k seznamu:

```
>>> x = ["zebra", 49, -879, "hrabáč", 200]
>>> x.append("další")
>>> x
['zebra', 49, -879, 'hrabáč', 200, 'další']
```

Objekt `x` ví, že je seznamem (všechny objekty jazyka Python znají svůj vlastní datový typ), takže datový typ nemusíme explicitně uvádět. V implementaci metody `append()` bude prvním argumentem samotný objekt `x`. O to se stará Python zcela automaticky v rámci své syntaktické podpory pro metody.

Metoda `append()` mění původní seznam. To je možné díky tomu, že seznamy jsou měnitelné. Je to také potenciálně efektivnější než vytvoření nového seznamu s původními a nově přidaným prvkem, s nímž by se opětovně svázal náš odkaz na objekt. To platí zvláště pro velmi dlouhé seznamy.

V procedurálním jazyku lze téhož dosáhnout pomocí metody `append()` datového typu `list` (což je naprosto platná syntaxe jazyka Python):

```
>>> list.append(x, "extra")
>>> x
['zebra', 49, -879, 'hrabáč', 200, 'další', 'extra']
```

Zde specifikujeme datový typ a jeho metodu, které jako první argument předáváme datový prvek tohoto datového typu, na němž chceme metodu zavolat. Dále pokračujeme jakýmikoli dalšími argumenty metody. (S ohledem na dědičnost existuje mezi těmito dvěma syntaxemi drobný rozdíl. V praxi se nejčastěji používá první z uvedených forem. Dědičnost budeme probírat v lekci 6.)

Pokud vám objektově orientované programování nic neříká, pak vám to může na první pohled připadat trošku zvláštní. Prozatím se spokojte s tím, že Python nabízí konvenční funkce volané ve tvaru *`názevFunkce(argumenty)`* a metody, které se volají stylem *`názevObjektu.názevMetody(argumetny)`*. (Objektově orientovanému programování se budeme věnovat v lekci 6.)

Operátor tečka („přístup k atributům“) se používá pro přístup k atributům objektu. Atributem může být libovolný druh objektu, i když zatím jsme se ukázali jen atributy ve formě metod. Vzhledem k tomu, že atributem může být objekt, který má také atributy, které zase mohou mít svoje atributy (a tak pořád dál), můžeme pro přístup k určitému atributu použít tolik operátorů tečka, kolik jen potřebujeme.

Typ list má mnoho dalších metod, včetně metody `insert()`, která se používá k vložení prvku na zadanou pozici, a metody `remove()`, která odstraní prvek na zadané pozici. Jak jsme si řekli již dříve, indexy jazyka Python začínají vždy od nuly.

Viděli jsme, že pomocí operátoru `[]` můžeme získat znaky z řetězce, a řekli jsme si, že tento operátor lze použít s libovolnou posloupností. Seznamy jsou posloupnosti, a proto můžeme provádět následující:

```
>>> x
['zebra', 49, -879, 'hrabáč', 200, 'další', 'extra']
>>> x[0]
'zebra'
>>> x[4]
200
```

N-tice jsou také posloupnosti, takže pokud by byl objekt `x` n-ticí, mohli bychom získat jeho prvky pomocí hranatých závorek úplně stejně jako v případě seznamu `x`. Avšak vzhledem k tomu, že seznamy jsou měnitelné (na rozdíl od řetězců a n-tic), můžeme pomocí operátoru `[]` prvky seznamu také nastavovat:

```
>>> x[1] = "čtyřicet devět"
>>> x
['zebra', 'čtyřicet devět', -879, 'hrabáč', 200, 'další', 'extra']
```

Pokud použijeme index, který je mimo rozsah, dojde k vyvolání výjimky (s výjimkami se stručně seznámíme v oblasti č. 5 a podrobně se jim budeme věnovat v lekci 4).

Termín posloupnost jsme použili již několikrát, přičemž jsme spoléhali na neformální porozumění jeho významu – v tomto budeme prozatím pokračovat i nadále. Nicméně jazyk Python přesně definuje, jaké funkce musí posloupnost podporovat, a podobně definuje, jaké funkce musí nabízet objekt podporující velikost. Tato „pravidla" se samozřejmě týkají mnoha další kategorií, do nichž mohou spadat určité datové typy, což si ukážeme v lekci 8.

Seznamy, n-tice a ostatní vestavěné datové typy pro kolekce jazyka Python jsou obsahem lekce 3.

Oblast č. 4: Logické operátory

Jedním ze základních prvků jakéhokoliv programovacího jazyka jsou jeho logické operace. Jazyk Python nabízí čtyři skupiny logických operací a my si zde představíme základy každé z nich.

Operátor identita

Vzhledem k tomu, že všechny proměnné jazyka Python jsou ve skutečnosti odkazy na objekty, může být někdy užitečné ptát se, zda dva či více odkazů na objekty odkazují na tentýž objekt. K tomuto účelu slouží binární operátor `is`, který vrací hodnotu `True`, pokud se odkaz na objekt na levé straně odkazuje na stejný objekt jako odkaz na objekt na pravé straně. Zde je několik příkladů:

```
>>> a = ["retence", 3, None]
>>> b = ["retence", 3, None]
>>> a is b
False
>>> b = a
>>> a is b
True
```

Všimněte si, že obvykle nemá smysl používat operátor `is` k porovnání objektů `int`, `str` a většiny ostatních datových typů, protože téměř vždy chceme porovnat jejich hodnoty. Ve skutečnosti může vést použití operátoru `is` k porovnání datových prvků k neintuitivním výsledům, což je patrné z předchozího příkladu, kde jsme sice na začátku nastavili `a` a `b` na seznam se stejnými hodnotami, avšak samotné seznamy jsou uloženy jako samostatné objekty typu `list`, a proto při prvním použití operátoru `is` obdržíme hodnotu `False`.

Jednou z výhod porovnávání na základě identity je jeho rychlost. Ta je dána tím, že není nutné prozkoumávat samotné odkazované objekty. Operátoru `is` totiž stačí porovnat pouze paměťové adresy objektů – stejná adresa znamená stejný objekt.

Operátor `is` se nejčastěji používá k porovnání datového prvku s vestavěným objektem `None`, který se většinou používá pro označení neznámé nebo neexistující hodnoty:

```
>>> a = "něco"
>>> b = None
>>> a is not None, b is None
(True, True)
```

K obrácení testu na základě identity se používá `is not`.

Smyslem operátoru identita je zjistit, zda se dva odkazy na objekt odkazují na tentýž objekt, nebo zda je daný objekt `None`. Chceme-li porovnat hodnoty objektů, musíme použít porovnávací operátory.

Porovnávací operátory

Jazyk Python nabízí standardní skupinu binárních porovnávacích operátorů s očekávanou sémantikou: < menší než, <= menší nebo rovno, == rovná se, != nerovná se, >= větší nebo rovnost a > větší než. Tyto operátory porovnávají hodnoty objektů, neboli objekty, na které se ukazují odkazy na objekt použité v porovnávání. Zde je několik příkladů zapsaných do okna Python Shell:

```
>>> a = 2
>>> b = 6
>>> a == b
False
>>> a < b
True
>>> a <= b, a != b, a >= b, a > b
(True, True, False, False)
```

U celých čísel funguje vše tak, jak bychom očekávali. Podobně funguje také porovnávání řetězců:

```
>>> a = "mnoho cest"
>>> b = "mnoho cest"
>>> a is b
False
>>> a == b
True
```

Porovnávání řetězců ➤ 73

Ačkoliv jsou `a` a `b` odlišné objekty (mají odlišné identity), mají stejné hodnoty, a proto jsou při porovnání stejné. Dávejte si však pozor, protože Python používá pro reprezentaci řetězců znakovou sadu Unicode, a proto může být porovnávání řetězců, jež obsahují znaky mimo kódování ASCII, mnohem komplikovanější, než jak by se na první pohled mohlo zdát (tomuto problému se budeme plně věnovat v lekci 2).

Porovnání dvou řetězců nebo čísel na základě identity (např. pomocí `a is b`) vrátí v některých případech hodnotu `True`, přestože jsme oba objekty přiřadili samostatně. To je dáno tím, že některé implementace jazyka Python opětovně použijí kvůli lepší efektivitě stejný objekt (protože hodnota je stejná a neměnitelná). Z toho plyne, že operátory `==` a `!=` se mají používat k porovnání hodnot a operátor `is` a `is not` pouze k porovnání s objektem `None` nebo pokud skutečně chceme zjistit, zda jsou stejné odkazy na objekty, a ne objekty samotné.

Jednou z krásných vlastností porovnávacích operátorů jazyka Python je možnost jejich řetězení:

```
>>> a = 9
>>> 0 <= a <= 10
True
```

Jedná se o hezčí způsob testování, zda je zadaný datový prvek v určitém rozsahu, než dvě samostatná porovnání spojená logickým operátorem `and`, což je nezbytné ve většině ostatních jazyků. Další předností je to, že se datový prvek vyhodnotí pouze jednou (protože se ve výrazu vyskytuje pouze jednou), což může mít značný význam v případě, kdy je výpočet hodnoty datového prvku drahý nebo pokud má přístup k datovému prvku za následek nějaké vedlejší efekty.

Díky „síle“ jazyka Python v oblasti dynamické práce s typy způsobí porovnání, které nedává smysl, vyvolání výjimky:

```
>>> "tři" < 4
Traceback (most recent call last):
...
TypeError: unorderable types: str() < int()
```

Při vyvolání výjimky, která není ošetřena, vypíše Python zpětné volání společně s chybovou zprávou výjimky. Pro srozumitelnost jsme část se zpětným voláním nahradili třemi tečkami.* Ke stejné výjimce typu `TypeError` by došlo také v případě porovnání `"3" < 4`, protože Python se nepokouší uhádnout naše záměry. Správně bychom měli provést explicitní převod (např. `int("3") < 4`) nebo použít porovnatelné typy, tedy buď jen čísla, nebo jen řetězce.

Chyby za běhu programu ➤ 402

Jazyk Python usnadňuje vytváření vlastních datových typů, které lze pěkně integrovat tak, že můžeme kupříkladu vytvořit vlastní číselný typ, který by byl schopen účastnit se porovnávání s vestavěným typem `int`, ne však s řetězci nebo s jinými nečíselnými typy.

Alternativní typ `FuzzyBool` ➤ 250

Operátor příslušnosti

U datových typů, které jsou posloupnostmi nebo kolekcemi, jako například řetězce, seznamy nebo ntice, můžeme testovat příslušnost pomocí operátoru `in` a nepříslušnost pomocí operátoru `not in`:

```
>>> p = (4, "žába", 9, -33, 9, 2)
>>> 2 in p
True
>>> "pes" not in p
True
```

* Zpětné volání je seznam všech volání provedených od okamžiku, kdy došlo k vyvolání neošetřené výjimky, až po vrchol zásobníku volání.

U seznamů a n-tic používá operátor `in` lineární vyhledávání, které může být pomalé u velkých kolekcí (desítky tisíc a více prvků). Na druhou stranu je operátor `in` velmi rychlý při použití na slovník nebo množinu (oba tyto datové typy pro kolekce budeme probírat v lekci 3). Zde je příklad použití operátoru `in` s řetězcem:

```
>>> phrase = "Pestrobarevná kalkulačka"
>>> "k" in phrase
True
>>> "barev" in phrase
True
```

V případě řetězců lze operátor příslušnosti použít i pro testování podřetězců libovolné délky (jak jsme si řekli již dříve, znak je v podstatě řetězec délky 1).

Logické operátory

Jazyk Python nabízí tři logické (neboli booleovské) operátory: `and`, `or` a `not`. Operátory `and` a `or` používají logiku zkráceného vyhodnocování a vracejí operand, který rozhodl o výsledku – nevracejí logickou hodnotu (pokud tedy samy operandy neobsahují logickou hodnotu). Podívejme se, co to znamená v praxi:

```
>>> five = 5
>>> two = 2
>>> zero = 0
>>> five and two
2
>>> two and five
5
>>> five and zero
0
```

Pokud se výraz objeví v kontextu logického výrazu, pak je výsledek vyhodnocen jako logická hodnota, takže předchozí výrazy by se například v příkazu `if` vyhodnotily na `True`, `True` a `False`.

```
>>> nought = 0
>>> five or two
5
>>> two or five
2
>>> zero or five
5
>>> zero or nought
0
```

Operátor `or` je podobný. Zde bychom v kontextu logického výrazu obdrželi výsledky `True`, `True`, `True` a `False`.

Unární operátor `not` vyhodnotí své argumenty v kontextu logického výrazu a vždy vrátí logickou hodnotu, takže v případě výše uvedeného příkladu by se výraz `not (zero or nought)` vyhodnotil na `True` a výraz `not two` na `False`.

Oblast č. 5: Příkazy pro řízení toku programu

Již dříve jsme si řekli, že příkazy uvedené v souboru `.py` se provádějí jeden za druhým, přičemž se začíná prvním příkazem a pokračuje se řádek po řádku. Tok programu lze odklonit voláním funkcí či metody nebo řídící strukturou, jako je podmíněná větev nebo příkaz cyklu. Tok programu je též odkloněn při vyvolání výjimky.

V této podčásti se podíváme na příkaz `if` jazyka Python a na jeho cykly `while` a `for`, přičemž o funkcích si více řekneme v Oblasti č. 8 a metody ponecháme na lekci 6. Kromě toho se podíváme na naprosté základy ošetřování výjimek (tomuto tématu se budeme plně věnovat v lekci 4). Nejdříve si ale uděláme jasno v terminologii.

Logický výraz je cokoliv, co lze vyhodnotit na logickou hodnotu (`True` nebo `False`). V jazyku Python se takový výraz vyhodnotí na hodnotu `False`, jedná-li se o předdefinovanou konstantu `False`, o speciální objekt `None`, o prázdnou posloupnost nebo kolekci (např. prázdný řetězec, seznam či n-tice) nebo o číselný datový prvek s hodnotou 0. Cokoliv jiného se vyhodnotí na hodnotu `True`. Při vytváření vlastních datových typů (např. v lekci 6) se můžeme sami rozhodnout, co mají v kontextu logického výrazu vrátit.

V řeči jazyka Python se bloku kódu, což je posloupnost jednoho či více příkazů, nazývá *sada* (suite). Některé prvky syntaxe jazyka Python vyžadují přítomnost sady, a proto Python nabízí klíčové slovo `pass`, které představuje příkaz, který neudělá nic a který lze použít všude tam, kde je vyžadována sada (nebo kde chceme říci, že jsme daný případ zvážili), ale kde není potřeba provést žádné zpracování.

Příkaz if

Obecná syntaxe pro příkaz `if` jazyka Python vypadá takto:*

```
if logický_výraz1:
    sada1
elif logický_výraz2:
    sada2
...
elif logický_výrazN:
    sadaN
else:
    jiná_sada
```

Klauzulí `elif` může být nula či více, přičemž finální klauzule `else` je volitelná. Pokud nás určitý případ zajímá, ale nechceme v případě jeho výskytu nic provádět, můžeme pro sadu takové větve použít klíčové slovo `pass`.

* V této knize používáme výpustek (...) pro reprezentaci řádků, které jsme záměrně skryli.

První věcí, která zarazí programátory zvyklé na jazyk C++ nebo Java, je nepřítomnost jednoduchých či složených závorek. Další podstatnou věcí je dvojtečka. Ta je součástí syntaxe a zpočátku se na ni snadno zapomíná. Dvojtečky se používají také u klauzulí `else`, `elif` a v podstatě na libovolném jiném místě, za nímž následuje sada.

Na rozdíl od jiných programovacích jazyků používá Python odsazení k označení své blokové struktury. Někteří programátoři to nemají rádi, zvláště pokud to ještě nevyzkoušeli, a někteří se k tomuto problému staví až příliš afektovaně. Na druhou stranu se na to dá zvyknout za několik dnů a za několik týdnů či měsíců se vám bude kód bez závorek jevit mnohem hezčí a čitelnější.

Sady se označují pomocí odsazení, a proto se můžeme zeptat, o jaké odsazení se vlastně jedná. Směrnice ohledně stylu programování v jazyku Python doporučují použít pro každou úroveň odsazení čtyři mezery (bez tabulátorů). Většinu moderních textových editorů lze nastavit tak, aby se o to postaraly zcela automaticky (což splňuje editor IDLE a většina ostatních editorů s podporou jazyka Python). Python bude fungovat skvěle s libovolným počtem mezer či tabulátorů nebo se směsicí obou, bude-li používané odsazení konzistentní. V této knize budeme dodržovat oficiální směrnice jazyka Python.

Zde je velmi jednoduchá ukázka příkazu `if`:

```
if x:
    print("x je nenulové")
```

V tomto případě se sada (volání funkce `print()`) provede tehdy, pokud se podmínka (`x`) vyhodnotí na hodnotu `True`.

```
if lines < 1000:
    print("malé")
elif lines < 10000:
    print("střední")
else:
    print("velké")
```

Tohle už je malinko propracovanější příkaz `if`, který vypíše slovo popisující číslo uložené v proměnné `lines`.

Příkaz while

Příkaz `while` se používá k provedení sady nulakrát či vícekrát, přičemž tento počet závisí na stavu logického výrazu cyklu `while`. Zde je syntaxe:

```
while logický_výraz:
    sada
```

Úplná syntaxe cyklu `while` je ve skutečnosti mnohem sofistikovanější, poněvadž podporuje příkazy `break` a `continue` a také volitelnou klauzuli `else`, které se budeme věnovat v lekci 4. Příkaz `break` přepne tok programu na příkaz následující za nejvnitřnějším cyklem, v němž se daný příkaz `break` nachází, což znamená, že z tohoto cyklu vyskočí. Příkaz `continue` přepne tok programu na začátek

cyklu. Oba příkazy `break` a `continue` se obvykle používají uvnitř příkazů `if` k podmíněné změně chování cyklu:

```
while True:
    item = get_next_item()
    if not item:
        break
    process_item(item)
```

Tento cyklus `while` má velmi typickou strukturu a běží do té doby, dokud jsou k dispozici prvky na zpracování (`get_next_item()` a `process_item()` jsou naše vlastní funkce definované na jiném místě. V tomto příkladu obsahuje sada příkazu `while` příkaz `if`, který sám o sobě má další sadu (kterou musí mít) obsahující jediný příkaz `break`.

Příkaz for ... in

Cyklus `for` jazyka Python používá opětovně klíčové slovo `in` (které se v jiném kontextu chová jako operátor příslušnosti) a má následující syntaxi:

```
for proměnná in iterovatelný_objekt:
    sada
```

Podobně jako cyklus `while` také cyklus `for` podporuje oba příkazy `break` a `continue` a má také volitelnou klauzuli `else`. Proměnná je nastavena tak, aby postupně odkazovala na každý objekt v iterovatelném objektu, což je libovolný datový typ, který lze procházet, což zahrnuje řetězce (prochází se po jednotlivých znacích), seznamy, n-tice a další datové typy pro kolekce jazyka Python.

```
for country in ["Dánsko", "Finsko", "Norsko", "Švédsko"]:
    print(country)
```

Zde používáme velice zjednodušený přístup k vypisování seznamu zemí. V praxi se mnohem častěji používá proměnná:

```
countries = ["Dánsko", "Finsko", "Norsko", "Švédsko"]
for country in countries:
    print(country)
```

Ve skutečnosti lze celý seznam (nebo n-tici) vypsat pomocí funkce `print()` i přímo (například `print(countries)`), ale často se kvůli lepší kontrole formátování upřednostňuje výpis kolekce pomocí cyklu `for` (nebo pomocí seznamové komprehenze, které se budeme věnovat později):

Seznamové komprehenze ➤ 120

```
for letter in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
    if letter in "AEIOU":
        print(letter, "je samohláska")
    else:
        print(letter, "je souhláska")
```

V tomto úryvku je první použití klíčového slova `in` součástí příkazu `for`, přičemž proměnná `letter` nabývá v každé iteraci cyklu hodnot `"A"`, `"B"` a tak dále až po `"Z"`. Na druhém řádku úryvku používáme opět `in`, ovšem tentokrát jako operátor testující příslušnost. Všimněte si také, že zde máme vnořené sady. Sadou cyklu `for` je příkaz `if ... else` a obě větvě `if` a `else` mají své vlastní sady.

Základní ošetřování výjimek

Řada funkcí a metod jazyka Python hlásí chyby nebo jiné důležité události vyvoláním určité výjimky. Výjimka je objekt stejně jako jakýkoliv jiný objekt jazyka Python a při převodu na řetězec (např. při výpisu) obdržíme textovou zprávu výjimky. Jednoduchý tvar syntaxe pro ošetřování výjimek vypadá takto:

```
try:
    sada_try
except výjimka1 as proměnná1:
    sada_výjimky1
...
except výjimkaN as proměnnáN:
    sada_výjimkyN
```

Část `as` *`proměnná`* je volitelná. Může nás totiž zajímat pouze to, že byla vyvolána určitá výjimka, a ne už text její zprávy.

Úplná syntaxe je mnohem sofistikovanější. Například každá klauzule `except` může ošetřit více výjimek a dále tu je volitelná klauzule `else`. Všechny tyto prvky budeme probírat v lekci 4.

Celé to funguje takto. Pokud se všechny příkazy v sadě bloku `try` provedou bez vyvolání výjimky, bloky `except` se přeskočí. Pokud dojde uvnitř bloku `try` k vyvolání výjimky, předá se řízení okamžitě do sady odpovídající první vyhovující výjimce. To znamená, že žádný příkaz v sadě, který následuje za tím, jenž způsobil výjimku, se již neprovede. Pokud k tomu dojde a navíc je uvedena také část `as` *`proměnná`*, pak uvnitř sady ošetřující výjimku bude tato proměnná odkazovat na objekt výjimky.

Chyby za běhu programu ➤ 402

Pokud v bloku ošetřujícím výjimku dojde k výjimce nebo pokud se vyvolá výjimka, která neodpovídá žádnému z bloků `except`, pak se Python podívá po odpovídajícím bloku `except` v dalším obklopujícím oboru platnosti. Hledání vhodné obsluhy výjimky probíhá směrem k vnějším oborům platnosti podle zásobníku volání až do té doby, dokud není nalezena shoda a výjimka obsloužena. Pokud není shoda nalezena, program se ukončí s neošetřenou výjimkou. V takovém případě vypíše Python zpětné volání společně s textovou zprávou výjimky.

Zde je příklad:

```
s = input("zadejte celé číslo: ")
try:
  i = int(s)
  print("zadáno platné celé číslo:", i)
except ValueError as err:
  print(err)
```

Pokud uživatel zadá „3.5“, vypíše se toto:

```
invalid literal for int() with base 10: '3.5'
```

Pokud ale zadá „13“, vypíše se:

```
valid integer entered: 13
```

Mnoho knih považuje ošetřování výjimek za pokročilé téma a odkládá je na co nejpozdější dobu. Ale vyvolávání a zvláště ošetřování výjimek je vzhledem ke způsobu fungování Pythonu naprosto zásadní, takže od této chvíle jej budeme využívat. A jak uvidíme, díky používání obsluh výjimek může být kód mnohem čitelnější, protože se „výjimečné“ případy oddělí od vlastního zpracování, které nás ve skutečnosti jako jediné zajímá.

Oblast č. 6: Aritmetické operátory

Python nabízí kompletní sadu aritmetických operátorů, včetně binárních operátorů pro čtyři základní matematické operace: operátor + (sčítání), operátor - (odčítání), operátor * (násobení) a operátor / (dělení). Kromě toho řada datových typů jazyka Python lze použít s operátory rozšířeného přiřazení, jako je += a *=. Operátory +, - a * se v případě operandů jakožto celých čísel chovají dle očekávání:

```
>>> 5 + 6
11
>>> 3 - 7
-4
>>> 4 * 8
32
```

Všimněte si, že operátor - může být použit jako unární (negace) nebo binární operátor (odčítání), což je obvyklé u většiny programovacích jazyků. V oblasti dělení však jazyk Python vyčnívá z davu:

```
>>> 12 / 3
4.0
>>> 3 / 2
1.5
```

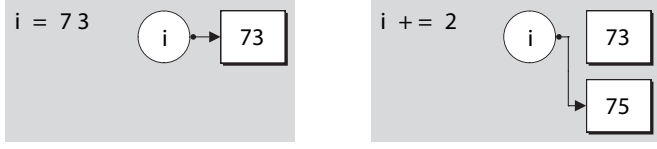
Operátor dělení nevrací celé, ale desetinné číslo. Většina ostatních jazyků vrátí celé číslo vytvořené ořezáním desetinné části. Pokud potřebujeme celočíselný výsledek, můžeme jej vždy převést pomocí `int()` (nebo pomocí ořezávacího operátoru dělení `//`, k němuž se dostaneme později).

Číselné operace a funkce ➤ 61

```
>>> a = 5
>>> a
5
>>> a += 8
>>> a
13
```

Na první pohled výše uvedené příkazy ničím nepřekvapí, především pak ty, kteří znají jazyky podobné jazyku C. V takovýchto jazycích představuje rozšířené přiřazení zkratku pro přiřazení výsledku operace (např. `a += 8` je v podstatě totéž jako `a = a + 8`). Zde ovšem existují dvě podstatné delikátnosti, jedna specifická pro Python a jedna pro rozšířené operátory v libovolném jazyku.

První věcí, kterou je třeba si zapamatovat, tkví v tom, že datový typ `int` je neměnitelný, což znamená, že po jeho přiřazení již nemůže být jeho hodnota změněna. Při použití operátoru rozšířeného přiřazení na neměnitelný objekt se tedy v pozadí odehraje to, že se provede daná operace a vytvoří se objekt uchovávající výsledek. Poté se cílový odkaz na objekt opětovně sváže tak, aby místo původního objektu odkazoval na objekt s výsledkem. V případě příkazu `a += 8` Python tedy nejdříve vypočítá `a + 8`, výsledek uloží do nového objektu `int` a poté opětovně sváže `a` tak, aby odkazovalo na tento nový objekt `int` (a pokud na původní objekt již neukazují žádné odkazy na objekt, tak jej naplánuje pro úklid z paměti). Tuto situaci znázorňuje obrázek 1.3.



Obrázek 1.3: Rozšířené přiřazení neměnitelného objektu

Druhá delikátnost spočívá v tom, že `a` *`operátor`*`= b` není úplně totéž jako `a = a` *`operátor`* `b`. Rozšířená verze vyhledá hodnotu a pouze jednou, je tedy potenciálně rychlejší. Je-li kromě toho `a` komplexní výraz (např. seznam prvků s výpočtem pozice indexu – `items[offset + index]`), pak může být použití rozšířené verze méně náchylné k chybám, protože v případě změny výpočtu není nutné zasahovat do dvou výrazů, ale úpravy stačí provést pouze jednou.

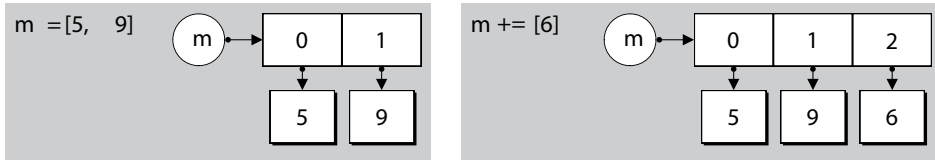
Jazyk Python přetěžuje (tj. opětovně používá pro jiný datový typ) operátory + a += pro řetězce i seznamy, přičemž první znamená zřetězení a druhý připojení pro řetězce a rozšíření (připojení dalšího seznamu) pro seznamy:

```
>>> name = "Jan"
>>> name + "Dudek"
'JanDudek'
>>> name += " Dudek"
>>> name
'Jan Dudek'
```

Podobně jako celá čísla také řetězce jsou neměnitelné, takže při použití operátoru += se vytvoří nový řetězec, s nímž se opětovně sváže odkaz na objekt na levé straně výrazu, tedy úplně stejně jako v případě celých čísel. Seznamy podporují tutéž syntaxi, ovšem v pozadí to probíhá trošku jinak:

```
>>> seeds = ["sezam", "slunečnice"]
>>> seeds += ["dýně"]
>>> seeds
['sezam', 'slunečnice', 'dýně']
```

Seznamy jsou měnitelné, a proto se při použití operátoru += modifikuje původní seznam bez nutnosti opětovného svazování. Tuto situaci zachycuje obrázek 1.4.



Obrázek 1.4: Rozšířené přiřazení měnitelného objektu

Vzhledem k tomu, že syntaxe jazyka Python chytře skrývá odlišnosti mezi měnitelnými a neměnitelnými datovými typy, můžeme se ptát, proč vlastně potřebuje oba druhy? Důvody se povětšinou týkají výkonu. Implementace neměnitelných typů je ve srovnání s měnitelnými typy potenciálně mnohem efektivnější (protože se nikdy nemění). Kromě toho některé datové typy pro kolekce (např. množiny) mohou pracovat pouze s neměnitelnými typy. Na druhou stranu s měnitelnými typy se pohodlněji pracuje. Na tuto odlišnost budeme upozorňovat všude tam, kde má svůj význam (např. v lekci 4 při diskuzi ohledně nastavování výchozích argumentů pro vlastní funkce, v lekci 3 při výkladu seznamů, množin a některých dalších datových typů a opět v lekci 6, kde si ukážeme, jak vytvářet naše vlastní datové typy).

V případě seznamu musí být operand na pravé straně operátoru += iterovatelný, jinak dojde k vyvolání výjimky:

Chyby za běhu programu ➢ 402

```
>>> seeds += 5
Traceback (most recent call last):
...
TypeError: 'int' object is not iterable
```

Seznam lze náležitě rozšířit pomocí iterovatelného objektu, jako je kupříkladu seznam:

```
>>> seeds += [5]
>>> seeds
['sezam', 'slunečnice', 'dýně', 5]
```

Iterovatelný objekt použitý pro rozšíření seznamu může mít samozřejmě více než jeden prvek:

```
>>> seeds += [9, 1, 5, "mák"]
>>> seeds
['sezam', 'slunečnice', 'dýně', 5, 9, 1, 5, 'mák']
```

Připojení obyčejného řetězce (např. `"durian"`) místo seznamu obsahujícího řetězec (`["durian"]`) vede k logickému, i když možná překvapivému výsledku:

```
>>> seeds = ["sezam", "slunečnice", "dýně"]
>>> seeds += "durian"
>>> seeds
['sezam', 'slunečnice', 'dýně', 'd', 'u', 'r', 'i', 'a', 'n']
```

Operátor += rozšířil seznam připojením každého prvku zadaného iterovatelného objektu. Řetězec je iterovatelný, a proto se každý znak zadaného řetězce připojí samostatně. Pokud bychom použili metodu seznamu s názvem `append()`, pak by se argument vždy připojil jako jediný prvek.

Oblast č. 7: Vstup a výstup

Pro psaní opravdu užitečných programů musíme být schopni číst vstup (např. od uživatele z konzoly nebo ze souborů) a vytvářet výstup (ať už do konzoly nebo do souborů). Vestavěnou funkci Pythonu `print()` jsme již používali, její podrobnější popis však odložíme až do lekce 4. V této podčásti se soustředíme na vstup a výstup v rámci konzoly a pro čtení a zápis souborů použijeme přesměrování shellu.

Python nabízí pro přijímání vstupu od uživatele vestavěnou funkci `input()`. Tato funkce přijímá volitelný řetězcový argument (který vypíše na konzolu). Poté počká, až uživatel zadá odpověď, kterou ukončí stiskem klávesy Enter (nebo Return). Pokud uživatel nenapíše žádný text, ale jen stiskne klávesu Enter, funkce `input()` vrátí prázdný řetězec. V opačném případě vrátí řetězec obsahující to, co uživatel zadal, ovšem bez znaků ukončujících řádek.

Zde je náš první úplný „užitečný" prográmek, který staví na řadě již dříve probraných oblastí. Jedinou novinkou je funkce `input()`:

```
print("Zadávejte celá čísla, za každým Enter; nebo jen Enter pro ukončení")

total = 0
count = 0

while True:
    line = input("číslo: ")
    if line:
        try:
            number = int(line)
        except ValueError as err:
            print(err)
            continue
        total += number
        count += 1
    else:
        break

if count:
    print("počet =", count, "celkem =", total, "průměr =", total / count)
```

Příklady ke knize ➢ 14

Uvedený program (v příkladech ke knize v souboru `sum1.py`) má pouze 17 prováděných řádků. Takto vypadá jeho typické spuštění:

```
Zadávejte celá čísla, za každým Enter; nebo jen Enter pro ukončení
číslo: 12
číslo: 7
číslo: 1x
invalid literal for int() with base 10: '1x'
číslo: 15
číslo: 5
```

```
číslo:
počet = 4 celkem = 39 průměr = 9.75
```

I když je tento program velice krátký, je poměrně robustní. Pokud uživatel zadá řetězec, který nelze převést na číslo, problém zachytí obsluha výjimky, která vypíše vhodnou zprávu a předá řízení na začátek cyklu. Poslední příkaz `if` zajistí, že pokud uživatel nezadá žádné číslo, souhrnné údaje se nevypíšou, takže nedojde k dělení nulou.

Práci se soubory se budeme plně věnovat v lekci 7. Nyní můžeme vytvářet soubory přesměrováním výstupu funkce `print()` z konzoly:

```
C:\>test.py > results.txt
```

Tento příkaz způsobí, že se výstup obyčejných volání funkce `print()` ve smyšleném programu `test.py` zapíše do souboru `results.txt`. Tato syntaxe funguje v konzole Windows (většinou) a v unixových konzolách. V případě Windows musíme zapsat `C:\Python31\python.exe test.py > results.txt`, pokud je výchozí verzí Python 2 nebo pokud se konzoly týká chyba s asociací souborů. V opačném případě bude za předpokladu, že Python 3 je v proměnné PATH, stačit jen python `test.py > results.txt`, pokud samotné `test.py > results.txt` nefunguje. V případě Unixů musíme změnit soubor na spustitelný soubor (`chmod +x test.py`) a poté jej vyvolat zapsáním `./test.py`. Pokud je adresář, v němž je soubor obsažen, uložen v proměnné PATH, pak pro jeho spuštění stačí zapsat jen `test.py`.

Chyba Windows ohledně asociace souborů ➢ 22

Čtení dat lze realizovat přesměrováním souboru s daty jako vstupu podobným způsobem jako přesměrování výstupu. Pokud bychom však použili přesměrování na soubor `sum1.py`, náš program by selhal. To je dáno tím, že funkce `input()` vyvolá výjimku, pokud obdrží znak EOF (End Of File – konec souboru). Zde je robustnější verze (sum2.py), která přijímá vstup od uživatele píšícího na klávesnici nebo skrze přesměrování souboru:

```
print("Zadávejte celá čísla, za každým Enter; nebo ^D nebo ^Z pro ukončení")

total = 0
count = 0

while True:
    try:
        line = input()
        if line:
            number = int(line)
            total += number
            count += 1
    except ValueError as err:
        print(err)
        continue
    except EOFError:
        break

if count:
    print("počet =", count, "celkem =", total, "průměr =", total / count)
```

Při použití příkazu `sum2.py < data/sum2.dat` (kde soubor `sum2.dat` umístěný v podadresáři `data` obsahuje seznam čísel) obdržíme následující výstup:

```
Zadávejte celá čísla, za každým Enter; nebo ^D nebo ^Z pro ukončení
počet = 37 celkem = 1839 průměr = 49.7027027027
```

Provedli jsme několik malých změn, díky kterým je program mnohem vhodnější pro použití interaktivním způsobem i při přesměrování. Zaprvé jsme změnili ukončení z prázdného řádku na znak EOF (Ctrl+D v Unixu, Ctrl+Z, Enter ve Windows). Díky tomu je náš program robustnější při zpracování vstupních souborů obsahujících prázdné řádky. Dále jsme přestali vypisovat výzvu pro každé číslo, protože v případě přesměrovaného vstupu to nemá žádný význam. A nakonec jsme použili jediný blok `try` se dvěma obsluhami výjimek.

Všimněte si, že pokud je zadáno neplatné číslo (ať už přes klávesnici nebo kvůli nesprávnému řádku dat v přesměrovaném vstupním souboru), vyvolá převod `int()` výjimku `ValueError` a tok programu okamžitě přejde k relevantnímu bloku `except`. To znamená, že při výskytu neplatných dat se nezvýší proměnná `count` ani `total`, což je přesně to, co chceme.

Stejně snadno bychom mohli použít dva samostatné bloky `try`:

```
while True:
    try:
        line = input()
        if line:
            try:
                number = int(line)
            except ValueError as err:
                print(err)
                continue
       total += number
       count += 1
    except EOFError:
        break
```

Ovšem mnohem lepší je seskupit výjimky na konci, aby tak hlavní zpracování zůstalo co nejméně rozházené.

Oblast č. 8: Tvorba a volání funkcí

Není vůbec žádný problém psát programy pomocí datových typů a řídících struktur, které jsme probírali v předchozích oblastech. Nicméně velmi často potřebujeme provádět v podstatě stejné zpracování opakovaně, ale s malou odlišností, jako je odlišná počáteční hodnota. Jazyk Python nabízí prostředky pro zapouzdření sad do podoby funkcí, které lze parametrizovat předávanými argumenty. Zde je obecná syntaxe pro tvorbu funkcí:

```
def názevFunkce(argumenty):
    sada
```

Argumenty jsou volitelné, přičemž více argumentů musí být odděleno čárkou. Každá funkce jazyka Python má nějakou návratovou hodnotu. Ve výchozím stavu se jedná o `None`, pokud ovšem ve funkci nepoužijeme příkaz `return` *hodnota*, kdy se vrátí zadaná hodnota. Vrácenou hodnotou může být jediná hodnota nebo n-tice hodnot. Volající ji může ignorovat. V takovém případě je prostě zahozena.

Příkaz return ➤ 172

Všimněte si, že `def` je příkaz, který funguje podobně jako operátor přiřazení. Při provedení příkazu `def` je vytvořen funkční objekt a dále se vytvoří odkaz na objekt se zadaným názvem, který se nastaví tak, aby ukazoval na nový funkční objekt. Funkce jsou tedy objekty, a proto je lze uchovávat v datových typech pro kolekce a předávat jako argumenty jiným funkcím, k čemuž se ještě dostaneme v pozdějších lekcích.

Jedna z častých potřeb při psaní interaktivní konzolové aplikace spočívá v získání čísla od uživatele. Zde je funkce, která se o to postará:

```
def get_int(msg):
    while True:
        try:
            i = int(input(msg))
            return i
        except ValueError as err:
            print(err)
```

Tato funkce přijímá jeden argument `msg`. Uvnitř cyklu `while` je uživatel vyzván k zadání čísla. Pokud zadá něco neplatného, vyvolá se výjimka `ValueError`, vypíše se chybová zpráva a cyklus se opakuje. Jakmile dojde k zadání platného čísla, vrátíme jej volajícímu. Zde je příklad volání naší funkce:

```
age = get_int("zadej svůj věk: ")
```

V tomto příkladu je jediný argument povinný, protože jsme neuvedli jeho výchozí hodnotu. Ve skutečnosti jazyk Python nabízí pro parametry funkcí velice sofistikovanou a flexibilní syntaxi, která podporuje výchozí hodnoty argumentů a argumenty definované dle pozice nebo klíčového slova. Celou tuto syntaxi si podrobně rozebereme v lekci 4.

I když vytváření vlastních funkcí může být velmi uspokojující, v řadě případů není nezbytně nutné. To je dáno tím, že jazyk Python má spoustu vestavěných funkcí a mnohonásobně více funkcí v modulech své standardní knihovny, takže to, co potřebujeme, je již s největší pravděpodobností k dispozici.

Modul Pythonu je obyčejný soubor `.py`, který obsahuje kód jazyka Python, jako jsou definice vlastních funkcí a tříd (vlastních datových typů) a někdy též proměnných. Pro přístup k funkčnosti v určitém modulu je nutné jej nejdříve importovat:

```
import sys
```

K importování modulu se používá příkaz import, za nímž následuje název soubor `.py` bez přípony.*

* Modul sys, některé další vestavěné moduly a moduly implementované v jazyku C nemusejí mít nutně odpovídající soubor .py. Používají se však naprosto stejně jako ty, které jej mají.

Jakmile je modul importován, můžeme přistupovat k libovolným funkcím, třídám nebo proměnným, které obsahuje:

```
print(sys.argv)
```

Modul `sys` poskytuje proměnnou `argv` obsahující seznam, jehož první prvek je název, pod kterým byl daný program spuštěn, a jehož druhý prvek a všechny následující představují argumenty předané programu z příkazového řádku. Dva výše uvedené řádky tvoří dohromady celý program `echoargs.py`. Pokud program spustíme na příkazovém řádku jako `echoargs.py -v`, vypíše na konzoli `['echoargs.py', '-v']` (v Unixu může být první záznam `'./echoargs.py'`).

Operátor tečka (.) ➢ 30

Syntaxe pro použití funkce z nějakého modulu má obecný tvar *`nazevModulu.nazevFunkce(argumenty)`*. Využíváme zde operátor tečka („přístup k atributu“), s nímž jsme se seznámili v Oblasti č. 3. Standardní knihovna obsahuje spoustu modulů a my budeme v této knize používat velkou část z nich. Názvy všech standardních modulů mají malá písmena, takže někteří programátoři používají pro odlišení svých vlastních modulů názvy, jejichž slova začínají velkými písmeny (např. `MyModule`).

Podívejme se nyní na jeden příklad využívající modul `random` (umístěný v souboru `random.py` standardní knihovny), který nabízí řadu užitečných funkcí:

```
import random
x = random.randint(1, 6)
y = random.choice(["jablko", "banán", "hruška", "švestka"])
```

Po provedení těchto příkazů bude proměnná x obsahovat číslo mezi 1 a 6 včetně a proměnná y bude obsahovat jeden z řetězců ze seznamu, který jsme předali funkci `random.choice()`.

Řádek shebang (#!) ➢ 22

Všechny příkazy import se standardně umisťují na začátek souborů `.py` za řádek shebang a za dokumentaci k modulu (na dokumentování modulů se podíváme v lekci 5). Doporučujeme nejdříve importovat moduly standardní knihovny, poté moduly třetích stran a nakonec své vlastní moduly.

Příklady

V předchozí části jsme se o jazyku Python naučili dost na to, abychom mohli vytvářet skutečné programy. V této části prostudujeme dva hotové programy, které využívají pouze ty části jazyka Python, které jsme již probrali. Na těchto programech si ukážeme, co je možné vytvořit, a také si na nich upevníme dosud získané znalosti.

V následujících lekcích se budeme postupně zabývat dalšími oblastmi jazyka Python a jeho knihovny, abychom pak byli schopni psát programy, které budou stručnější a robustnější ve srovnání s těmi, které si ukážeme v této části. Nejdříve ale musíme položit základy, na nichž pak budeme stavět.

Program bigdigits.py

První program, na který se podíváme, je docela krátký, i když má několik zajímavých aspektů, mezi něž patří seznam seznamů. Program funguje tak, že na příkazovém řádku zadáme číslo a program jej vypíše do konzoly pomocí „velkých“ číslic.

Na serverech se spoustou uživatelů, kteří sdílejí vysokorychlostní řádkovou tiskárnu, to běžně probíhalo tak, že před tiskovou úlohou každého uživatele se pomocí této techniky vytiskl úvodní list, který obsahoval jeho uživatelské jméno a nějaké další identifikační údaje.

Kód programu prostudujeme ve třech částech: příkaz `import`, tvorba seznamů uchovávajících data a samotné zpracování. Podívejme se nejdříve na jeho ukázkové spuštění:

```
bigdigits.py 41072819
   *     *     ***    *****   ***    ***    *    ****
  **    **    *   *       *  *   *  *   *  **   *   *
 * *     *    *     *     *   *  *  *   *   *   *   *
*  *     *    *     *    *      *     ***    *    ****
******   *    *     *   *      *     *   *   *       *
   *     *    *   *   *      *      *   *   *       *
   *    ***    ***    *      *****   ***   ***      *
```

Výzvu konzoly (nebo úvodní znaky `./` pro uživatele Unixu) jsme si zde neukázali. Od této chvíle je budeme považovat za samozřejmou součást.

```
import sys
```

Vzhledem k tomu, že argument (číslo, které se má vypsat) musíme načíst z příkazového řádku, potřebujeme přístup k seznamu `sys.argv`. Z tohoto důvodu začínáme importem modulu `sys`.

Každé číslo reprezentujeme jako seznam řetězců. Například nula vypadá takto:

```
Zero = ["  ***  ",
        " *   * ",
        "*     *",
        "*     *",
        "*     *",
        " *   * ",
        "  ***  "]
```

Všimněte si, že seznam řetězců `Zero` je rozložen na několik řádků. Příkazy jazyka Python obvykle zabírají jediný řádek, lze je však rozložit na více řádků, jedná-li se o uzávorkovaný výraz, literál seznamu, množiny či slovníku, seznam argumentů při volání funkce nebo víceřádkový příkaz, v němž je každý znak konce řádku vyjma posledního potlačen předsazením zpětného lomítka (`\`). Ve všech těchto případech lze použít libovolný počet řádků, přičemž u druhého a následujících řádků již nezáleží na odsazení.

Typ `set` ➤ 123

Typ `dict` ➤ 128

Každý seznam reprezentující určité číslo má sedm řetězců jednotné délky, která však může být u každého čísla jiná. Seznamy pro ostatní čísla fungují stejným způsobem jako pro nulu, ovšem kvůli kompaktnosti jsou uvedena na jediném řádku:

```
One = [" * ", "** ", " * ", " * ", " * ", " * ", "***"]
Two = [" *** ", "* *", "* * ", " * ", " * ", "* ", "*****"]
# ...
Nine = [" ****", "* *", "* *", " ****", " *", " *", " *"]
```

Posledním kouskem dat, která potřebujeme, je seznam seznamů všech čísel:

```
Digits = [Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine]
```

Seznam `Digits` bychom mohli vytvořit také přímo a vyhnout se tak tvorbě dalších proměnných:

```
Digits = [
    [" *** ", " * * ", "* *", "* *", "* *", " * * ", " *** "], # Nula
    [" * ", "** ", " * ", " * ", " * ", " * ", "***"], # Jedna
    # ...
    [" ****", "* *", "* *", " ****", " *", " *", " *"] # Devět
]
```

Raději jsme však pro každé číslo použili samostatnou proměnnou – jednak kvůli snazšímu pochopení, a také kvůli tomu, že kód pak působí čistším dojmem.

Nyní uvedeme celou zbývající část kódu, takže si ještě před tím, než začnete číst vysvětlení, můžete vyzkoušet, zda přijdete na to, jak vlastně funguje:

```
try:
    digits = sys.argv[1]
    row = 0
    while row < 7:
        line = ""
        column = 0
        while column < len(digits):
            number = int(digits[column])
            digit = Digits[number]
            line += digit[row] + " "
            column += 1
        print(line)
        row += 1
except IndexError:
    print("použití: bigdigits.py <číslo>")
except ValueError as err:
    print(err, "v", digits)
```

Celý kód je zabalen do obsluhy výjimky, která zachytí dvě chybové situace. Začínáme načtením argumentu z příkazového řádku programu. Seznam `sys.argv` začíná jako všechny seznamy jazyka Python od nuly. Prvek na pozici 0 je název, pod kterým byl program spuštěn, takže v běžícím programu má tento seznam vždy alespoň jeden prvek. Pokud nebyl zadán žádný argument, pak budeme zkoušet přistupovat ke druhému prvku jednoprvkového seznamu, což způsobí vyvolání výjimky `IndexError`. V takovém případě se řízení předá odpovídajícímu bloku obsluhy výjimky, kde jednoduše vypíšeme, jak se program používá. Provádění pak pokračuje za koncem bloku `try`, kde však již není žádný kód, a proto program skončí.

range() ➤ 141

Pokud k výjimce `IndexError` nedojde, uchovává řetězec `digits` argument příkazového řádku, o němž se domníváme, že obsahuje posloupnost číselných znaků. (Vzpomeňte si z Oblasti č. 2, že u identifikátorů se rozlišuje velikost písmen, takže `digits` a `Digits` jsou dvě různé proměnné.) Každá velká číslice je reprezentována sedmi řetězci, takže pro správný výstup musíme nejdříve vypsat horní řádek každé číslice, pak další řádek a tak pořád dál, dokud se nevypíše všech sedm řádků. K průchodu přes všechny řádky používáme cyklus `while`. Stejně by fungoval i cyklus `for` (`for row in (0, 1, 2, 3, 4, 5, 6):`). Později se seznámíme s mnohem lepším způsobem využívajícím vestavěnou funkci `range()`.

Řetězec `line` používáme pro uložení řetězců daného řádku ze všech zpracovávaných číslic. Poté procházíme jednotlivé sloupce, což znamená jednotlivé znaky v argumentu příkazového řádku. Každý znak získáváme pomocí výrazu `digits[column]` a převádíme jej na celé číslo s názvem `number`. Pokud převod selže, vyvolá se výjimka `ValueError` a řízení se okamžitě předá odpovídající obsluze výjimky. V tomto případě vypíše chybovou zprávu a program pokračuje za blokem `try`. Zde ale, jak jsme si řekli již dříve, není již žádný kód, a proto náš program jednoduše skončí.

Pokud převod uspěje, použijeme proměnnou `number` jako index do seznamu `Digits`, z něhož extrahujeme seznam řetězců příslušné číslice. Poté z tohoto seznamu přidáme řetězec na pozici `row` do proměnné `line`, k níž přidáme také mezeru pro vodorovné odsazení každé číslice.

Při každém ukončení vnitřního cyklu `while` vypíšeme obsah sestavený v proměnné `line`. Klíčem k pochopení tohoto programu je místo, kde připojujeme řetězec s řádkem každé číslice k proměnné `line` reprezentující aktuální řádek. Vyzkoušejte si spustit program, abyste se přesvědčili, jak pracuje. Vrátíme se k němu ve cvičení, kde malinko vylepšíme jeho výstup.

Program generate_grid.py

Jedna z častých potřeb je generování testovacích dat. Neexistuje žádný generický program, který by toto prováděl, poněvadž testovací data se značně liší. Python se často používá k vytváření testovacích dat, protože jeho programy se velice snadno píšou a upravují. V této podčásti vytvoříme program, který generuje tabulku náhodných čísel. Uživatel může stanovit, kolik má mít řádků a sloupců a v jakém rozmezí se mají generovaná čísla nacházet. Začneme pohledem na ukázkové spuštění:

```
generate_grid.py
řádků: 4x
invalid literal for int() with base 10: '4x'
řádků: 4
sloupců: 7
minimum (nebo Enter pro 0): -100
maximum (nebo Enter pro 1000):
554 720 550 217 810 649 912
-24 908 742 -65 -74 724 825
711 968 824 505 741 55 723
180 -60 794 173 487 4 -35
```

Program pracuje interaktivně a na začátku jsme udělali chybu při zadávání počtu řádků. Program reagoval vypsáním chybové zprávy a poté nás znovu požádal o zadání počtu řádků. Pro maximum jsme jen stiskli klávesu Enter, čímž jsme přijali výchozí hodnotu.

Kód si prohlédneme ve čtyřech částech: příkaz `import`, definice funkce `get_int()` (sofistikovanější varianta původní verze definované v Oblasti č. 8), interakce s uživatelem pro získání hodnot a samotné zpracování.

```
import random
```

`random.randint()` ➤ 46

Pro přístup k funkci `random.randint()` potřebujeme modul `random`.

```
def get_int(msg, minimum, default):
    while True:
        try:
            line = input(msg)
            if not line and default is not None:
                return default
            i = int(line)
            if i < minimum:
                print("musí být >=", minimum)
            else:
                return i
        except ValueError as err:
            print(err)
```

Tato funkce vyžaduje tři argumenty: řetězec se zprávou, minimální hodnotu a výchozí hodnotu. Pokud uživatel stiskne jen klávesu Enter, pak nastávají dvě možnosti. Má-li parametr default hodnotu `None`, což znamená, že výchozí hodnota nebyla zadána, předá se řízení programu až na řádek s převodem `int()`, který selže (protože `''` nelze převést na číslo) a vyvolá výjimku `ValueError`. Pokud ale parametr default není `None`, vrátíme jeho hodnotu. V opačném případě se funkce pokusí převést text zadaný uživatelem na celé číslo a v případě úspěchu zkontroluje, zda je toto číslo alespoň zadané minimum.

Funkce tedy vždy vrátí buď výchozí hodnotu (pokud uživatel jen stiskl Enter), nebo platné celé číslo, které je větší nebo rovno než stanovené minimum.

```
rows = get_int("řádků: ", 1, None)
columns = get_int("sloupců: ", 1, None)
minimum = get_int("minimum (nebo Enter pro 0): ", -1000000, 0)

default = 1000
if default < minimum:
    default = 2 * minimum
maximum = get_int("maximum (nebo Enter pro " + str(default) + "): ",
                  minimum, default)
```

Naše funkce `get_int()` usnadňuje získávání počtu řádků a sloupců a minimální požadované hodnoty náhodného čísla. Pro řádky a sloupce nastavujeme výchozí hodnotu na `None`, což znamená žádná výchozí hodnota, takže uživatel musí zadat nějaké číslo. V případě minima používáme výchozí hodnotu 0 a pro maximum máme výchozí hodnotu 1000 nebo dvojnásobek minima v případě, že je minimum větší nebo rovno 1000.

Jak jsme si řekli již u předchozího příkladu, seznam argumentů při volání funkce se může rozkládat na více řádcích, přičemž na druhém a na všech následujících řádcích je odsazení bezvýznamné.

Jakmile víme, kolik řádků a sloupců uživatel požaduje a jaké jsou hodnoty pro minima a maxima pro náhodná čísla, můžeme přejít k vlastnímu zpracování.

```
row = 0
while row < rows:
    line = ""
    column = 0
    while column < columns:
        i = random.randint(minimum, maximum)
        s = str(i)
        while len(s) < 10:
            s = " " + s
        line += s
        column += 1
    print(line)
    row += 1
```

K vygenerování tabulky používáme tři cykly `while`. Vnější pracuje s řádky, prostřední se sloupci a vnitřní s jednotlivými znaky. V prostředním cyklu získáváme náhodné číslo ve stanoveném rozsahu, které poté převedeme na řetězec. Vnitřní cyklus `while` používáme k vyplnění řetězce úvodními mezerami, aby tak každé číslo reprezentoval řetězec dlouhý 10 znaků. Pro nashromáždění čísel na každém řádku používáme řetězec `line`, který vypisujeme vždy po přidání čísel každého sloupce. Tím jsme se dostali na konec našeho druhého příkladu.

Python nabízí velmi sofistikované funkce pro formátování řetězců a také skvělou podporu pro cykly `for ... in`, takže v realističtější verzi obou programů `bigdigits.py` a `generate_grid.py` bychom použili cykly `for ... in` a v programu `generate_ grid.py` bychom místo hrubě vyplňovaných mezer použili funkce Pythonu pro formátování řetězce. Omezili jsme se však na osm oblastí jazyka Python, s kterými jsme se seznámili v této lekci a které jsou dostatečné pro psaní ucelených a užitečných programů. V každé z následujících lekci si osvojíme nové prvky jazyka Python, takže při postupu touto knihou budou naše programy a dovednosti stále sofistikovanější.

`str.format()` ➤ 83

Shrnutí

V této lekci jsme se naučili, jak upravovat a spouštět programy napsané v jazyku Python, a prostudovali jsme několik malých, ale ucelených programů. Většina stránek této lekce byla věnována osmi oblastem „nádherného srdce“ jazyka Python, jejichž osvojení stačí pro psaní skutečných programů.

Začali jsme dvěma nejzákladnějšími datovými typy jazyka Python: `int` (celé číslo) a `str` (řetězec). Celočíselné literály se zapisují stejně jako ve většině ostatních programovacích jazyků. Řetězcové literály se zapisují pomocí jednoduchých nebo dvojitých uvozovek. Nezáleží na tom, které použijeme, podstatné je, aby byl na obou koncích stejný druh uvozovek. Řetězce a celá čísla můžeme navzájem převádět (např. `int("250")` a `str(125)`). Pokud převod na celé číslo selže, vyvolá se výjimka `ValueError`. Na druhou stranu na řetězec lze převést téměř cokoliv.

Řetězce jsou posloupnosti, takže funkce a operace, které lze použít u posloupností, lze použít také pro řetězce. Můžeme tak například pomocí operátoru pro přístup k prvku (`[]`) přistoupit k určitému znaku, pomocí operátoru + spojit řetězce a pomocí operátoru += připojit jeden řetězec k druhému. Řetězce jsou neměnitelné, a proto se při připojování vytvoří v pozadí nový řetězec, který vznikne spojením daných řetězců, a s výsledným řetězcem se opětovně sváže odkaz na objekt řetězce na levé straně operátoru. Pomocí cyklu `for ... in` můžeme procházet jednotlivé znaky řetězce a pomocí vestavěné funkce `len()` můžeme zjistit počet znaků v řetězci.

U neměnitelných objektů, jako jsou řetězce, celá čísla a n-tice, můžeme psát svůj kód tak, jako by odkaz na objekt byl proměnná, což znamená, jako by odkaz na objekt byl samotný objekt, na který ukazuje. Totéž můžeme dělat i v případě měnitelných objektů, i když jakákoliv změna provedená na měnitelném objektu ovlivní všechny výskyty daného objektu (tj. všechny odkazy na daný objekt). Tomuto tématu se budeme podrobněji věnovat v lekci 3.

Jazyk Python nabízí několik vestavěných datových typů pro kolekce a několik dalších je k dispozici v jeho standardní knihovně. Seznámili jsme se s typy `list` (seznam) a `tuple` (n-tice) a s tím, jak se vytvářejí n-tice a seznamy z literálů (např. `even = [2, 4, 6, 8]`). Seznamy, podobně jako vše ostatní v jazyku Python, jsou objekty, takže na nich můžeme volat metody. Například `even.append(10)` přidá další prvek do seznamu. Seznamy a n-tice jsou stejně jako řetězce posloupnosti, a proto je můžeme pomocí cyklu `for ... in` procházet po jednotlivých prvcích a pomocí funkce `len()` zjistit, kolik prvků obsahují. Pomocí operátoru pro přístup k prvku (`[]`) můžeme též získat určitý prvek ze seznamu nebo n-tice, pomocí operátoru + dva seznamy nebo n-tice spojíme a pomocí operátoru += připojíme jeden k druhému. Pokud bychom chtěli připojit jediný prvek k seznamu, pak musíme použít buď metodu `list.append()`, nebo operátor += s prvkem přetvořeným do jednoprvkového seznamu (např. `even += [12]`). Seznamy jsou měnitelné, a proto můžeme použít operátor [] ke změně jednotlivých prvků (např. `even[1] = 16`).

Rychlé operátory `is` a `is not` lze použít pro kontrolu, zda dva odkazy na objekty ukazují na tentýž objekt, což je zvláště užitečné při kontrole proti unikátnímu vestavěnému objektu `None`. K dispozici jsou všechny obvyklé porovnávací operátory (<, <=, ==, !=, >=, >), lze je však použít pouze s kompatibilními datovými typy, které navíc musejí danou operaci podporovat. Všechny datové typy, s nimiž jsme se dosud seznámili (`int`, `str`, `list` a `tuple`), podporují celou skupinu porovnávacích operátorů. Při porovnávání nekompatibilních typů (např. typ `int` s typem `str` nebo `list`) dojde logicky k výjimce `TypeError`.

Jazyk Python podporuje standardní logické operátory `and`, `or` a `not`. Operátory `and` a `or` používají logiku zkráceného vyhodnocování, takže vracejí operand, který rozhoduje o výsledku, což nemusí nutně být logická hodnota (i když jej lze převést na logickou hodnotu). To znamená, že ne vždy obdržíme buď `True`, nebo `False`.

Příslušnost k typům pro posloupnosti včetně řetězců, seznamů a n-tic můžeme testovat pomocí operátorů `in` a `not in`. Při testování příslušnosti se u seznamů a n-tic používá pomalé lineární vyhledávání a u řetězců potenciálně mnohem rychlejší hybridní algoritmus, avšak výkon je jen málokdy problém, tedy až na velmi dlouhé řetězce, seznamy a n-tice. V lekci 3 se seznámíme s datovými typy jazyka Python pro asociativní pole a množinové kolekce, které realizují velmi rychlé testování příslušnosti. Pomocí funkce `type()` je možné zjistit typ objektu proměnné (tj. typ objektu, na který ukazuje daný odkaz na objekt). Tato funkce se však většinou používá pouze pro ladění a testování.

Jazyk Python nabízí několik řídících struktur, mezi něž patří podmíněné větvení `if ... elif ... else`, podmíněný cyklus `while`, cyklus přes posloupnosti `for ... in` a bloky pro ošetřování výjimek `try ... except`. Cykly `while` i `for ... in` lze předčasně ukončit pomoc příkazu `break` a pomocí příkazu `continue` lze též předat řízení zpět na začátek cyklu.

Podporovány jsou obvyklé aritmetické operace, včetně `+`, `-`, `*` a `/`, ačkoliv Python je výjimečný v tom, že operátor `/` vždy vrací desetinné číslo, a to i tehdy, jsou-li operandy celočíselné. (Ořezávané dělení používané v řadě ostatních jazyků je v Pythonu k dispozici jako operátor `//`.) Python nabízí také operátory rozšířeného přiřazení, jako je `+=` a `*=`. Tyto operátory vytvářejí v pozadí nové objekty a provádějí opětovné svázání, je-li jejich levý operand neměnitelný. Aritmetické operace jsou přetížené také pro typy `str` a `list`.

Vstup a výstup na konzolu lze realizovat pomocí funkcí `input()` a `print()`, přičemž prostřednictvím přesměrování souboru v konzole můžeme tytéž vestavěné funkce použít i pro čtení a zapisování souborů.

Kromě bohaté palety vestavěných funkcí nabízí Python také svoji rozsáhlou standardní knihovnu s moduly přístupnými po jejich importu příkazem `import`. Jedním z často importovaných modulů je modul `sys`, který uchovává seznam `sys.argv` s argumenty příkazového řádku. Pokud Python nějakou funkci, kterou potřebujeme, nemá, můžeme si ji pomocí příkazu `def` snadno vytvořit.

S využitím funkčních prvků popsaných v této lekci je možné psát v jazyku Python krátké, ale užitečné programy. V následující lekci se dozvíme více o datových typech jazyka Python, podrobněji se podíváme na typy `int` a `str` a představíme si několik zcela nových datových typů. V lekci 3 se naučíme více o n-ticích a seznamech a také o některých z dalších datových typů jazyka Python určených pro kolekce. Pak se v lekci 4 budeme detailněji věnovat řídícím strukturám jazyka Python a naučíme se, jak vytvářet své vlastní funkce tak, abychom funkčnost správně zabalili, vyhnuli se duplicitnímu kódu a podporovali jeho znovupoužití.

Cvičení

Smyslem cvičení nejen v této lekci, ale v celé knize je přimět vás experimentovat s Pythonem, abyste si v praxi osvojili látku probíranou v dané lekci. V příkladech a cvičeních se budeme věnovat zpracování číselných i textových údajů, přičemž jednotlivé příklady budou co nejmenší, abyste se mohli soustředit na uvažování a učení spíše než na psaní kódu. Ke každému cvičení existuje řešení, které najdete v příkladech ke knize.

Příklady ke knize ➢ 14

1. Jedna pěkná varianta programu `bigdigits.py` může vypadat tak, že místo vypsání hvězdiček (`*`) vypíše odpovídající číslici:

```
bigdigits_ans.py 719428306
77777   1    9999     4     222    888    333     000     666
    7  11   9   9    44    2   2  8   8  3   3   0   0   6
   7    1   9   9   4 4    2  2   8   8      3  0     0  6
  7     1    9999  4  4      2     888     33   0     0  6666
 7      1       9  444444   2     8   8      3  0     0  6   6
7       1       9     4    2      8   8  3   3   0   0   6   6
7      111      9     4    22222   888    333     000     666
```

Lze použít dva přístupy. Nejjednodušší je prostě změnit hvězdičky v seznamech. To však není příliš flexibilní, a proto byste takto neměli postupovat. Místo toho změňte kód provádějící zpracování tak, aby se místo jednorázového přidání řetězce s řádkem každé číslice přidávaly jednotlivé znaky postupně a při každém výskytu hvězdičky se použila příslušná číslice.

To lze provést zkopírováním souboru `bigdigits.py` a změnou asi pěti řádků. Není to těžké, ale malinko zákeřné. Hotové řešení najdete v souboru `bigdigits_ans.py`.

2. Editor IDLE lze použít jako velice výkonnou a flexibilní kalkulačku, někdy je ale užitečné mít kalkulačku specifickou pro určitý úkol. Vytvořte program, který vyzve uživatele k zadání čísla v cyklu `while`, v němž bude postupně sestavovat seznam zadaných čísel. Jakmile uživatel dokončí zadávání (prostým stiskem klávesy Enter), vypíšou se zadaná čísla, jejich počet, součet, nejmenší a největší čísla a jejich průměr (součet / počet). Zde je ukázkový běh programu:

```
average1_ans.py
zadejte číslo nebo Enter pro ukončení: 5
zadejte číslo nebo Enter pro ukončení: 4
zadejte číslo nebo Enter pro ukončení: 1
zadejte číslo nebo Enter pro ukončení: 8
zadejte číslo nebo Enter pro ukončení: 5
zadejte číslo nebo Enter pro ukončení: 2
zadejte číslo nebo Enter pro ukončení:
čísla: [5, 4, 1, 8, 5, 2]
počet = 6 součet = 25 nejmenší = 1 největší = 8 průměr = 4.16666666667
```

Inicializace proměnných (prázdného seznamu je prostě `[]`) zabere přibližně čtyři řádky a cyklus `while` včetně základního ošetření chyb méně než 15 řádků. Výpis na konci lze provést pomocí několika málo řádků, takže celý program včetně prázdných řádků kvůli čitelnosti by měl zabírat kolem 25 řádků.

3. V některých situacích potřebujeme vygenerovat testovací text – například pro naplnění návrhu webové stránky před tím, než bude dostupný skutečný obsah, nebo pro testování obsahu při vývoji generátoru sestav. Za tímto účelem napište program, který generuje děsivé básničky (takový ten typ, při kterém by se zastyděl i Vogon).

random.randint()
a random.choice()
➤ 46

Vytvořte nějaké seznamy slov – například zájmena („můj“, „tvůj“, „její“, „jeho“), podstatná jména („kočka“, „pes“, „muž“, „žena“), slovesa („zpívá“, „běží“, „skáče“) a přísloví („hlasitě“, „tiše“, „kvalitně“, „hrozně“). Poté proveďte pět cyklů a v každé iteraci použijte funkci `random.choice()` pro

výběr zájmena, podstatného jména, příslovce a slovesa nebo jen zájmena, podstatného jména a slovesa a výslednou větu vypište. Zde je ukázkový běh programu:

```
awfulpoetry1_ans.py
její muž kvalitně říka
můj pes cítí
jeho holka tiše hopká
tvůj holka cítí
její pes plave
```

V tomto programu musíte importovat modul `random`. Seznam lze napsat na 4–10 řádků v závislosti na tom, kolik slov do něj chcete vložit, přičemž samotný cyklus vyžaduje méně než deset řádků, takže s nějakými prázdnými řádky by měl mít celý program přibližně 20 řádků kódu. Řešení najdete v souboru `awfulpoetry1_ans.py`.

4. Aby byl program s hroznou poezií všestrannější, přidejte do něj takový kód, aby v případě, že uživatel zadá na příkazovém řádku nějaké číslo (mezi 1 a 10 včetně), program vypsal zadaný počet řádků. Pokud žádný argument na příkazovém řádku není zadán, vypíše se 5 řádků jako dříve. Musíte změnit hlavní cyklus (např. na cyklus `while`). Mějte na paměti, že porovnávací operátory jazyka Python lze řetězit, není tedy nutné při kontrole rozsahu argumentu používat logický operátor `and`. Dodatečnou funkčnost lze zajistit přidáním přibližně deseti řádků kódu. Řešení najdete v souboru `awfulpoetry2_ans.py`.

5. Bylo by pěkné mít možnost vypočítat medián (střední hodnotu) a také střed pro průměry ze cvičení 2, k tomu je ale nutné seznam seřadit. V Pythonu lze seznam snadno seřadit pomocí metody `list.sort()`, o které jsme se zatím nezmínili, takže ji zde nepoužijeme. Rozšiřte program `average1_ans.py` o blok kódu, který seřadí seznam čísel. Efektivita nás nezajímá, použijte ten nejjednodušší postup, který vás napadne. Jakmile je seznam seřazen, je mediánem prostřední hodnota, má-li seznam lichý počet prvků, nebo průměr dvou prostředních hodnot, má-li seznam sudý počet prvků. Vypočtěte medián a společně s ostatními informacemi jej vypište.

Tohle je malinko složitější, zvláště pro nezkušené programátory. Může to být těžší i pro ty, kteří s Pythonem již nějaké ty zkušenosti mají, alespoň tedy v případě, že se budete držet stanoveného omezení a použijete pouze ty části Pythonu, které jsme již probírali. Řazení lze provést asi na desítce řádků a výpočet mediánu (v němž nemůžeme použít operátor modulo, protože jsme jej ještě neprobírali) nezabere víc než čtyři řádky. Řešení najdete v souboru `average2_ans.py`.

LEKCE 2

Datové typy

V této lekci:

- Identifikátory a klíčová slova
- Celočíselné typy
- Typy s pohyblivou řádovou čárkou
- Řetězce

V této lekci se na jazyk Python podíváme mnohem detailněji. Začneme diskuzí o pravidlech pojmenovávání odkazů na objekty a ukážeme si seznam klíčových slov jazyka Python. Poté se podíváme na všechny důležité datové typy jazyka Python vyjma datových typů představujících kolekce, kterým se budeme věnovat v lekci 3. Všechny zde probírané datové typy jsou vestavěné kromě jednoho, který pochází ze standardní knihovny. Jediný rozdíl mezi vestavěnými a knihovními datovými typy spočívá v tom, že v druhém případě musíme nejdříve importovat příslušný modul a název datového typu kvalifikovat názvem modulu, z něhož pochází (více viz Lekce 5).

Identifikátory a klíčová slova

Odkazy na objekty ➢ 26

Při vytváření datového prvku jej můžeme buď přiřadit proměnné, nebo vložit do nějaké kolekce. (Jak jsme si řekli již v předchozí lekci, při přiřazování se v Pythonu ve skutečnosti děje to, že se sváže odkaz na objekt tak, aby ukazoval na objekt v paměti uchovávající daná data.) Název, který dáme našemu odkazu na objekt, se nazývá *identifikátor* nebo prostě *jméno*.

Platným identifikátorem v jazyku Python je neprázdná posloupnost znaků libovolné délky, která se skládá z „počátečního znaku" a nulového nebo většího počtu „pokračovacích znaků". Takovýto identifikátor musí splňovat několik pravidel a dodržovat následující konvence. První pravidlo se týká počátečního a pokračovacích znaků. Počátečním znakem může být cokoliv, co znaková sada Unicode považuje za písmeno, tedy všechny písmena ASCII („a", „b", ..., „z", „A", „B", ..., „Z"), podtržítko („_") a písmena z většiny neanglických jazyků. Každý pokračovací znak může být libovolným znakem, který je povolen pro počáteční znak, nebo téměř jakýkoliv znak různý od bílého místa, což mimo jiné znamená všechny znaky, které znaková sada Unicode považuje za číslice, jako je („0", „1", ..., „9"), nebo český znak „ř". U identifikátorů se rozlišuje velikost písmen, takže například `TAXRATE`, `Taxrate`, `TaxRate`, `taxRate` a `taxrate` je pět odlišných identifikátorů.

Přesná sada znaků, které jsou povolené pro počáteční a pokračovací znaky, je popsána v dokumentaci (viz *http://docs.python.org/reference/lexical_analysis.html#identifiers*) a v návrhu PEP 3131 (viz *http://www.python.org/dev/peps/pep-3131/*).*

Druhé pravidlo říká, že žádný identifikátor nemůže mít stejný název jako některé klíčové slovo jazyka Python, takže nemůžeme použít žádné ze jmen uvedených v tabulce 2.1.

Tabulka 2.1: Klíčová slova jazyka Python

and	continue	except	global	lambda	pass	while
as	def	False	if	None	raise	with
assert	del	finally	import	nonlocal	return	yield
break	elif	for	in	not	True	
class	else	from	is	or	try	

S většinou z těchto klíčových slov jsme se již setkali v předchozí lekci, i když 11 z nich (`assert`, `class`, `del`, `finally`, `from`, `global`, `lambda`, `nonlocal`, `raise`, `with` a `yield`) budeme ještě probírat.

* Zkratka PEP znamená Python Enhancement Proposal (návrh na rozšíření Pythonu). Pokud chce někdo Python změnit nebo rozšířit, pak může za předpokladu, že má dostatečnou podporu komunity Pythonu, předložit návrh PEP s podrobnostmi o navrhovaných změnách, aby jej bylo možné formálně posoudit a v některých případech, jako je třeba právě PEP 3131, přijmout a implementovat. Všechny návrhy PEP jsou přístupné ze stránky *www.python.org/dev/peps/*.

První konvence zní takto: Pro své vlastní identifikátory nepoužívejte název žádného z předdefinovaných identifikátorů Pythonu. To tedy znamená, že byste neměli používat názvy `NotImplemented` a `Ellipsis`, žádný z názvů vestavěných datových typů (jako je `int`, `float`, `list`, `str` a `tuple`), funkcí nebo výjimek jazyka Python. Jak ale zjistíme, zda daný identifikátor spadá do některé z těchto kategorií? Python nabízí vestavěnou funkci `dir()`, která vrací seznam atributů zadaného objektu. Pokud ji zavoláme bez argumentů, vrátí seznam vestavěných atributů Pythonu:

```
>>> dir() # senam Pythonu 3.1 má navíc prvek '__package__'
['__builtins__', '__doc__', '__name__']
```

Atribut `__builtins__` je ve skutečnosti modul, který uchovává všechny vestavěné atributy Pythonu. Můžeme jej tedy použít jako argument funkce `dir()`:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
...
'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

V tomto seznamu je přibližně 130 jmen, takže většinu z nich jsme vypustili. Jména začínající velkým písmenem jsou názvy vestavěných výjimek Pythonu. Ostatní jsou názvy funkcí a datových typů.

Druhá konvence se týká použití podtržítek (_). Neměly by se používat názvy, které začínají a končí podtržítky (např. `__lt__`). Python definuje rozličné speciální metody a proměnné s těmito názvy (takovéto speciální metody můžeme opětovně implementovat, to znamená vytvořit si jejich vlastní verze), sami bychom však nové názvy tohoto typu zavádět neměli. Více se těmto typům názvů budeme věnovat v lekci 6. S názvy, které začínají jedním nebo dvěma podtržítky (a které nekončí dvěma podtržítky), se v určitých kontextech zachází zvláštním způsobem. Příklad použití názvů s jedním úvodním podtržítkem si ukážeme v lekci 5 a na praktické využití těch, které obsahují dvě úvodní podtržítka, se podíváme v lekci 6.

Jediné osamocené podtržítko lze použít jako identifikátor, přičemž v interaktivním interpretu nebo v okně Python Shell uchovává _ výsledek posledního výrazu, který byl vyhodnocen. V normálním běžícím programu žádná proměnná _ neexistuje, pokud ji sami explicitně nepoužijeme. Někteří programátoři rádi používají _ v cyklech `for ... in`, když se nezajímají o procházené prvky:

```
for _ in (0, 1, 2, 3, 4, 5):
    print("Ahoj")
```

Mějte však na paměti, že při psaní internacionalizovaných programů se _ používá často jako název překladové funkce. Důvodem je to, že místo `gettext.gettext("Přelož mne")` stačí napsat jen `_("Přelož mne")`. (Abychom mohli přistupovat k funkci `gettext()`, musíme nejdříve importovat modul `gettext`.)

import ➤ 45
import ➤ 194

Pojďme se nyní podívat na několik platných identifikátorů, jejichž názvy jsou v češtině. V kódu předpokládáme, že jsme již provedli příkaz `import math` a že proměnné `poloměr` a `stará_oblast` již byly vytvořeny v předchozí části programu:

```
π = math.pi
ε = 0.0000001
nová_oblast = π * poloměr * poloměr
if abs(nová_oblast - stará_oblast) < ε:
   print("oblast konverguje")
```

Použili jsme modul `math`, proměnnou epsilon (ε) jsme nastavili na velmi malé desetinné číslo a pomocí funkce `abs()` jsme získali absolutní hodnotu rozdílu mezi oblastmi – ke všem těmto prvků se ještě vrátíme v pozdější části této lekce. Pro nás je ale podstatné to, že pro identifikátory můžeme klidně používat znaky s diakritikou a znaky řecké abecedy. Stejně jednoduše bychom mohli vytvořit identifikátory s použitím arabských, čínských, hebrejských, japonských a ruských znaků anebo znaků z kteréhokoli jiného jazyka podporovaného znakovou sadou Unicode.

Nejjednodušším způsobem, jak otestovat, zda je daný identifikátor platný, je vyzkoušet přiřazení v interaktivním interpretu jazyka Python nebo v okně Python Shell v editoru IDLE. Zde je několik příkladů:

```
>>> stretch-factor = 1
SyntaxError: can't assign to operator (...)
>>> 2miles = 2
SyntaxError: invalid syntax (...)
>>> str = 3 # Platné, ale NEVHODNÉ
>>> l'impôt31 = 4
SyntaxError: EOL while scanning single-quoted string (...)
>>> l_impôt31 = 5
>>>
```

Syntaktické chyby ➤ 401

Použití neplatného identifikátoru způsobí vyvolání výjimky `SyntaxError`. V jednotlivých případech se část chybové zprávy umístěné v závorkách liší, a proto jsme ji nahradili výpustkem. První přiřazení selže, protože „-“ není písmenem, číslicí ani podtržítkem znakové sady Unicode. Druhé selže kvůli tomu, že počáteční znak není písmenem nebo podtržítkem znakové sady Unicode (pouze pokračovací znaky mohou být číslicemi). Pokud vytvoříme identifikátor, který je platný, tak se žádná výjimka nevyvolá, a to ani tehdy, je-li jeho název stejný jako některý z vestavěných datových typů, výjimek nebo funkcí. Třetí přiřazení tedy funguje, i když je krajně nevhodné. Čtvrté přiřazení selže, protože uvozovka není písmenem, číslicí ani podtržítkem znakové sady Unicode. Páté přiřazení je v pořádku.

Celočíselné typy

Python nabízí dva vestavěné celočíselné typy `int` a `bool`.* Celá čísla i logické hodnoty jsou neměnitelné, ale díky operátorům rozšířeného přiřazení jazyka Python si toho jen zřídkakdy všimnete. Při použití v logických výrazech má `0` a `False` hodnotu `False`, přičemž jakékoliv jiné celé číslo a `True` mají hodnotu `True`. Při použití v číselných výrazech se `True` vyhodnotí na `1` a `False` na `0`. To zname-

* Standardní knihovna nabízí také typ `fractions.Fraction` (racionální čísla s neomezenou přesností), který může být v některých specializovaných matematických a vědeckých kontextech užitečný.

ná, že můžeme psát poněkud zvláštní věci. Můžeme tak například inkrementovat celé číslo `i` pomocí výrazu `i += True`. Správně bychom měli samozřejmě použít `i += 1`.

Celá čísla

Velikost celého čísla je omezena pouze pamětí počítače, takže můžeme snadno vytvářet a zpracovávat celá čísla o velikosti stovek cifer, i když to bude pomalé ve srovnání s celými čísly, která lze reprezentovat nativně procesorem počítače.

Tabulka 2.2: Číselné operace a funkce

Syntaxe	Popis
x + y	Sečte číslo x a číslo y.
x - y	Odečte číslo y od čísla x.
x * y	Vynásobí číslo x číslem y.
x / y	Podělí číslo x číslem y, přičemž vždy vrátí typ `float` (nebo `complex`, je-li x nebo y typu `complex`).
x // y	Podělí číslo x číslem y, přičemž ořeže případnou desetinnou část, takže vždy vrátí typ `int` (viz také funkce `round()`).
x % y	Vrátí modul (zbytek) po dělení čísla x číslem y.
x ** y	Umocní číslo x mocninou y (viz také funkce `pow()`).
-x	Obrátí číslo x, takže změní jeho znaménko, pokud se nejedná o nulu (nula zůstane nezměněna).
+x	Neudělá nic (používá se pro lepší srozumitelnost kódu).
abs(x)	Vrátí absolutní hodnotu čísla x.
divmod(x, y)	Vrátí podíl a zbytek po dělení čísla x číslem y jako n-tici dvou celých čísel.
pow(x, y)	Umocní x na mocninu y (stejný výsledek jako operátor **).
pow(x, y, z)	Rychlejší alternativa k `(x ** y) % z`.
round(x, n)	Vrátí číslo x zaokrouhlené na n celočíselných cifer, je-li n záporné celé číslo, nebo na n desetinných míst, je-li n kladné celé číslo. Vrácená hodnota má stejný typ jako x (viz následující text).

N-tice
➢ 28

Typ `tuple`
➢ 110

Tabulka 2.3: Celočíselné převodní funkce

Syntaxe	Popis
bin(i)	Vrátí binární reprezentaci celého čísla `i` ve formě řetězce (např. `bin(1980) == '0b11110111100'`).
hex(i)	Vrátí šestnáctkovou reprezentaci čísla `i` ve formě řetězce (např. `hex(1980) == '0x7bc'`).
int(x)	Převede objekt `x` na celé číslo. V případě chyby vyvolá výjimku `ValueError` nebo výjimku `TypeError`, pokud datový typ objektu `x` nepodporuje převod na celé číslo. Je-li `x` desetinné číslo, tak se ořeže.
int(s, base)	Převede řetězec `s` na celé číslo. V případě chyby vyvolá výjimku `ValueError`. Je-li zadán volitelný argument `base`, pak by měl obsahovat celé číslo v rozmezí od 2 a do 36 včetně.
oct(i)	Vrátí osmičkovou reprezentaci `i` ve formě řetězce (např. `oct(1980) == '0o3674'`).

Celočíselné literály se standardně zapisují s použitím základu 10 (desítková čísla), lze však použít i čísla o jiném základu:

```
>>> 14600926                    # desítkové číslo
14600926
>>> 0b110111101100101011011110  # binární číslo
14600926
>>> 0o67545336                  # osmičkové číslo
14600926
>>> 0xDECADE                    # šestnáctkové číslo
14600926
```

Binární čísla se zapisují s předponou `0b`, osmičková čísla s předponou `0o`* a šestnáctková čísla s předponou `0x`. Pro všechny uvedené předpony lze použít také velká písmena.

Jak je patrné z tabulky 2.2, s celými čísly lze použít všechny obvyklé matematické funkce a operátory. Některé funkce poskytují vestavěné funkce, jako je `abs()` (např. `abs(i)` vrací absolutní hodnotu celého čísla `i`), a jiné poskytují operátory pro typ `int` (např. `i + j` vrátí součet celých čísel `i` a `j`.)

Nyní se podíváme pouze na jednu z funkcí uvedených v tabulce 2.2, protože všechny ostatní jsou dostatečné popsány v samotné tabulce. Funkce `round()` funguje v případě typu `float` očekávaným způsobem (např. `round(1.246, 2)` vrátí `1.25`), avšak v případě typu `int` nemá použití kladné zaokrouhlovací hodnoty žádný efekt a vede k vrácení téhož čísla, poněvadž zde nejsou žádná desetinná místa. Ovšem při použití záporné zaokrouhlovací hodnoty můžeme dosáhnout užitečného chování (např. `round(13579, -3)` vrátí `14000` a `round(34.8, -1)` vrátí `30.0`).

* Uživatelé zvyklí na jazyky ve stylu C si jistě všimli, že pro zápis osmičkového čísla samotná předpona `0` nestačí. V jazyku Python je nutné použít `0o` (nula a písmeno „*o*“).

Veškeré číselné binární operace (+, -, /, //, % a **) mají příslušné verze s rozšířeným přiřazením (+=, -=, /=, //=, %= a **=), kde `x op= y` je v běžném případě, kdy čtení hodnoty `x` nemá žádný vedlejší efekt, logickým ekvivalentem `x = x op y`.

Objekty lze vytvářet přiřazením literálu proměnné (např. `x = 17`) nebo zavoláním relevantního datového typu jako funkce (např. `x = int(17)`). Některé objekty (např. typu `decimal.Decimal`) můžeme vytvářet pouze pomocí datového typu, protože nemají odpovídající literálovou reprezentaci. Při vytvoření objektu pomocí jeho datového typu existují tři možné případy užití.

První případ užití nastává při volání datového typu bez argumentů. V takovém případě se vytvoří objekt s výchozí hodnotou (např. `x = int()` vytvoří celé číslo s hodnotou `0`). Všechny vestavěné typy lze zavolat bez argumentů.

K druhému případu užití dochází tehdy, když se datový typ volá s jedním argumentem. Je-li zadaný argument stejného typu, vytvoří se nový objekt, který je mělkou kopií původního objektu (mělké kopírování budeme probírat v lekci 3). Je-li zadán argument jiného typu, vyzkouší se jeho převod. Tuto situaci jsme si ukázali v tabulce 2.3 pro typ `int`. Je-li argument takového typu, který podporuje převod na daný typ, a tento převod selže, vyvolá se výjimka `ValueError`. V opačném případě se vrátí výsledný objekt daného typu. Pokud datový typ argumentu nepodporuje převod na daný typ, vyvolá se výjimka `TypeError`. Vestavěné typy `float` a `str` poskytují převod na celá čísla. Tento a další převody můžeme implementovat také v našich vlastních datových typech (viz Lekce 6).

Kopírování kolekcí ➤ 146

Převody mezi typy ➤ 250

Třetí případ užití nastává při použití dvou a více argumentů. To však nepodporují všechny typy a u těch, které to podporují, se typy a význam argumentů liší. U typu `int` jsou povoleny dva argumenty. První je řetězec, který představuje celé číslo, a druhý je číselný základ řetězcové reprezentace. Například `int("A4", 16)` vytvoří celé číslo `164` (toto použití ukazuje tabulka 2.3).

Bitové operátory jsou uvedeny v tabulce 2.4. Veškeré bitové binární operace (|, ^, &, << a >>) mají příslušné verze s rozšířeným přiřazením (|=, ^=, &=, <<= a >>=), kde `i op= j` je v běžném případě, kdy čtení hodnoty `i` nemá žádný vedlejší efekt, logickým ekvivalentem `i = i op j`.

3.1

Od Pythonu ve verzi 3.1 je k dispozici metoda `int.bit_length()`, která vrací počet bitů nezbytných k reprezentaci daného celého čísla. Například `(2145).bit_length()` vrátí 12. (Závorky jsou při použití celočíselného literálu nezbytné. V případě celočíselné proměnné je však můžeme vypustit.)

Pokud potřebujeme uchovávat mnoho příznaků pravda/nepravda, pak můžeme použít jediné celé číslo a testovat jednotlivé bity pomocí bitových operátorů. Toho stejného lze docílit sice méně kompaktním, ale zato pohodlnějším způsobem – pomocí seznamu logických hodnot.

Tabulka 2.4: Celočíselné bitové operátory

Syntaxe	Popis
i | j	Bitová disjunkce (OR) celých čísel `i` a `j`. Záporná čísla jsou reprezentována pomocí dvojkového doplňkového kódu.
i ^ j	Bitová nonekvivalence (XOR) čísel `i` a `j`.
i & j	Bitová konjunkce (AND) čísel `i` a `j`.

Syntaxe	Popis
i << j	Posune číslo `i` doleva o `j` bitů. Stejné jako `i * (2 ** j)`, ale bez kontroly přetečení.
i >> j	Posune číslo `i` doprava o `j` bitů. Stejné jako `i // (2 ** j)`, ale bez kontroly přetečení.
~i	Obrátí bity čísla `i`.

Logické hodnoty

Existují dva vestavěné Booleovské objekty: `True` a `False`. Podobně jako všechny ostatní datové typy jazyka Python (ať už vestavěné, knihovní nebo vlastní), také datový typ `bool` lze volat jako funkci. Bez argumentů vrací hodnotu `False`, s argumentem typu `bool` vrací jeho kopii a s libovolným dalším argumentem se pokusí daný objekt převést na typ `bool`. Všechny vestavěné datové typy a datové typy standardní knihovní lze převést na logickou (Booleovskou) hodnotu, přičemž je snadné implementovat převod na logickou hodnotu i ve vlastních datových typech. Zde je několik Booleovských přiřazení a výrazů:

```
>>> t = True
>>> f = False
>>> t and f
False
>>> t and True
True
```

Logické operátory ➤ 34

Jak jsme si řekli již dříve, jazyk Python nabízí tři logické operátory: `and`, `or` a `not`. Operátory `and` a `or` používají logiku zkráceného vyhodnocování a vracejí operand, který rozhodl o výsledku. To znamená, že ne vždy vracejí `True` nebo `False`.

Programátoři, kteří jsou zvyklí na starší verzi Pythonu, používají někdy místo hodnot `True` a `False` hodnoty `1` a `0`. Tento přístup téměř vždy funguje bez problémů, ale nový kód by měl v místech, kde se vyžaduje logická hodnota, používat vestavěné booleovské objekty.

Typy s pohyblivou řádovou čárkou

Jazyk Python nabízí tři druhy hodnot s pohyblivou řádovou čárkou: vestavěné typy `float` a `complex` a typ `decimal.Decimal` ze standardní knihovny. Všechny tři uvedené typy jsou neměnitelné. Typ `float` uchovává čísla s pohyblivou řádovou čárkou s dvojitou přesností, jejichž rozsah závisí na kompilátoru jazyka C (nebo C# nebo Java) použitém pro sestavení Pythonu. Tato čísla mají omezenou přesnost a nelze je spolehlivě porovnávat na rovnost. Čísla typu `float` se zapisují s desetinnou tečkou nebo pomocí exponenciální notace (např. `0.0`, `4.`, `5.7`, `-2.5`, `-2e9`, `8.9e-4`).

Počítače nativně reprezentují čísla s pohyblivou řádovou čárkou pomocí kódování base 2, což znamená, že některá desetinná čísla lze reprezentovat přesně (např. 0,5), ale jiná jen přibližně (např. 0,1 nebo 0,2). Tato reprezentace dále používá fixní počet bitů, takže počet uchovávaných číslic je omezen. Zde je zajímavý příklad zapsaný do editoru IDLE:

3.0

```
3.0 >>> 0.0, 5.4, -2.5, 8.9e-4
(0.0, 5.4000000000000004, -2.5, 0.0008899999999999995)
```

Nepřesnost není problém, který by se týkal pouze jazyka Python. Všechny programovací jazyky trpí tímto problémem s čísly s pohyblivou řádovou čárkou.

3.1

Python 3.1 vrací mnohem smysluplněji vypadající výstup:

```
>>> 0.0, 5.4, -2.5, 8.9e-4
(0.0, 5.4, -2.5, 0.00089)
```

Když Python 3.1 vypisuje číslo s pohyblivou řádovou řádkou, pak ve většině případů používá algoritmus Davida Gaye. Ten vypisuje nejmenší možný počet číslic bez ztráty přesnosti. I když takto získáme hezčí výstup, nic tím nezměníme na faktu, že počítače (bez ohledu na použitý počítačový jazyk) v podstatě ukládají čísla s pohyblivou řádovou čárkou jako aproximace.

Pokud potřebujeme opravdu vysokou přesnost, pak můžeme postupovat dvěma způsoby. Jeden z nich spočívá ve využití typů `int` (např. pro práci s desetinami či setinami) a jejich škálování v případě potřeby. To vyžaduje jistou opatrnost, zejména pak tehdy, když provádíme dělení nebo počítáme procentní podíl. Další možností je využít typ `decimal.Decimal` z modulu `decimal`. Čísla tohoto typu provádějí výpočty, které jsou přesné až do námi stanovené úrovně (ve výchozím stavu až na 28 desetinných míst). Tato čísla mohou navíc přesným způsobem reprezentovat periodická čísla jako 0,1. Jejich zpracování je ale ve srovnání s čísly typu `float` mnohem pomalejší. Díky své přesnosti jsou čísla typu `decimal.Decimal` vhodná pro finanční výpočty.

Podporovány jsou též aritmetické operace se smíšenými typy, takže při použití typů `int` a `float` obdržíme typy `float` a při použití typů `float` a `complex` obdržíme `complex`. Čísla typu `decimal.Decimal` mají fixní přesnost, a proto je lze použít pouze s čísly typu `decimal`. Pokud s těmito čísly použijeme celá čísla (`int`), obdržíme výsledek typu `decimal.Decimal`. Pokud se pokusíme provést nějakou operaci s nekompatibilními typy, vyvolá se výjimka `TypeError`.

Čísla s pohyblivou řádovou čárkou

Veškeré číselné operace a funkce v tabulce 2.2 (strana 61) lze použít i s typem `float`, a to včetně verzí s rozšířením přiřazením. Datový typ `float` je možné zavolat i jako funkci. Bez argumentů vrátí 0.0, s argumentem typu `float` vrátí jeho kopii a argument jiného typu se pokusí převést na typ `float`. Při použití pro převod lze použít i řetězcový argument buď v obyčejné desetinné notaci, nebo s použitím exponenciální notace. Při výpočtech s čísly typu `float` můžeme obdržet výsledek `NaN` (Not a Number – není číslo) nebo „nekonečno“. Toto chování však není konzistentní napříč jednotlivými implementacemi a může se lišit v závislosti na systémem používané matematické knihovně.

Zde je jednoduchá funkce pro porovnání čísel typu `float` na rovnost omezená pouze přesností daného počítače:

```
def equal_float(a, b):
    return abs(a - b) <= sys.float_info.epsilon
```

Pro tento kód musíme importovat modul `sys`. Objekt `sys.float_info` nabízí celou řadu atributů. Atribut `sys.float_info.epsilon` je v podstatě nejmenším rozdílem dvou čísel s pohyblivou

řádovou čárkou, který dokáže daný stroj rozlišit. Na jistém 32bitovém stroji je to jen něco přes 0,000 000 000 000 000 2. (Toto číslo se tradičně označuje jako epsilon.) Typ `float` jazyka Python nabízí běžně spolehlivou přesnost až po 17 významných číslic.

Pokud napíšete objekt `sys.float_info` do editoru IDLE, zobrazí se všechny jeho atributy. Patří mezi ně minimální a maximální čísla s pohyblivou řádovou čárkou, která dokáže daný stroj reprezentovat. Pokud napíšete `help(sys.float_info)`, obdržíte určité informace o objektu `sys.float_info`.

Čísla s pohyblivou řádovou čárkou lze převést na celá čísla buď pomocí funkce `int()`, která vrátí celou část a desetinnou část zahodí, nebo pomocí funkce `round()`, která zohlední desetinnou část, anebo pomocí funkce `math.floor()`, resp. `math.ceil()`, která vrátí nejbližší menší resp. větší celé číslo. Metoda `float.is_integer()` vrátí hodnotu `True`, je-li zlomková část 0, přičemž zlomkovou reprezentaci lze získat pomocí metody `float.as_integer_ratio()`. Předpokládejme, že `x = 2.75`, pak volání `x.as_integer_ratio()` vrátí `(11, 4)`. Celá čísla je možné převést na čísla s pohyblivou řádovou čárkou pomocí funkce `float()`.

Čísla s pohyblivou řádovou čárkou lze reprezentovat jako řetězce v šestnáctkovém formátu pomocí metody `float.hex()`. Takové řetězce lze pak převést zpět na čísla s pohyblivou řádovou čárkou metodou `float.fromhex()`:

```
s = 14.25.hex()       # str s == '0x1.c800000000000p+3'
f = float.fromhex(s)  # float f == 14.25
t = f.hex()           # str t == '0x1.c800000000000p+3'
```

Exponent je místo písmena `e` označen písmenem `p` (power – mocnina), protože `e` je platná šestnáctková číslice.

Tabulka 2.5: Funkce a konstanty modulu `Math`

Syntaxe	Popis
math.acos(x)	Vrátí arkus kosinus x (v radiánech).
math.acosh(x)	Vrátí hyperbolický arkus kosinus x (v radiánech).
math.asin(x)	Vrátí arkus sinus x (v radiánech).
math.asinh(x)	Vrátí hyperbolický arkus sinus x (v radiánech).
math.atan(x)	Vrátí arkus tangens x (v radiánech).
math.atan2(y, x)	Vrátí arkus tangens x / y (v radiánech).
math.atanh(x)	Vrátí hyperbolický arkus tangens x (v radiánech).
math.ceil(x)	Vrátí $\lceil x \rceil$, tj. nejmenší celé číslo (`int`) větší nebo stejné jako x (např. `math.ceil(5.4) == 6`).
math.copysign(x,y)	Vrátí číslo x se znaménkem čísla y.
math.cos(x)	Vrátí kosinus x (v radiánech).

Syntaxe	Popis
math.cosh(x)	Vrátí hyperbolický kosinus x (v radiánech).
math.degrees(r)	Převede číslo r typu `float` z radiánů na stupně.
math.e	Konstanta e (přibližně `2.7182818284590451`).
math.exp(x)	Vrátí e^x, tj. `math.e**x`.
math.fabs(x)	Vrátí $\lvert x \rvert$, tj. absolutní hodnotu x jako typ `float`.
math.factorial(x)	Vrátí x!
math.floor(x)	Vrátí $\lfloor x \rfloor$, tj. největší celé číslo (`int`) menší nebo stejné jako x (např. `math.floor(5.4) == 5`).
math.fmod(x, y)	Vrátí modul (zbytek) po dělení čísla x číslem y (pro čísla typu `float` vrací lepší výsledky než operátor `%`).
math.frexp(x)	Vrátí dvouprvkovou n-tici s mantisou (jako typ `float`) a exponentem (jako typ `int`) tak, že $x = m \times 2^e$ (viz `math.ldexp()`).
math.fsum(i)	Vrátí součet hodnot v iterovatelném objektu i jako číslo typu `float`.
math.hypot(x, y)	Vrátí odmocninu z $\sqrt{x^2 + y^2}$.
math.isinf(x)	Vrátí hodnotu `True`, pokud číslo x typu `float` je ±`inf(±∞)`.
math.isnan(x)	Vrátí hodnotu `True`, pokud číslo x typu `float` je NaN (není číslo).
math.ldexp(m, e)	Vrátí $m \times 2^e$, což je v podstatě inverze funkce `math.frexp()`.
math.log(x, b)	Vrátí $\log_b x$ (b je volitelné a jeho výchozí hodnota je `math.e`).
math.log10(x)	Vrátí $\log_{10} x$.
math.log1p(x)	Vrátí $\log_e(1+x)$ (přesné i v případě, kdy je x blízko 0).
math.modf(x)	Vrátí zlomkovou a celou část čísla x jako dvě čísla typu `float`.
math.pi	Konstanta π (přibližně 3.1415926535897931).
math.pow(x, y)	Vrátí x^y jako číslo typu `float`.
math.radians(d)	Převede číslo d typu `float` ze stupňů na radiány.
math.sin(x)	Vrátí sinus x (v radiánech).
math.sinh(x)	Vrátí hyperbolický sinus x (v radiánech).
math.sqrt(x)	Vrátí odmocninu z čísla $\sqrt{x}$.

N-tice ➤ 28

Typ `tuple` ➤ 110

Syntaxe	Popis
math.tan(x)	Vrátí tangens čísla x (v radiánech).
math.tanh(x)	Vrátí hyperbolický tangens čísla x (v radiánech).
math.trunc(x)	Vrátí celou část x jako číslo typu `int` (stejné jako `int(x)`).

Jak je patrné z tabulky 2.5, nabízí modul `math` kromě vestavěných funkcí pro práci s pohyblivou řádovou čárkou ještě spoustu dalších funkcí, které pracují na těchto číslech. Zde je několik úryvků kódu, které ukazují, jak se používají funkce tohoto modulu:

3.x

```
>>> import math
>>> math.pi * (5 ** 2)  # Python 3.1 vypíše: 78.53981633974483
78.539816339744831
>>> math.hypot(5, 12)
13.0
>>> math.modf(13.732)   # Python 3.1 vypíše: (0.7319999999999993, 13.0)
(0.73199999999999932, 13.0)
```

Funkce `math.hypot()` vypočítá vzdálenost od počátku k bodu (x, y), takže vrátí stejný výsledek jako `math.sqrt((x ** 2) + (y ** 2))`.

Modul `math` je velice závislý na matematické knihovně, s níž byl Python zkompilován. To znamená, že některé chybové podmínky a hraniční případy se mohou na různých platformách lišit.

Komplexní čísla

Typ `complex` je neměnitelný datový typ uchovávající dvojici čísel typu `float`, kdy jedno reprezentuje reálnou a druhé imaginární složku komplexního čísla. Literály komplexních čísel se zapisují s reálnou a imaginární složkou spojenou znaménkem + nebo -, přičemž za imaginární složkou se nachází písmeno `j`.* Zde je několik příkladů: `3.5+2j`, `0.5j`, `4+0j` a `-1-3.7j`. Všimněte si, že pokud je reálná část, tak ji můžeme zcela vypustit.

Jednotlivé složky komplexního čísla jsou přístupné jako atributy `real` a `imag`:

```
>>> z = -89.5+2.125j
>>> z.real, z.imag
(-89.5, 2.125)
```

Až na operace `//`, `%`, `divmod()` a funkci `pow()` se třemi argumenty lze s komplexními čísly použít všechny číselné operace a funkce uvedené v tabulce 2.2 (strana 61) včetně verzí s rozšířeným přiřazením. Kromě toho nabízí typ `complex` metodu `conjugate()`, která změní znaménko imaginární části:

```
>>> z.conjugate()
(-89.5-2.125j)
>>> 3-4j.conjugate()
(3+4j)
```

* Matematici mohou pro označení $\sqrt{-1}$ používat i, avšak Python dodržuje označení j tradiční pro inženýrské prostředí.

Všimněte si, že jsme metodu zavolali na literálovém komplexním čísle. Python nám obvykle umožňuje volat metody nebo přistupovat k atributům na libovolném literálu, pokud datový typ literálu poskytuje volanou metodu nebo atribut. Nicméně to se netýká speciálních metod, protože ty mají vždy odpovídající operátory, jako je +, které by se měly použít místo nich. Například `4j.real` vrátí `0.0`, `4j.imag` vrátí `4.0` a `4j + 3+2j` vrátí `3+6j`.

Datový typ `complex` lze volat jako funkci. Bez parametrů vrátí `0j`, s argumentem typu `complex` vrátí jeho kopii a argument jiného typu se pokusí převést na typ `complex`. Při použití pro převod přijímá funkce `complex()` buď jediný řetězec, nebo jedno či dvě čísla typu `float`. Je-li zadáno pouze jedno, pak je imaginární část nastavena na `0j`.

Funkce v modulu `math` s komplexními čísly nefungují. Jedná se o úmyslné návrhové rozhodnutí, které zajišťuje, aby uživatelé modulu `math` neobdrželi v některých situacích nevědomě komplexní čísla, ale aby se raději vyvolaly výjimky.

Uživatelé komplexních čísel mohou importovat modul `cmath`, který nabízí verze většiny trigonometrických a logaritmických funkcí modulu `math` pro komplexní čísla plus několik další funkcí specifických pro komplexní čísla, jako jsou například `cmath.phase()`, `cmath.polar()` a `cmath.rect()`, a také konstanty `cmath.pi` a `cmath.e`, které uchovávají stejné hodnoty typu `float` jako jejich protějšky v modulu `math`.

Desetinná čísla

V mnoha aplikacích nepředstavují nepřesnosti, které se mohou vyskytnout při použití čísel typu `float`, žádný problém a tak jako tak jsou vyváženy rychlostí výpočtu nabízené typem `float`. Ovšem v některých případech potřebujeme dát přednost přesnosti, a to i za cenu zpomalení výpočtu. Modul `decimal` poskytuje neměnitelný typ `Decimal`. Čísla tohoto typu jsou tak přesná, jak si sami stanovíme. Výpočty zahrnující čísla typu `Decimal` jsou ve srovnání s čísly typu `float` pomalejší, ale zda to bude postřehnutelné, závisí na konkrétní aplikaci.

K vytvoření čísla typu `Decimal` musíme nejdříve importovat modul `decimal`:

```
>>> import decimal
>>> a = decimal.Decimal(9876)
>>> b = decimal.Decimal("54321.012345678987654321")
>>> a + b
Decimal('64197.012345678987654321')
```

Desetinná čísla se vytvářejí pomocí funkce `decimal.Decimal()`. Tato funkce přijímá celé číslo nebo řetězec, ne však číslo typu `float`, protože tato čísla jsou uchovávána nepřesně, zatímco desetinná čísla jsou reprezentována přesně. Pokud použijeme řetězec, můžeme použít desetinnou nebo exponenciální notaci. Přesná reprezentace čísel typu `decimal.Decimals` znamená kromě vyšší přesnosti také schopnost porovnávání na rovnost.

Počínaje Pythonem ve verzi 3.1 je možné převádět čísla typu `float` na čísla typu `decimal` funkcí `decimal.Decimal.from_float()`. Tato funkce přijímá číslo typu `float` a vrací číslo typu `decimal.Decimal`, které je nejblíže číslu, které aproximuje zadaný argument.

S typem `decimal.Decimal` lze použít všechny číselné operace a funkce uvedené v tabulce 2.2 (strana 61) včetně verzí s rozšířeným přiřazením. Je tu však několik pravidel, která je nutné dodržovat. Pokud je levý operand operátoru `**` typu `decimal.Decimal`, pak pravý operand musí být celé číslo. A podobné je to i u funkce `pow()`. Je-li její první argument typu `decimal.Decimal`, pak její druhý i volitelný třetí argument musí být celé číslo.

Moduly `math` a `cmath` nejsou vhodné pro použití s typem `decimal.Decimal`, avšak některé funkce poskytované modulem `math` jsou k dispozici jako metody typu `decimal.Decimal`. Například pro výpočet e^x, kde `x` je typu `float`, napíšeme `math.exp(x)`. Je-li však `x` typu `decimal.Decimal`, pak napíšeme `x.exp()`. Z diskuze v Oblasti č. 3 předchozí lekce (strana 29) je patrné, že metoda `x.exp()` je v podstatě syntaktickým cukrem pro metodu `decimal.Decimal.exp(x)`.

Datový typ `decimal.Decimal` nabízí také metodu `ln()`, která počítá přirozený (o základu e) logaritmus (podobně jako funkce `math.log()` s jedním argumentem), `log10()` a `sqrt()` společně s řadou dalších metod specifických pro datový typ `decimal.Decimal`.

Čísla typu `decimal.Decimal` pracují v rámci určitého kontextu. Tímto kontextem je kolekce nastavení, jež ovlivňují způsob chování těchto čísel. Kontext stanoví přesnost, která by se měla používat (výchozí je 28 desetinných míst), zaokrouhlovací techniku a některé další detaily.

V některých situacích je rozdíl v přesnosti mezi čísly typu `float` a `decimal.Decimal` zřejmý:

```
>>> 23 / 1.05
21.904761904761905
>>> print(23 / 1.05)
21.9047619048
>>> print(decimal.Decimal(23) / decimal.Decimal("1.05"))
21.904761904761904761904761904761
>>> decimal.Decimal(23) / decimal.Decimal("1.05")
Decimal('21.904761904761904761904761904761')
```

Ačkoliv dělení pomocí typu `decimal.Decimal` je mnohem přesnější než s použitím typu `float`, v tomto případě (32bitový stroj) se rozdíl objevil až na patnáctém desetinném místě. V mnoha situacích je tak malý rozdíl nepodstatný, a proto budeme ve všech příkladech v této knize používat pro čísla s pohyblivou řádovou čárkou typ `float`.

Další věcí, na kterou je třeba upozornit, je skutečnost, že poslední dva výše uvedené příklady poprvé odhalují, že při vypisování objektu dochází v pozadí k určitému formátování. Když se zavolá funkce `print()` na výsledku `decimal.Decimal(23) / decimal.Decimal("1.05")`, vypíše se holé číslo. Tento výstup tedy probíhá v řetězcové formě. Pokud jednoduše zadáme výraz, pak obdržíme výstup z typu `decimal.Decimal`. Jedná se tedy o výstup v reprezentační formě. Všechny objekty jazyka Python mají dvě formy výstupu. Řetězcová forma je navržena tak, aby byla čitelná pro člověka. Reprezentační forma je navržena pro vytvoření výstupu, který po předložení interpretu jazyka Python (je-li to možné) povede k vytvoření reprezentovaného objektu. K tomuto tématu se ještě vrátíme v následující části, v níž se budeme věnovat řetězcům, a pak znovu v lekci 6, kde budeme probírat implementaci řetězcové a reprezentační formy pro naše vlastní datové typy.

Veškeré podrobnosti ohledně modulu `decimal` (včetně dalších příkladů a seznamu otázek a odpovědí), které jsou nad rámec této knihy, najdete v příslušné dokumentaci.

Řetězce

Řetězce představují neměnitelný datový typ `str`, který uchovává posloupnost znaků ze znakové sady Unicode. Datový typ `str` lze pro vytvoření řetězcových objektů zavolat také jako funkci. Bez argumentů vrátí prázdný řetězec, s neřetězcovým argumentem vrátí jeho řetězcovou podobu a s řetězcovým argumentem vrátí jeho kopii. Funkci `str()` lze též použít jako převodní funkci. V takovém případě by prvním argumentem měl být řetězec nebo něco, co lze na řetězec převést. Pak mohou následovat další dva nepovinné argumenty, z nichž jeden stanoví kódování, které se má použít, a druhý způsob zpracování kódovacích chyb.

Kódování znaků ➢ 95

Již dříve jsme si řekli, že se řetězcové literály vytvářejí pomocí uvozovek a že je jen na nás, zda použijeme jednoduché či dvojité uvozovky, jsou-li na obou stranách stejné. Kromě toho můžeme použít řetězec s trojitými uvozovkami, což v řeči jazyka Python znamená, že začíná a končí *třemi uvozovkami* (ať už jednoduchými nebo dvojitými):

```
text = """Řetězec s trojitými uvozovkami může obsahovat 'uvozovky' a
také "uvozovky" bez jakýchkoli formalit. Můžeme též potlačit nové řádky \
tkaže tento řetězec bude mít pouze dva řádky."""
```

Pokud bychom chtěli použít uvozovky uvnitř běžně uzavřeného řetězce, pak je můžeme zapsat přímo, pokud se liší od ohraničujících uvozovek. V opačném případě je musíme potlačit zpětným lomítkem:

```
a = "Jednoduché 'uvozovky' jsou v pořádku; \"dvojité\" musíme potlačit."
b = 'Jednoduché \'uvozovky\' musíme potlačit; "dvojité" jsou v pořádku.'
```

Tabulka 2.6: Speciální řetězcové posloupnosti v jazyku Python

Speciální řetězcová posloupnost	Popis
`\`*`newline`*	Potlač (tj. ignoruj) nový řádek
`\\`	Zpětné lomítko (\)
`\'`	Jednoduché uvozovky (')
`\"`	Dvojité uvozovky (")
`\a`	Zvonek ve znakové sadě ASCII (BEL)
`\b`	Zpětné lomítko ve znakové sadě ASCII (BS – Backspace)
`\f`	Posun na další stránku ve znakové sadě ASCII (FF – Form Feed)
`\n`	Posun na další řádek ve znakové sadě ASCII (LF – Line Feed)
`\N{`*`název`*`}`	Znak se zadaným názvem ze znakové sady Unicode

Speciální řetězcová posloupnost	Popis
`\ooo`	Znak se zadanou osmičkovou hodnotou
`\r`	Návrat tiskové hlavy na začátek ve znakové sadě ASCII (CR – Carriage Return)
`\t`	Tabulátor ve znakové sadě ASCII (TAB)
`\u`*hhhh*	Znak se zadanou 16bitovou hexadecimální hodnotou ze znakové sady Unicode
`\U`*hhhhhhhh*	Znak se zadanou 32bitovou hexadecimální hodnotou ze znakové sady Unicode
`\v`	Vertikální tabulátor ve znakové sadě ASCII (VT – Vertical Tab)
`\x`*hh*	Znak se zadanou 8bitovou hexadecimální hodnotou

Python používá nové řádky jako terminátor příkazů. Výjimkou je vnitřek kulatých závorek (`()`), hranatých závorek (`[]`), složených závorek (`{}`) nebo vnitřek řetězců s trojitými uvozovkami. Nové řádky lze v řetězcích s trojitými uvozovkami použít přímo, přičemž do libovolného řetězcového literálu můžeme vložit nový řádek pomocí speciální posloupnosti `\n`. Všechny speciální řetězcové posloupnosti jazyka Python uvádí tabulka 2.6. V některých situacích (např. při psaní regulárních výrazů) potřebujeme vytvořit řetězec se spoustou zpětných lomítek (regulární výrazy jsou předmětem lekce 13). To může být nepohodlné, protože každé musíme dalším zpětným lomítkem potlačit:

```
import re
phone1 = re.compile("^((?:[(]\\d+[)])?\\s*\\d+(?:-\\d+)?)$")
```

Řešením jsou *holé* řetězce. Jedná se libovolným způsobem ohraničené řetězce, které mají před první uvozovkou umístěno písmeno `r`. Uvnitř takovýchto řetězců jsou všechny znaky považovány za literály, takže není nutné používat speciální řetězcové posloupnosti. Zde je regulární výraz pro telefon napsaný jako holý řetězec.

```
phone2 = re.compile(r"^((?:[(]\d+[)])?\s*\d+(?:-\d+)?)$")
```

Pokud chceme bez použití trojitých uvozovek napsat dlouhý řetězcový literál rozprostřený na dva či více řádků, můžeme použít jednu z následujících možností:

```
t = "Toto není nejvhodnější způsob spojení dvou dlouhých řetězců, " + \
    "protože se opírá o nevzhledné potlačení nového řádku"

s = ("Toto je pěkný způsob spojení dvou dlouhých řetězců, "
     "který se opírá o řetězení řetězcových literálů.")
```

Všimněte si, že ve druhém případě musíme pro vytvoření jediného výrazu použít závorky. Bez nich by se proměnné `s` přiřadil pouze první řetězec a druhý by způsobil výjimku `IndentationError`. Jistá část dokumentace jazyka Python (*http://docs.python.org/dev/howto/doanddont.html*) doporučuje používat pro příkazy jakéhokoliv druhu rozprostřené na více řádků vždy závorky, ne potlačování nových řádků. Tímto doporučením se zde budeme řídit.

Vzhledem k tomu, že soubory `.py` používají ve výchozím nastavení kódování UTF-8, můžeme do svých řetězcových literálů psát libovolné znaky přímo. Můžeme také do řetězců umístit libovolné znaky ze znakové sady Unicode pomocí hexadecimální speciální posloupnosti nebo pomocí názvů znakové sady Unicode:

```
>>> euros = " \N{euro sign} \u20AC \U000020AC"
>>> print(euros)
```

V tomto případě bychom nemohli použít hexadecimální speciální posloupnost, protože ty jsou omezeny na dvě číslice, takže nemohou přesáhnout hodnotu `0xFF`. Je třeba poznamenat, že u názvů znaků ve znakové sadě Unicode se nerozlišuje velikost písmen a mezery, které obsahují, jsou volitelné.

Pokud chceme pro určitý znak v řetězci zjistit jeho kódový bod ve znakové sadě Unicode (celočíselná hodnota přiřazená danému znaku v kódování Unicode), můžeme použít vestavěnou funkci `ord()`:

Kódování znaků ➤ 95

```
>>> ord(euros[0])
8364
>>> hex(ord(euros[0]))
'0x20ac'
```

Podobně můžeme pomocí vestavěné `chr()` funkce převést libovolné celé číslo, jež reprezentuje platný kódový bod, na odpovídající znak ve znakové sadě Unicode:

```
>>> s = "anarchitsti jsou " + chr(8734) + chr(0x23B7)
>>> s
'anarchists are ∞⎷'
>>> ascii(s)
"'anarchists are \u221e\u23b7'"
```

Pokud v editoru IDLE zadáme jen samotné `s`, pak obdržíme výstup v jeho řetězcové formě, což u řetězců znamená výstup znaků uzavřených do uvozovek. Pokud chceme pouze znaky ze znakové sady ASCII, můžeme použít vestavěnou funkci `ascii()`, která vrátí reprezentační formu svého argumentu pomocí 7bitových znaků ASCII všude, kde je to možné, a pomocí co nejkratší formy speciální řetězcové posloupnosti `\xhh`, `\uhhhh` nebo `\Uhhhhhhhh` ve všech ostatních případech. O způsobu přesné kontroly výstupu řetězců si více řekneme v pozdější části této lekce.

str.format() ➤ 83

Porovnávání řetězců

Řetězce podporují obvyklé porovnávací operátory `<`, `<=`, `==`, `!=`, `>` a `>=`. Tyto operátory porovnávají řetězce po jednotlivých bajtech v paměti. Při porovnávání (např. při řazení seznamu řetězců) však mohou vyvstat dva problémy, které nepostihují jen Python, ale každý programovací jazyk používající řetězce s kódováním Unicode.

Prvním problém je, že některé znaky znakové sady Unicode lze reprezentovat dvěma či více různými posloupnostmi bajtů. Například znak A (kódový bod `0x00C5` ve znakové sadě Unicode) může být reprezentován bajty v kódování UTF-8 třemi různými způsoby: `[0xE2, 0x84, 0xAB]`, `[0xC3, 0x85]` a `[0x41, 0xCC, 0x8A]`. Tento problém naštěstí můžeme vyřešit. Pokud importujeme modul

Kódování znaků ➤ 95

`unicodedata` a zavoláme metodu `unicodedata.normalize()`, které jako argumenty předáme `"NFKC"` a řetězec obsahující znak A zakódovaný v libovolné platné posloupnosti, pak obdržíme řetězec, který bude v kódování UTF-8 vždy obsahovat pro znak A posloupnost bajtů `[0xC3, 0x85]`.

Druhý problém tkví v tom, že řazení některých znaků je závislé na konkrétním jazyku. Jako příklad můžeme uvést písmeno „ä“, které je ve Švédsku řazeno za písmenem „z“, kdežto v Německu je řazeno jako „ae“. Dalším příkladem je angličtina, kde se písmeno „o“ řadí standardním způsobem, zatímco v Dánsku a Norsku se řadí až za písmeno „z“. Těchto problémů je celá řada a mohou být dále zkomplikovány tím, když jednu aplikaci používají lidé různých národností (kteří tím pádem očekávají odlišné pořadí při řazení) nebo když řetězce obsahují směsici více jazyků (např. něco v češtině, něco v angličtině), přičemž některé znaky (např. šipky, ozdobné znaky či matematické symboly) nemají ve skutečnosti žádnou smysluplnou pozici pro řazení.

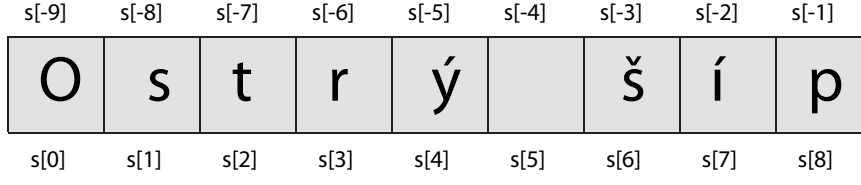
Python ze zásady (aby zamezil zákeřným chybám) nedělá žádné odhady. V případě porovnávání řetězců se tedy používá bajtová reprezentace řetězců v paměti. Řetězce se tak řadí podle kódových bodů znakové sady Unicode, což v případě angličtiny znamená řazení podle kódování ASCII. Při porovnávání s malými a velkými písmeny proto v angličtině obdržíme mnohem přirozenější uspořádání. Normalizace většinou není potřeba, pokud řetězce nejsou z externího zdroje, jako jsou soubory nebo síťové sokety. Avšak ani v těchto případech by se normalizace neměla provádět, nemáme-li jistotu, že je skutečně zapotřebí. Řadicí metody Pythonu si samozřejmě můžeme přizpůsobit, o čemž se přesvědčíme v lekci 3. Celý problém řazení řetězců v kódování Unicode je podrobně vysvětlen v dokumentu s názvem Unicode Collation Algorithm (viz *www.unicode.org/reports/tr10/*).

Řezání a krokování řetězců

Oblast č. 3
➢ 28

V Oblasti č. 3 jsme se dozvěděli, že jednotlivé prvky v posloupnosti, a tedy i znaky v řetězci lze extrahovat pomocí operátoru pro přístup k prvku (`[]`). Ve skutečnosti je tento operátor mnohem všestrannější a je možné jej použít pro extrakci ne jen jediného prvku či znaku, ale pro celý řez (podposloupnost) prvků či znaků. V tomto kontextu jej proto označujeme jako řezací (slicing) operátor.

Nejdříve se podíváme na extrahování jednotlivých znaků. Pozice indexů v řetězci začínají od 0 a pokračují až do délky řetězce minus 1. Můžeme nicméně použít i zápornou pozici indexu. Tyto pozice se počítají od posledního znaku zpět k prvnímu. Mějme přiřazení `s = "Ostrý šíp"`. Pak obrázek 2.1 ukazuje všechny platné pozice indexů pro řetězec `s`.



Obrázek 2.1: Pozice indexů v řetězci

Záporné indexy jsou překvapivě užitečné, zejména index -1, který vždy ukazuje na poslední znak v řetězci. Přístup k indexu mimo rozsah (nebo k libovolnému indexu v prázdném řetězci) způsobí vyvolání výjimky `IndexError`.

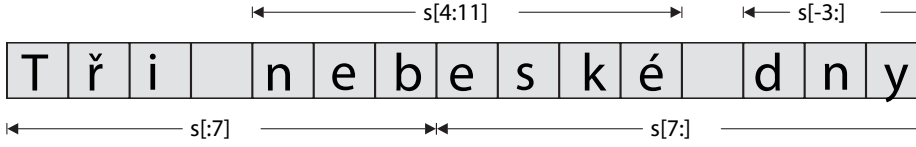
Řezací operátor má tři varianty syntaxe:

```
seq[začátek]
seq[začátek:konec]
seq[začátek:konec:krok]
```

Symbol *seq* představuje libovolnou posloupnost, jako je seznam, řetězec nebo n-tice. Hodnoty *začátek*, *konec* a *krok* jsou vždy celá čísla (nebo proměnné uchovávající celá čísla). První variantu syntaxe jsme již používali. Tato varianta extrahuje prvek posloupnosti na pozici *začátek*. Druhá varianta syntaxe extrahuje řez, jehož počáteční prvek je na pozici *začátek*, a koncový prvek *před* prvkem na pozici *konec*. Ke třetí variantě se dostaneme za okamžik.

Pokud použijeme druhou variantu syntaxe (s jednou dvojtečkou), pak můžeme kterýkoli z indexů vynechat. Vynecháme-li začáteční index, bude mít výchozí hodnotu 0. Pokud vynecháme koncový index, bude mít výchozí hodnotu `len(seq)`. To znamená, že pokud vynecháme oba indexy, tedy například `s[:]`, je to stejné, jako kdybychom napsali `s[0:len(s)]`, čímž extrahujeme (tj. zkopírujeme) celou sekvenci.

Mějme přiřazení `s = "Tři nebeské dny"`. Pak obrázek 2.2 znázorňuje několik ukázkových řezů pro řetězec `s`.



Obrázek 2.2: Řezání posloupnosti

Jednou z možností, jak vložit do řetězce podřetězec, je spojit řezání se zřetězením:

```
>>> s = s[:12] + "be" + s[12:]
>>> s
'Tři nebeské bedny'
```

Vzhledem k tomu, že se text „be“ objevuje také v původním řetězci, můžeme stejného výsledku docílit přiřazením `s[:12] + s[6:8] + s[12:]`.

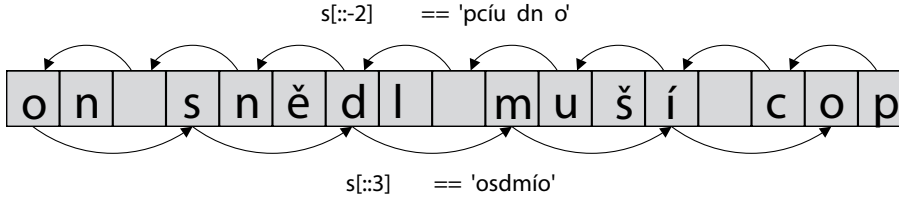
Používání operátoru + pro spojení a += pro připojení není příliš efektivní, pokud pracujeme s více řetězci. Pro spojování většího počtu řetězců je obvykle nejlepší sáhnout po metodě `str.joint()`, o které si více řekneme v následující podčásti.

Třetí varianta syntaxe řezacího operátoru (se dvěma dvojtečkami) je podobná druhé, pouze se místo všech znaků extrahuje vždy jen i-tý znak, kde i je definováno zadaným krokem. A podobně jako ve druhé variantě syntaxe můžeme i zde kterýkoliv index vynechat. Pokud vynecháme počáteční index, bude mít výchozí hodnotu 0, pokud ovšem není zadán záporný krok. V takovém případě bude mít výchozí hodnotu -1. Pokud vynecháme koncový index, bude mít výchozí hodnotu `len(seq)`. Je-li ovšem krok záporný, pak bude mít koncový index výchozí hodnotu ležící před začátkem řetězce. Pokud použijeme dvě dvojtečky, ale vynecháme velikost kroku, pak se pro krok použije výchozí hodnota 1. Je však naprosto zbytečné používat dvě dvojtečky s krokem velikosti 1, protože se tak jako tak jedná o výchozí hodnotu kroku. Dále je třeba si uvědomit, že krok velikosti nula není povolen.

Řetězcové operátory a metody ➢ 76

Mějme přiřazení `s = "on snědl muší cop"`. Pak obrázek 2.3 znázorňuje několik ukázkových krokování pro řetězec `s`.

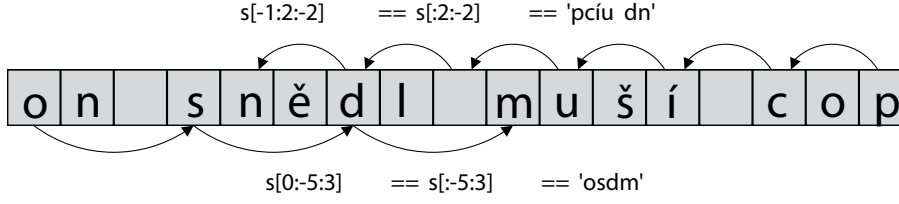
```
"he ate camel food"
"on snědl muší cop"
```



Obrázek 2.3: Krokování posloupnosti

Zde jsme použili výchozí počáteční a koncové indexy, takže `s[::-2]` začíná na posledním znaku a extrahuje každý druhý znak ve směru k začátku řetězce. Podobně `s[::3]` začíná na prvním znaku a extrahuje každý třetí znak směrem ke konci řetězce.

Je též možné zkombinovat řezací indexy s krokováním, což zachycuje obrázek 2.4.



Obrázek 2.4: Řezání a korkování posloupnosti

Krokování se nejčastěji nepoužívá s řetězci, ale s posloupnostmi jiného typu. Existuje však jedna situace, v níž se používá i pro řetězce:

```
>>> s, s[::-1]
('on snědl muší cop', 'poc íšum lděns no')
```

Krokování o -1 znamená, že se extrahuje každý znak od konce na začátek, čímž získáme obrácený řetězec.

Řetězcové operátory a metody

Iterovatelné operátory a funkce ➢ 138

Řetězce jsou neměnitelné posloupnosti, a proto veškerá funkčnost, která je dostupná u neměnitelných posloupností, je k dispozici také u řetězců. To zahrnuje testování příslušnosti, spojování operátorem `+`, připojování operátorem `+=` a replikace rozšířeným přiřazením `*=`. V této podčásti se kromě řady řetězcových metod podíváme také na všechny tyto operace v kontextu řetězců. Tabulka 2.7 uvádí přehled všech řetězcových metod kromě dvou poněkud specializovanějších (`str.maketrans()` a `str.translate()`), které si ve stručnosti probereme později.

Třída Sized ➢ 369

Řetězce jakožto posloupnosti znají pojem velikosti, a proto na nich můžeme volat metodu `len()`. Vrácená délka představuje počet znaků v řetězci (nula pro prázdný řetězec).

Viděli jsme, že operátor + je přetížen i pro řetězce, kde funguje jako spojování řetězců. Pro případy, kdy potřebujeme spojovat spoustu řetězců, nabízí metoda `str.join()` lepší řešení. Tato metoda přijímá jako svůj argument posloupnost (např. seznam nebo n-tici řetězců), jejíž prvky spojí do jediného řetězce, přičemž mezi ně umístí řetězec, na němž byla metoda zavolána:

```
>>> treatises = ["Aritmetika", "Kuželosečky", "Základy"]
>>> " ".join(treatises)
'Aritmetika Kuželosečky Základy'
>>> "-<>-".join(treatises)
'Aritmetika-<>-Kuželosečky-<>-Základy'
>>> "".join(treatises)
'AritmetikaKuželosečkyZáklady'
```

První příklad demonstrující spojování s jediným znakem (v tomto případě s mezerou) se používá pravděpodobně nejčastěji. Třetí příklad představuje díky prázdnému řetězci čisté spojení, což znamená, že se posloupnost řetězců spojí bez jakékoli výplně.

Metodu `str.joint()` můžeme společně s vestavěnou funkcí `reversed()` použít k obrácení řetězce (např. `"".join(reversed(s))`), ačkoliv stejného výsledky lze snadněji docílit krokováním (např. `s[::-1]`).

Operátor * funguje jako replikace řetězce:

```
>>> s = "=" * 5
>>> print(s)
=====
>>> s *= 10
>>> print(s)
==================================================
```

Jak je z tohoto příkladu patrné, můžeme použít také rozšířenou verzi přiřazení operátoru replikace.*

Při aplikaci na řetězce vrací operátor příslušnosti hodnotu `True` v případě, kdy je jeho levý argument podřetězcem pravého řetězcového argumentu nebo kdy jsou si jeho argumenty rovny.

Pro situace, kdy potřebujeme najít pozici jednoho řetězce uvnitř jiného, máme k dispozici dvě metody. První je metoda `str.index()`, která vrací indexovou pozici podřetězce nebo v případě neúspěchu vyvolá výjimku `ValueError`. Druhou je metoda `str.find()`, která vrací indexovou pozici podřetězce nebo -1 v případě neúspěchu. Obě metody přijímají jako první argument řetězec, který se má najít, a dále mohou přijímat několik volitelných argumentů. Druhým argumentem může být počáteční a třetím koncová pozice v prohledávaném řetězci.

* Řetězce podporují také operátor % pro formátování. Tento operátor je však zastaralý a slouží pouze ke snazšímu přechodu z Pythonu 2 na Python 3. V příkladech této knihy jej proto vůbec nebudeme používat.

Tabulka 2.7: Řetězcové metody

Syntaxe	Popis
s.capitalize()	Vrátí kopii řetězce s, jejíž první písmeno bude převedené na velké písmeno (viz též metoda `str.title()`).
s.center(délka, znak)	Vrátí kopii řetězce s vycentrovanou v řetězci zadané délky a vyplněnou mezerami nebo případně zadaným znakem, což je řetězec délky 1 (viz metody `str.ljust()`, `str.rjust()` a `str.format()`).
s.count(t, začátek, konec)	Vrátí počet výskytů řetězce t v řetězci s (nebo v řezu *začátek*:*konec* řetězce s).
s.encode(kódování, chyby)	Vrátí objekt typu `bytes` představující řetězec, který používá výchozí kódování nebo zadané kódování, přičemž chyby se ošetří podle volitelného argumentu *chyby*.
s.endswith(x, začátek, konec)	Vrátí hodnotu `True`, pokud s (nebo řez *začátek*:*konec* řetězce s) končí řetězcem x nebo libovolným řetězcem v n-tici x. V opačném případě vrátí hodnotu `False` (viz též metoda `str.startswith()`).
s.expandtabs(počet)	Vrátí kopii řetězce s, která má místo tabulátorů mezery. Každý tabulátor se nahradí 8 mezerami nebo zadaným počtem mezer.
s.find(t, začátek, konec)	Vrátí nejlevější pozici řetězce t v řetězci s (nebo v řezu *začátek*:*konec* řetězce s) nebo -1, pokud nebyl nalezen. K vyhledání nejpravějšího řetězce se používá metoda `str.rfind()` (viz též metoda `str.index()`).
s.format(...)	Vrátí kopii řetězce s naformátovanou podle zadaných argumentů. Této metodě a jejím argumentům se budeme věnovat v následující podčásti.
s.index(t, start, end)	Vrátí nejlevější pozici řetězce t v řetězci s (nebo v řezu *začátek*:*konec* řetězce s) nebo vyvolá výjimku `ValueError`, pokud nebyl nalezen. K vyhledání nejpravějšího řetězce se používá metoda `str.rindex()` (viz též metoda `str.find()`).
s.isalnum()	Vrátí hodnotu `True`, je-li řetězec s neprázdný a je-li každý znak v s alfanumerický.
s.isalpha()	Vrátí hodnotu `True`, je-li řetězec s neprázdný a je-li každý znak v s abecední.
s.isdecimal()	Vrátí hodnotu `True`, je-li řetězec s neprázdný a je-li každý znak v s číslicí o základu 10 ze znakové sady Unicode.
s.isdigit()	Vrátí hodnotu `True`, je-li řetězec s neprázdný a je-li každý znak v s číslicí ze znakové sady ASCII.
s.isidentifier()	Vrátí hodnotu `True`, je-li řetězec s neprázdný a představuje-li platný identifikátor.

typ bytes ➤ 286
Kódování znaků ➤ 95
str.format() ➤ 83
Identifikátory a klíčová slova ➤ 58

Syntaxe	Popis
s.islower()	Vrátí hodnotu `True`, má-li řetězec `s` alespoň jeden znak, který může mít podobu malého písmene, a všechny znaky, které mohou být malými písmeny, malými písmeny skutečně jsou (viz též metoda `str.isupper()`).
s.isnumeric()	Vrátí hodnotu `True`, je-li řetězec `s` neprázdný a je-li každý znak v řetězci `s` číselným znakem ze znakové sady Unicode ve formě číslice nebo zlomku.
s.isprintable()	Vrátí hodnotu `True`, je-li řetězec `s` prázdný nebo je-li každý znak v `s` považován za tisknutelný, což zahrnuje mezeru, ale nezahrnuje znak nového řádku.
s.isspace()	Vrátí hodnotu `True`, je-li řetězec `s` neprázdný a je-li každý znak v `s` bílým znakem.
s.istitle()	Vrátí hodnotu `True`, je-li řetězec `s` neprázdný a splňuje-li podmínky nadpisu (viz metoda `str.title()`).
s.isupper()	Vrátí hodnotu `True`, má-li řetězec `s` alespoň jeden znak, který může mít podobu velkého písmene, a všechny znaky, které mohou být velkými písmeny, velkými písmeny skutečně jsou (viz též metoda `str.islower()`).
s.join(seq)	Vrátí zřetězení všech prvků v posloupnosti *seq*, mezi které vloží řetězec `s` (který může být prázdný).
s.ljust(délka, znak)	Vrátí kopii řetězce `s` zarovnaného směrem doleva v řetězci zadané délky, který vyplní mezerami nebo volitelným argumentem znak (což je řetězec délky 1). Pro zarovnání doprava se používá metoda `str.rjust()` a pro zarovnání na střed metoda `str.center()` (viz též metoda `str.format()`).
s.lower()	Vrátí kopii řetězce `s` převedeného na malá písmena.
s.maketrans()	Doprovodná metoda metody `str.translate()`.
s.partition(t)	Vrátí n-tici tří řetězců: část řetězce `s` před nejlevějším výskytem řetězce `t`, řetězec `t` a část za řetězcem `t`.
s.replace(t, u, n)	Vrátí kopii řetězce `s`, v níž je každý výskyt (nebo maximálně `n` výskytů) řetězce `t` nahrazen řetězcem `u`.
s.split(t, n)	Vrátí seznam řetězců rozdělených nejvýše `n`-krát podle řetězce `t`. Není-li `n` zadáno, pak dělení probíhá v maximálním možné míře. Není-li zadáno `t`, pak se řetězec rozdělí podle mezer. Pro dělení zprava se používá metoda `str.rsplit()` – výsledek je odlišný pouze tehdy, je-li zadáno `n` a je-li jeho hodnota menší než maximální počet možných dělení.
s.splitlines(f)	Vrátí seznam řádků, který je výsledkem rozdělení řetězce `s` podle oddělovačů řádků. Nemá-li `f` hodnotu `True`, pak se oddělovače řádků odstraní.

Syntaxe	Popis
s.startswith(x, začátek, konec)	Vrátí hodnotu True, pokud řetězec s (nebo jeho řez *začátek*:*konec*) začíná řetězcem x nebo některým z řetězců v n-tici x.
s.strip(znaky)	Vrátí kopii řetězce s bez počátečního a koncového bílého znaku (nebo bez znaků v řetězci *znaky*).
s.swapcase()	Vrátí kopii řetězce s, která má místo malých písmen písmena velká a místo velkých písmen malá.
s.title()	Vrátí kopii řetězce s, jejíž první znak každého slova obsahuje velké písmeno a všechna ostatní písmena jsou malá (viz str.istitle()).
s.translate()	Doprovodná metoda metody str.maketrans() (podrobnosti viz následující text).
s.upper()	Vrátí kopii řetězce s převedeného na malá písmena (viz str.lower()).
s.zfill(délka)	Vrátí kopii řetězce s, která je na začátku vyplněna nulami, je-li kratší než zadaná délka.

Kterou vyhledávací metodu použijeme, je čistě záležitostí chuti a aktuálních okolností, i když pokud hledáme více indexových pozicí, pak s metodou str.index() budeme mít obvykle čistší kód, což demonstrují následující dvě ekvivalentní funkce:

```
def extract_from_tag(tag, line):
    opener = "<" + tag + ">"
    closer = "</" + tag + ">"
    try:
        i = line.index(opener)
        start = i + len(opener)
        j = line.index(closer, start)
        return line[start:j]
    except ValueError:
        return None
```

```
def extract_from_tag(tag, line):
    opener = "<" + tag + ">"
    closer = "</" + tag + ">"
    i = line.find(opener)
    if i != -1:
        start = i + len(opener)
        j = line.find(closer, start)
        if j != -1:
            return line[start:j]
    return None
```

Obě verze funkce `extract_from_tag()` se chovají naprosto stejně. Například volání `extract_from_tag("red", "jaká nádherná <red>růže</red>")` vrátí řetězec „růže“. Verze vlevo s ošetřováním výjimek odděluje kód, který dělá to, co chceme, od kódu, který ošetřuje chyby, kdežto verze vpravo promíchává vlastní zpracování s ošetřením chyb.

Všechny metody `str.count()`, `str.endswith()`, `str.find()`, `str.rfind()`, `str.index()`, `str.rindex()` a `str.startswith()` přijímají dva volitelné argumenty: počáteční a koncovou pozici. Jejich použití si ukážeme na následujících ekvivalentních příkazech (`s` je nějaký řetězec):

```
s.count("m", 6) == s[6:].count("m")
s.count("m", 5, -3) == s[5:-3].count("m")
```

Jak můžeme vidět, řetězcové metody, jež přijímají počáteční a koncové indexy, operují na řezu řetězce stanoveném těmito indexy.

Nyní se podíváme na další dva úryvky kódu, na nichž si objasníme chování metody `str.partition()` – i když v tomto příkladu použijeme metodu `str.rpartition()`:

```
result = s.rpartition("/")
```

```
i = s.rfind("/")
if i == -1:
    result = "", "", s
else:
    result = s[:i], s[i], s[i + 1:]
```

Úryvky kódu na levé i pravé straně nejsou úplně stejné, protože kód vpravo navíc vytváří novou proměnnou `i`. Všimněte si, že můžeme přímo přiřazovat n-tice a že v obou případech hledáme nejpravější výskyt lomítka (`/`). Pokud řetězec `s` obsahuje `"/usr/local/bin/firefox"`, pak oba úryvky vytvoří stejný výsledek: `('/usr/local/bin', '/', 'firefox')`.

Metodu `str.endswith()` (a `str.startswith()`) můžeme použít s jediným řetězcovým argumentem (např. `s.startswith("From:")`) nebo s n-ticí řetězců. Zde je příkaz, který pomocí metod `str.endswith()` a `str.lower()` otestuje, zda se jedná o soubor JPEG:

```
if filename.lower().endswith((".jpg", ".jpeg")):
    print(filename, " je obrázek JPEG")
```

Metody s názvem ve tvaru `is*()` (např. `isalpha()` nebo `isspace()`) vracejí hodnotu `True`, pokud řetězec, na kterém jsou zavolány, obsahuje nejméně jeden znak a pokud každý znak v tomto řetězci splňuje určitá kritéria:

```
>>> "917.5".isdigit(), "".isdigit(), "-2".isdigit(), "203".isdigit()
(False, False, False, True)
```

Metody `is*()` fungují na bázi klasifikace znaků ve znakové sadě Unicode, takže například volání metody `str.isdigit()` na řetězcích `"\N{circled digit two}03"` a `"②03"` vrátí v obou případech hodnotu `True`. Z tohoto důvodu nemůžeme předpokládat, že hodnota `True` vrácená metodou `isdigit()` automaticky znamená, že lze daný řetězec převést na celé číslo.

Když přijímáme řetězce z externích zdrojů (jiné programy, soubory, síťová připojení a zejména vstup od uživatelů), mohou tyto řetězce obsahovat nechtěné úvodní nebo koncové bílé místo. Úvodní bílé

místo odstraníme metodou `str.lstrip()`, koncové bílé místo metodou `str.rstrip()` a obě metodou `str.strip()`. Těmto metodám můžeme též předat jako argument řetězec, přičemž se odstraní každý výskyt všech znaků uvedených v zadaném řetězci:

```
>>> s = "\t neparkovat "
>>> s.lstrip(), s.rstrip(), s.strip()
(' neparkovat ', '\t neparkovat', 'neparkovat')
>>> "<[bez závorek]>".strip("[](){}<>")
'bez závorek'
```

příklad csv2-html.py ➤ 100

Pomoc metody `str.replace()` můžeme nahrazovat řetězce uvnitř řetězců. Tato metoda přijímá dva řetězcové argumenty a vrací kopii řetězce, na kterém je zavolána. V této kopii jsou všechny výskyty prvního řetězce nahrazeny druhým. Je-li druhý argument prázdný řetězec, pak se všechny výskyty prvního řetězce vymažou. Na ukázky použití metody `str.replace()` a některých dalších řetězcových metod se podíváme v části Příklady v příkladu `csv2html.py` (na konci této lekce).

Častým požadavkem je rozdělení řetězce na seznam řetězců. Představte si, že máme textový soubor s daty obsahující na každém řádku jeden záznam, jehož pole jsou oddělena hvězdičkou. V této situaci můžeme využít metodu `str.split()`, které předáme jako první argument řetězec, který se má rozdělit, a dále volitelný druhý argument představující maximální počet dělení. Pokud druhý argument neuvedeme, provede se maximální možný počet dělení. Zde je příklad:

```
>>> record = "Lev N. Tolstoj*28.8.1828*20.11.1910"
>>> fields = record.split("*")
>>> fields
['Lev N. Tolstoj', '28.8.1828', '20.11.1910']
```

Nyní můžeme znovu aplikovat metodu `str.split()` na data narození a úmrtí a vypočítat počet roků, kterých se Lev N. Tolstoj dožil (plus minus jeden rok):

```
>>> born = fields[1].split(".")
>>> born
['28', '8', '1828']
>>> died = fields[2].split(".")
>>> print("dožil se asi", int(died[2]) - int(born[2]), "let")
dožil se asi 82 let
```

Kromě funkce `int()`, kterou jsme použili pro převod let z řetězců na celá čísla, je uvedený úryvek kódu poměrně jednoduchý. K rokům se můžeme dostat také přímo ze seznamu `fields` (např. `year_born = int(fields[1].split(".")[2])`).

V tabulce 2.7 jsme u metod `str.maketrans()` a `str.translate()` neuvedli žádný popis. Metoda `str.maketrans()` se používá ke tvorbě překladové tabulky, která mapuje znaky na znaky. Tato metoda přijímá jeden, dva nebo tři argumenty. My si zde ale ukážeme pouze nejjednodušší volání (se dvěma argumenty), kdy je první argument řetězec obsahující znaky, z nichž se má překládat, a druhý argument řetězec obsahující znaky, na které se má překlad provést.

Oba řetězcové argumenty musejí mít stejnou délku. Metoda `str.translate()` přijímá jako argument překladovou tabulku a vrací kopii jejího řetězce se znaky přeloženými podle překladové tabulky. Zde je příklad přeložení řetězců obsahujících bengálské číslice na anglické číslice:

```
table = "".maketrans("\N{bengali digit zero}"
    "\N{bengali digit one}\N{bengali digit two}"
    "\N{bengali digit three}\N{bengali digit four}"
    "\N{bengali digit five}\N{bengali digit six}"
    "\N{bengali digit seven}\N{bengali digit eight}"
    "\N{bengali digit nine}", "0123456789")
print("20749".translate(table))                # vypíše: 20749
print("\N{bengali digit two}07\N{bengali digit four}"
    "\N{bengali digit nine}".translate(table))  # vypíše: 20749
```

Všimněte si, že jsme při volání metody `str.maketrans()` a při druhém volání funkce `print()` využili spojování řetězcových literálů Pythonu, díky kterému jsme mohli rozprostřít řetězce na více řádků bez nutnosti potlačení znaků nového řádku nebo explicitního zřetězení.

Metodu `str.maketrans()` jsme klidně zavolali na prázdném řetězci, protože vůbec nezáleží na tom, na jakém řetězci ji zavoláme. Tato metoda totiž jen zpracuje své argumenty a vrátí překladovou tabulku. Metody `str.maketrans()` a `str.translate()` lze použít také k vymazání znaků. Stačí metodě `str.maketrans()` předat jako třetí argument řetězec obsahující nechtěné znaky. Pokud potřebujeme sofistikovanější překlady znaků, pak bychom mohli vytvořit vlastní kodek. Více informací na toto téma naleznete v dokumentaci k modulu `codecs` (viz *http://docs.python.org/library/codecs.html*).

Python nabízí ještě několik dalších knihovních modulů, které poskytují funkce související s řetězci. Již jsme se stručně zmínili o modulu `unicodedata`, jehož použití si ukážeme v následující podčásti. Mezi další moduly, které stojí za povšimnutí, patří modul `difflib`, který lze použít k zobrazení rozdílů mezi soubory nebo řetězci, třída `io.StringIO` modulu `io`, která umožňuje čtení a zápis řetězců jako by se jednalo o soubory, a modul `textwrap`, který nabízí funkce pro zabalování a vyplňování řetězců. K dispozici je též modul `string`, který obsahuje několik užitečných konstant, jako jsou `ascii_letters` a `ascii_lowercase`. S příklady použití některých z těchto modulů se setkáme v lekci 5. Kromě toho poskytuje Python v modulu `re` skvělou podporu regulárních výrazů (viz Lekce 13).

Formátování řetězců metodou str.format()

Metoda `str.format()` nabízí velmi flexibilní a výkonný prostředek pro vytváření řetězců. Pro jednoduché případy se sice používá snadno, ale pro složité formátování se musíme naučit formátovací syntaxi, kterou tato metoda vyžaduje.

Metoda `str.format()` vrací nový řetězec, který má místo nahrazovacích polí vhodně naformátované argumenty:

```
>>> "Román '{0}' vyšel v roce {1}".format("Těžké časy", 1854)
"Román 'Těžké časy' vyšel v roce 1854"
```

Každé nahrazovací pole je označeno názvem pole ve složených závorkách. Má-li název pole podobu celého čísla, pak se jedná o index jednoho z argumentů předaných metodě `str.format()`. V tomto případě bylo tedy pole s názvem `0` nahrazeno prvním argumentem a pole s názvem `1` druhým argumentem.

Pokud potřebujeme do formátovacího řetězce umístit složené závorky, musíme je zdvojit. Zde je příklad:

```
>>> "{{{0}}} {1} ;-}}".format("Já jsem v závorkách", "A já ne")
'{Já jsem v závorkách} A já ne ;-}'
```

Pokud se pokusíme zřetězit řetězec a číslo, Python tiše vyvolá výjimku `TypeError`. Požadovaného výsledku však můžeme snadno dosáhnout pomocí metody `str.format()`:

```
>>> "{0}{1} Kč".format("Dlužná částka činí ", 200)
'Dlužná částka činí 200 Kč'
```

Pomocí metody `str.format()` můžeme také spojovat řetězce (i když metoda `str.join()` je pro tento účel vhodnější):

```
>>> x = "tři"
>>> s ="{0} {1} {2}"
>>> s = s.format("Moje", x, "tečky")
>>> s
'Moje tři tečky'
```

Zde jsme použili několik řetězcových proměnných, avšak ve většině příkladů v této části budeme v souvislosti s metodou `str.format()` používat řetězcové literály. Mějte ale na paměti, že jakýkoliv příklad používající řetězcový literál by mohl naprosto stejným způsobem používat řetězcovou proměnnou.

Nahrazovací pole může mít kteroukoli z následujících obecných syntaxí:

```
{název_pole}
{název_pole!převod}
{název_pole:specifikace_formátu}
{název_pole!převod:specifikace_formátu}
```

Důležité je, že nahrazovací pole mohou *obsahovat* další nahrazovací pole. Vnořená nahrazovací pole nemohou mít žádné formátování. Jejich účelem je podpora odvozených formátovacích specifikací. Podrobněji se tuto situaci podíváme v rámci výkladu specifikace formátu. Nyní prostudujeme postupně každou z částí nahrazovacího pole, přičemž začneme názvem pole.

Název pole

Názvem pole může být buď celé číslo odpovídající jednomu z argumentů metody `str.format()`, nebo jméno jednoho z klíčovaných argumentů metod. Klíčované argumenty budeme probírat v lekci 4, nejedná se ale o nic složitého, a proto se pro úplnost podíváme na několik příkladů:

```
>>> "{who} je tento rok {age}".format(who="Janě", age=88)
'Janě je tento rok 88'
>>> "{who} bylo minulý týden {0}".format(12, who="chlapci")
'chlapci bylo minulý týden 12'
```

První příklad používá dva klíčované argumenty `who` a `age` a druhý příklad používá jeden poziční (jediný druh, který jsme dosud používali) a jeden klíčovaný argument. Všimněte si, že v seznamu argumentů jsou klíčované argumenty vždy uvedeny až za pozičními argumenty. Ve formátovacím řetězci můžeme samozřejmě využít kterýkoli z argumentů v libovolném pořadí.

Názvy polí se mohou odkazovat na datové typy představující kolekce – například tedy na seznamy. V takových případech můžeme uvést index (ne řez!) označující konkrétní prvek:

```
>>> stock = ["papír", "obálky", "zápisníky", "pera", "sponky"]
>>> "Na skladě máme {0[1]} a {0[2]}".format(stock)
'Na skladě máme obálky a zápisníky'
```

Číslice `0` označuje poziční argument, takže `{0[1]}` je druhý prvek argumentu `stock` a `{0[2]}` je třetí prvek argumentu `stock`.

Později se seznámíme se slovníky jazyka Python. Ty totiž uchovávají prvky spojující klíče s hodnotami a vzhledem k tomu, že je lze použít s metodou `str.format()`, ukážeme si zde rychlý příklad. Nic si z toho nedělejte, pokud vám to zatím nebude dávat smysl. Vše se vyjasní v lekci 3.

Typ `dict` ➢ 128

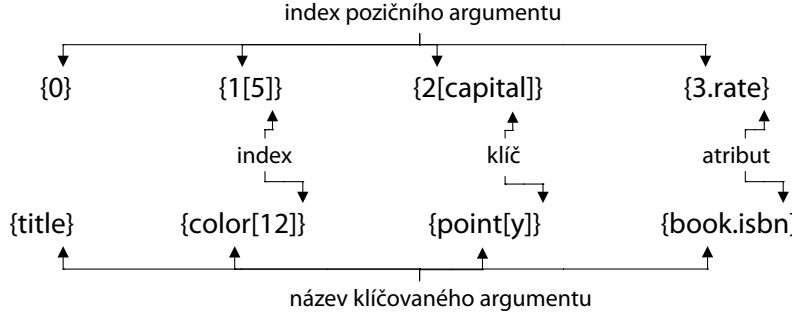
```
>>> d = dict(animal="slon", weight=12000)
>>> "Náš {0[animal]} váží {0[weight]} kg".format(d)
'Náš slon váží 12000 kg'
```

Stejně jako přistupujeme k prvkům seznamu a n-tice pomocí číselné pozice indexu, přistupujeme k prvkům slovníku pomocí klíče.

Můžeme také přistupovat k pojmenovaným atributům. Po importování modulů `math` a `sys` můžeme napsat následující kód:

```
>>> "math.pi=={0.pi} sys.maxunicode=={1.maxunicode}".format(math, sys)
'math.pi==3.14159265359 sys.maxunicode==65535'
```

Podtrženo sečteno: pomocí syntaxe pro název pole můžeme adresovat poziční i klíčované argumenty, které předáváme metodě `str.format()`. Jsou-li argumenty některého z datových typů představujících kolekce, jako jsou seznamy nebo slovníky, nebo pokud mají atributy, pak můžeme k části, kterou chceme, přistupovat pomocí závorkové (`[]`) nebo tečkové (`.`) notace. Tuto situaci zachycuje obrázek 2.5.



Obrázek 2.5: Ukázky specifikace formátu pro názvy polí

3.1

Od Pythonu 3.1 je možné názvy polí vynechat, přičemž Python za nás doplní čísla začínající od `0`:

```
>>> "{} {} {}".format("Python", "umí", "počítat")
'Python umí počítat'
```

Pokud používáte Python 3.0, pak musíte ve výše uvedeném příkladu použít formátovací řetězec ve tvaru `"{0} {1} {2}"`. Tato technika je výhodná pro formátování jednoho nebo dvou prvků, avšak postup, který si ukážeme vzápětí, je při práci s větším počtem prvků vhodnější a funguje skvěle i v Pythonu 3.0.

Ještě před uzavřením výkonu o názvech polí pro formátování řetězců je dobré zmínit poněkud odlišný způsob vkládání hodnot do formátovacího řetězce. K tomu je nutné sáhnout po pokročilé technice, kterou je vhodné si co nejdříve osvojit.

Rozbalení mapování ➤ 177

Lokální proměnné, které jsou aktuálně v oboru platnosti, jsou dostupné přes vestavěnou funkci `locals()`. Tato funkce vrací slovník, jehož klíče odpovídají názvům lokálních proměnných a jehož hodnoty se odkazují na hodnoty těchto proměnných. Nyní můžeme použít *rozbalení mapování* pro naládování tohoto slovníku do metody `str.format()`. K rozbalení mapování slouží operátor `**`, jehož aplikací na mapování (např. na slovník) vytvoříme seznam klíč-hodnota vhodný pro předání funkci:

```
>>> element = "Stříbro"
>>> number = 47
>>> "Prvek {number} je {element}".format(**locals())
'Prvek 47 je Stříbro'
```

Rozbalení parametrů ➤ 176

Tato syntaxe se může na první pohled jevit poněkud zvláštně, takže všichni programátoři v Perlu se budou cítit jako doma, ale ničeho se neobávejte – vše si vysvětlíme v lekci 4. Prozatím stačí vědět, že ve formátovacích řetězcích můžeme použít názvy proměnných a nechat Python, aby jejich hodnoty předal metodě `str.format()` prostým rozbalením slovníku vráceného funkcí `locals()` (nebo nějakým jiným slovníkem). Například výše uvedený příklad se slonem můžeme přepsat tak, aby měl hezčí formátovací řetězec s jednoduššími názvy polí:

```
>>> "Náš {animal} váží {weight} kg".format(**d)
'Náš slon váží 12000 kg'
```

Rozbalením slovníku do metody `str.format()` můžeme použít jeho klíče jako názvy polí. Díky tomu jsou formátovací řetězce mnohem srozumitelnější a také se snadněji udržují, protože nejsou závislé na pořadí argumentů. Je však třeba poznamenat, že pokud potřebujeme metodě `str.format()` předat více než jeden argument, pak můžeme rozbalení mapování použít pouze na poslední.

Převody

Desetinná čísla ➤ 69

Když jsme probírali čísla typu `decimal.Decimal`, řekli jsme si, že tato čísla se vypisují jedním ze dvou způsobů:

```
>>> decimal.Decimal("3.4084")
Decimal('3.4084')
>>> print(decimal.Decimal("3.4084"))
3.4084
```

První možnost, jak zobrazit číslo typu `decimal.Decimal`, je použít jeho reprezentační formu. Smyslem této formy je poskytovat řetězec, který by po interpretaci Pythonem opětovně vytvořil objekt, který reprezentuje. Programy napsané v Pythonu mohou vyhodnocovat úryvky kódu nebo celých programů napsaných v jazyku Python, takže tato schopnost může být v určitých situacích užitečná. Ne všechny objekty umí poskytovat reprodukční reprezentaci. V takovém případě vracejí řetězec uzavřený do lomených závorek. Například reprezentační forma modulu `sys` je řetězec `"<module 'sys' (built-in)>"`.

`eval()` ➤ 334

Druhou možnost pro zobrazení čísla typu `decimal.Decimal` nabízí jeho řetězcová forma. Tato forma je určena pro lidské čtenáře, a proto je zaměřena na zobrazení něčeho, co je srozumitelné pro člověka. Pokud datový typ řetězcovou formu nemá a je vyžadován řetězec, tak Python použije reprezentační formu.

Vestavěné datové typy jazyka Python znají metodu `str.format()`, takže při předání této metodě ve formě argumentu vrátí vhodný řetězec pro své zobrazení. Jak uvidíme v lekci 6, přidání podpory pro metodu `str.format()` do našich vlastních datových typů je také velice jednoduché. Kromě toho je možné běžné chování datového typu přepsat a přinutit jej poskytovat buď jeho řetězcovou, nebo jeho reprezentační formu. To lze provést tak, že k danému poli přidáme převodní specifikátor. V současnosti jsou k dispozici tři převodní specifikátory: `s` pro vynucení řetězcové formy, `r` pro vynucení reprezentační formy a specifikátor `a` pro vynucení reprezentační formy obsahující pouze znaky z kódování ASCII. Zde je příklad:

```
>>> "{0} {0!s} {0!r} {0!a}".format(decimal.Decimal("93.4"))
"93.4 93.4 Decimal('93.4') Decimal('93.4')"
```

V tomto případě vytvoří řetězcová forma čísla typu `decimal.Decimal` stejný řetězec, jako je ten, který poskytuje pro metodu `str.format()`, což se je docela obvyklé. Dále v tomto příkladu není žádný rozdíl mezi obyčejnou reprezentační formou a reprezentační formou v kódování ASCII, protože obě používají pouze znaky ze znakové sady ASCII.

Zde je další příklad, který se tentokrát týká řetězce, jenž obsahuje název filmu `"翻訳で失われる"` uchovávaný v proměnné `movie`. Pokud tento řetězec vypíšeme příkazem `"{0}".format(movie)`, pak se vypíše beze změny. Pokud však nechceme znaky mimo kódování ASCII, pak můžeme použít buď příkaz `ascii(movie)`, nebo `"{0!a}".format(movie)`, protože oba příkazy vytvoří řetězec `'\u7ffb\u8a33\u3067\u5931\u308f\u308c\u308b'`.

Dosud jsme viděli, jak umístit hodnoty proměnných do formátovaného řetězce a jak si vynutit použití řetězcové nebo reprezentační formy. Nyní jsme tedy připraveni podívat se formátování samotných hodnot.

Specifikace formátu

Často je výchozí formátování celých čísel, čísel s pohyblivou řádovou čárkou a řetězců naprosto dostačující. K snadnému procvičení jemné kontroly formátování můžeme využít specifikace formátu. Pro snazší osvojení podrobností se budeme věnovat formátování jednotlivých typů samostatně. Obecnou syntaxi vztahující se ke všem typům uvádí tabulka 2.8.

Tabulka 2.8: Obecný tvar specifikace formátu

Specifikátor	Popis
:	
výplň	Libovolný znak kromě znaku `}`.
zarovnání	< vlevo, > vpravo, ^ na střed, = vyplnění mezi znaménkem a číslicemi u čísel.
znaménko	+ vynucený zápis znaménka, - znaménko jen v případě potřeby, `" "` mezera nebo – dle aktuální potřeby.
#	Před celé číslo umístí prefix `0b`, `0o` nebo `0x`.
0	Číslo se vyplní nulami do požadované šířky.
šířka	Minimální šířka pole.
,	Čárka se používá pro seskupování*.
.přesnost	Maximální šířka pole pro řetězce. V případě čísel s pohyblivou řádovou čárkou udává počet desetinných míst.
typ	Pro typ `int`: `b`, `c`, `d`, `n`, `o`, `x`, `X`; pro typ `float`: `e`, `E`, `f`, `g`, `G`, `n`, `%`.

U řetězců můžeme nastavovat výplňový znak, zarovnání uvnitř pole a minimální a maximální šířku pole.

Specifikace formátu řetězce začíná dvojtečkou (`:`), za kterou následuje volitelná dvojice znaků: výplňový znak (což nesmí být znak `}`) a zarovnávací znak (`<` pro zarovnání vlevo, `^` pro zarovnání na střed, `>` pro zarovnání vpravo). Pak je na řadě volitelné celé číslo udávající minimální šířku, za nímž může následovat maximální šířka, která se zapisuje jako tečka a celé číslo.

Všimněte si, že pokud zadáme výplňový znak, musíme zadat též zarovnání. Části se znaménkem a typem jsme vynechali, protože na řetězce nemají žádný vliv. Dvojtečka bez volitelných prvků je sice neškodná, ale naprosto zbytečná.

Podívejme se na několik příkladů:

```
>>> s = "Žhnoucí meč pravdy"
>>> "{0}".format(s)       # výchozí formátování
'Žhnoucí meč pravdy'
>>> "{0:25}".format(s)    # minimální šířka 25
'Žhnoucí meč pravdy       '
>>> "{0:>25}".format(s)   # zarovnání vpravo, minimální šířka 25
'       Žhnoucí meč pravdy'
>>> "{0:^25}".format(s)   # zarovnání na střed, minimální šířka 25
'   Žhnoucí meč pravdy    '
>>> "{0:-^25}".format(s)  # výplň -, zarovnání na střed, minimální šířka 25
'---Žhnoucí meč pravdy----'
>>> "{0:.<25}".format(s)  # výplň ., zarovnání vlevo, minimální šířka 25
'Žhnoucí meč pravdy.......'
>>> "{0:.10}".format(s)   # maximální šířka 10
'Žhnoucí me'
```

* Seskupování bylo zavedeno v Pythonu 3.1.

V předposledním příkladu jsme museli uvést zarovnání vlevo (i když se jedná o výchozí zarovnání). Pokud bychom značku `<` vynechali, měli bychom `:.25`, což ale znamená, že maximální šířka pole je 25 znaků.

Jak jsme si řekli již dříve, specifikace formátu může obsahovat nahrazovací pole. Díky tomu můžeme formát definovat dynamicky. Zde jsou například dva způsoby nastavení maximální šířky řetězce pomocí proměnné `maxwidth`:

```
>>> maxwidth = 12
>>> "{0}".format(s[:maxwidth])
'Žhnoucí meč '
>>> "{0:.{1}}".format(s, maxwidth)
'Žhnoucí meč '
```

V prvním případě používáme standardní řezání řetězce, zatímco ve druhém máme vnitřní nahrazovací pole.

U celých čísel nám specifikace formátu umožňuje nastavovat výplňový znak, zarovnání uvnitř pole, znaménko, zda se má použít jiný oddělovač pro seskupení číslic (od Pythonu 3.1), minimální šířku pole a základ pro zobrazení čísla.

3.x

Specifikace formátu celého čísla začíná dvojtečkou, za níž můžeme uvést volitelnou dvojici znaků: výplňový znak (což nesmí být znak `}`) a zarovnávací znak (`<` pro zarovnání vlevo, `^` pro zarovnání na střed, `>` pro zarovnání vpravo a `=` pro výplň mezi znaménkem a číslem). Pak je na řadě volitelné celé číslo udávající minimální šířku, za nímž může následovat maximální šířka, která se zapisuje jako tečka a celé číslo. Dále zde máme volitelný znak znaménka: `+` způsobí vynucený výpis znaménka, `-` vypíše znaménko pouze pro záporná čísla a mezera vypíše mezeru pro kladná čísla a znaménko `-` pro záporná čísla. Pak následuje volitelná minimální šířka celého čísla, před níž může být umístěn znak `#` způsobující vypsání prefixu označujícího základ čísla (pro binární, osmičková a šestnáctková čísla) a znak `0` pro vyplnění znakem `0`. Počínaje Pythonem 3.1 můžeme uvést volitelnou čárku, která zapříčiní, že se cifry čísla seskupí do skupin po třech cifrách, při čemž se každá skupina oddělí čárkou. Pokud chceme mít výstup v jiném než desítkovém základu, musíme přidat znak pro typ: `b` pro binární, `o` pro osmičkový, `x` pro šestnáctkový s malými písmeny a `X` pro šestnáctkový s velkými písmeny (a pro úplnost lze uvést také `d` pro desítková celá čísla). Kromě toho jsou k dispozici ještě další dva znaky pro typ: `c`, což znamená, že se má použít znak ze znakové sady Unicode odpovídající celému číslu, a `n`, které vypíše číslo podle aktuálního národního prostředí (všimněte si, že při použití znaku `n` již nemá smysl používat čárku).

Vyplnění nulami můžeme provést dvěma různými způsoby:

```
>>> "{0:0=12}".format(8749203)  # vyplnění nulami, minimální šířka 12
'000008749203'
>>> "{0:0=12}".format(-8749203) # vyplnění nulami, minimální šířka 12
'-00008749203'
>>> "{0:012}".format(8749203)   # doplní nulami na minimální šířku 12
'000008749203'
>>> "{0:012}".format(-8749203)  # doplní nulami na minimální šířku 12
'-00008749203'
```

V prvních dvou příkladech je výplňový znak `0` a k vyplnění dochází mezi znaménkem a samotným číslem (`=`). Druhé dva příklady definují minimální šířku 12 a doplnění nulami.

Zde je několik příkladů zarovnání:

```
>>> "{0:*<15}".format(18340427)  # vyplnění *, zarovnání vlevo, min. šířka 15
'18340427*******'
>>> "{0:*>15}".format(18340427)  # vyplnění *, zarovnání vpravo, min. šířka 15
'*******18340427'
>>> "{0:*^15}".format(18340427)  # vyplnění *, zarovnání na střed, min. šířka 15
'***18340427****'
>>> "{0:*^15}".format(-18340427) # vyplnění *, zarovnání na střed, min. šířka 15
'***-18340427***'
Zde je několik příkladů, které ukazují výsledek aplikace znaménkových znaků:
>>> "[{0: }] [{1: }]".format(539802, -539802) # mezera nebo znak -
'[ 539802] [-539802]'
>>> "[{0:+}] [{1:+}]".format(539802, -539802) # vynucené vypsání znaménka
'[+539802] [-539802]'
>>> "[{0:-}] [{1:-}]".format(539802, -539802) # vypsání znaménka -, je-li třeba
'[539802] [-539802]'
```

A zde jsou dva příklady, které používají některé ze znaků pro typ:

```
>>> "{0:b} {0:o} {0:x} {0:X}".format(14613198)
'110111101111101011001110 67575316 deface DEFACE'
>>> "{0:#b} {0:#o} {0:#x} {0:#X}".format(14613198)
'0b110111101111101011001110 0o67575316 0xdeface 0XDEFACE'
```

V případě celých čísel nemůžeme stanovit maximální šířku pole. To je dáno tím, že by jinak bylo nutné osekat cifry, čímž by výpis čísla pozbyl smyslu.

3.1

Pokud používáme Python 3.1 a ve specifikaci formátu uvedeme čárku, pak celé číslo použije čárky pro seskupení:

```
>>> "{0:,} {0:*>13,}".format(int(2.39432185e6))
'2,394,321 ****2,394,321'
```

Na obě pole jsme aplikovali seskupení, přičemž druhé pole je doplněno hvězdičkami, zarovnáno vpravo a má nastavenou minimální šířku 13 znaků. Jedná se o obvyklý způsob zápisu v řadě vědeckých a finančních programů, který však nebere v potaz aktuální národní prostředí. Například v České republice se tisíce obvykle oddělují mezerou a jako oddělovač desetinné části se používá čárka.

Posledním formátovacím znakem dostupným pro celá čísla (a také pro čísla s pohyblivou řádovou čárkou) je znak `n`. Ten má při zadání celého čísla resp. čísla, s pohyblivou řádovou čárkou, stejný účinek jako znak `d`, resp. `g`. Jeho zvláštností je ovšem to, že respektuje národní prostředí, takže ve výstupu použije oddělovač desetinné části a seskupení pro aktuální národní prostředí. Výchozí národní prostředí označované jako `C` definuje pro oddělovač desetinné části tečku a pro oddělovač seskupení

prázdný řetězec. Národní prostředí uživatele můžeme respektovat tak, že na začátku svých programů napíšeme následující dva řádky kódu:*

```
import locale
locale.setlocale(locale.LC_ALL, "")
```

Předáním prázdného řetězce jako národního prostředí říkáme Pythonu, aby se automaticky pokusil zjistit národní prostředí uživatele (např. prozkoumáním proměnné prostředí `LANG`) s tím, že se v případě neúspěchu vrátí k národnímu prostředí `C`. Zde je několik příkladů, které ukazují účinky různých národních prostředí na formátování celých čísel a čísel s pohyblivou řádovou čárkou:

```
x, y = (1234567890, 1234.56)
locale.setlocale(locale.LC_ALL, "C")
c = "{0:n} {1:n}".format(x, y)    # c == '1234567890 1234.56'
locale.setlocale(locale.LC_ALL, "")
cz = "{0:n} {1:n}".format(x, y)   # cz == '1 234 567 890 1 234,56'
```

I když volba `n` je pro celá čísla velice užitečná, v případě čísel s pohyblivou řádovou čárkou jsou její možnosti omezené, protože jakmile jsou tato čísla příliš velká, zobrazují se pomocí exponenciální formy.

U čísel s pohyblivou řádovou čárkou nám specifikace formátu poskytuje kontrolu nad výplňovým znakem, zarovnáním uvnitř pole, znaménkem, dále nad tím, zda se má použít oddělovač skupin cifer bez ohledu na národní prostřední (od Pythonu 3.1), nad minimální šířkou pole, počtem desetinných míst a nad tím, zda se má číslo prezentovat ve standardní či exponenciální formě nebo jako procentní podíl.

Specifikace formátu pro čísla s pohyblivou řádovou čárkou je stejná jako pro celá čísla, ovšem až na dvě odlišnosti na konci. Za volitelnou minimální šířkou (od Pythonu 3.1 za volitelnou čárkou pro seskupení) můžeme zapsáním tečky a celého čísla stanovit počet číslic za oddělovačem desetinných míst. Na konec můžeme také přidat znak pro typ: `e` pro exponenciální formu s malým písmenem „e“, `E` pro exponenciální formu s velkým písmenem „E“, `f` pro standardní formu čísla s pohyblivou řádovou čárkou, `g` pro „obecnou“ formu (ta je stejná jako v případě `f`, pokud ovšem číslo není příliš velké, což způsobí, že se použije `e`) a `G`, což je stejné jako `g`, ale používá buď `f`, nebo `E`. Kromě toho máme k dispozici také znak `%`, který způsobí to, že se číslo vynásobí 100 a výsledek se zobrazí ve formátu f s připojeným symbolem `%`.

Zde je několik příkladů, které ukazují exponenciální a standardní formu:

```
>>> amount = (10 ** 3) * math.pi
>>> "[{0:12.2e}] [{0:12.2f}]".format(amount)
'[    3.14e+03] [     3141.59]'
>>> "[{0:*>12.2e}] [{0:*>12.2f}]".format(amount)
'[****3.14e+03] [*****3141.59]'
>>> "[{0:*>+12.2e}] [{0:*>+12.2f}]".format(amount)
'[***+3.14e+03] [****+3141.59]'
```

* Ve vícevláknových programech je nejvhodnější volat metodu `locale.setlocale()` pouze jednou při spouštění programu, než se spustí jakákoli další vlákna, neboť tato metoda obvykle není ve vícevláknovém prostředí bezpečná.

V prvním příkladu máme minimální šířku 12 znaků a 2 číslice na desetinných místech. Druhý příklad staví na prvním a přidává výplňový znak `*`. Pokud používáme výplňový znak, musíme uvést také zarovnávací znak. Proto jsme stanovili zarovnání vpravo (i když se jedná o výchozí zarovnání pro čísla). Třetí příklad staví na předchozích dvou a přidává znak `+`, kterým si vynutíme výpis znaménka.

3.1 V Pythonu 3.0 nepracuje metoda `str.format()` s čísly typu `decimal.Decimal` jako s čísly, ale jako s řetězci. Kvůli tomu je docela těžké vytvořit pěkně naformátovaný výstup. Počínaje Pythonem 3.1 lze čísla typu `decimal.Decimal` formátovat jako typ `float` včetně podpory pro volbu čárka (`,`) vytvářející čárkou oddělené skupiny. Zde je příklad (název pole jsme vynechali, protože jej v Pythonu 3.1 nepotřebujeme):

```
>>> "{:,.6f}".format(decimal.Decimal("1234567890.1234567890"))
'1,234,567,890.123457'
```

Pokud formátovací znak `f` vynecháme (nebo použijeme formátovací znak `g`), pak se číslo naformátuje jako `'1.23457E+9'`.

Python 3.0 neposkytuje žádnou přímou podporu pro formátování komplexních čísel – ta byla přidána až v Pythonu 3.1. To však můžeme snadno vyřešit naformátováním reálné a imaginární části jako samostatná čísla s pohyblivou řádovou čárkou:

```
>>> "{0.real:.3f}{0.imag:+.3f}j".format(4.75917+1.2042j)
'4.759+1.204j'
>>> "{0.real:.3f}{0.imag:+.3f}j".format(4.75917-1.2042j)
'4.759-1.204j'
```

Ke každému atributu komplexního čísla přistupujeme samostatně a formátujeme jej jako číslo s pohyblivou řádovou čárkou, v tomto případě se třemi číslicemi na desetinných místech. Vynutili jsme si také vypisování znaménka pro imaginární část, přičemž znak „j“ musíme přidat sami.

3.1 Python 3.1 podporuje formátování komplexních čísel pomocí stejné syntaxe, která se používá pro čísla typu `float`:

```
>>> "{:,.4f}".format(3.59284e6-8.984327843e6j)
'3,592,840.0000-8,984,327.8430j'
```

Jediná malinká nevýhoda tohoto přístupu tkví v tom, že stejné formátování se aplikuje na reálnou i imaginární část. Pokud potřebujeme každou z těchto částí naformátovat odlišným způsobem, můžeme vždy sáhnout po technice pro Python 3.0, v níž přistupujeme k atributům komplexního čísla samostatně.

Příklad: print_unicode.py

V předchozích podčástech jsme si podrobně rozebrali specifikaci formátu metody `str.format()` a viděli jsme řadu úryvků kódu, na kterých jsme si ukázali nejrůznější aspekty formátování řetězců. Zde se podíváme na malý, přesto však užitečný příklad, který využívá metodu `str.format()` tak, že si můžeme prohlédnout specifikace formátu v realistickém kontextu. V tomto příkladu používáme

také některé z řetězcových metod, s nimiž jsme se seznámili v předchozí části, a dále si představíme funkci z modulu `unicodedata`*.

Tento program má pouze 25 řádků spustitelného kódu. Importuje dva moduly, `sys` a `unicodedata`, a definuje jednu vlastní funkci `print_unicode_table()`. Začneme pohledem na ukázkový běh programu, abychom si ukázali, co program dělá, a poté se podíváme na kód na konci programu, kde začíná vlastní zpracování, a nakonec se zaměříme na námi definovanou funkci.

```
print_unicode.py spoked
desítk.  hexa.  znak                    název
-------  -----  ----  ---------------------------------------
  10018   2722   ✢   Four Teardrop-Spoked Asterisk
  10019   2723   ✣   Four Balloon-Spoked Asterisk
  10020   2724   ✤   Heavy Four Balloon-Spoked Asterisk
  10021   2725   ✥   Four Club-Spoked Asterisk
  10035   2733   ✳   Eight Spoked Asterisk
  10043   273B   ✻   Teardrop-Spoked Asterisk
  10044   273C   ✼   Open Centre Teardrop-Spoked Asterisk
  10045   273D   ✽   Heavy Teardrop-Spoked Asterisk
  10051   2743   ❃   Heavy Teardrop-Spoked Pinwheel Asterisk
  10057   2749   ❉   Balloon-Spoked Asterisk
  10058   274A   ❊   Eight Teardrop-Spoked Propeller Asterisk
  10059   274B   ❋   Heavy Eight Teardrop-Spoked Propeller Asterisk
```

Pokud program spustíme bez argumentů, obdržíme tabulku všech znaků znakové sady Unicode, počínaje znakem mezery až po znak s nejvyšší dostupným kódovým bodem. Zadáme-li nějaký argument, jako ve výše uvedeném případě, vypíšou se pouze ty řádky v tabulce, v nichž název znaku znakové sady Unicode převedený na malá písmena obsahuje zadaný argument.

```
word = None
if len(sys.argv) > 1:
    if sys.argv[1] in ("-h", "--help"):
        print("použití: {0} [řetězec]".format(sys.argv[0]))
        word = 0
    else:
        word = sys.argv[1].lower()
if word != 0:
    print_unicode_table(word)
```

Po importování modulů a vytvoření funkce `print_unicode_table()` se provádění programu dostane do výše uvedeného místa. Začneme předpokladem, že uživatel na příkazovém řádku nezadal slovo, které se má vyhledat. Pokud je na příkazovém řádku zadán argument `-h` nebo `--help`, vypíše-

* V tomto programu předpokládáme, že konzola používá kódování UTF-8 znakové sady Unicode. Naneštěstí konzola systému Windows má chabou podporu pro kódování UTF-8. Tento problém obcházíme zvláštní verzí příkladu s názvem `print_unicode_uni.py`, která svůj výstup zapisuje do souboru, který lze poté otevřít v nějakém editoru, který umí pracovat s kódováním UTF-8, jako je kupříkladu editor IDLE.

Lekce 7 (Práce se soubory) ➤ 280

me informace o použití programu a nastavíme `word` na hodnotu `0`, což signalizuje, že jsme skončili. V opačném případě nastavíme `word` na kopii argumentu zadaného uživatelem, který jsme převedli na malá písmena. Pokud proměnná `word` nemá hodnotu `0`, vypíšeme tabulku.

Při vypisování informací o použití programu používáme specifikaci formátu, která obsahuje jen formátovací název, kterým je v tomto případě číslo pozice argumentu. Tento řádek bychom mohli zapsat také takto:

```
print("použití: {0[0]} [řetězec]".format(sys.argv))
```

V tomto případě je první `0` indexem argumentu, který chceme použít, a `[0]` indexem *uvnitř* tohoto argumentu, což může být, protože `sys.argv` je seznam.

```
def print_unicode_table(word):
    print("desítk.  hexa.  znak  {0:^40}".format("název"))
    print("-------  -----  ----  {0:-<40}".format(""))

    code = ord(" ")
    end = min(0xD800, sys.maxunicode)

    while code < end:
        c = chr(code)
        name = unicodedata.name(c, "*** neznámé ***")
        if word is None or word in name.lower():
            print("{0:7} {0:5X} {0:^3c} {1}".format(code, name.title()))
        code += 1
```

Kvůli lepší čitelnosti jsme použili několik prázdných řádků. První dva řádky těla funkce vypíšou záhlaví tabulky. První volání metody `str.format()` vypíše text „název“ zarovnaný na střed pole širokého 40 znaků, zatímco druhé vypíše s použitím výplňového znaku „-“ prázdný řetězec v poli širokém 40 znaků. (Pokud stanovíme výplňový znak, musíme uvést zarovnání.) Druhý řádek bychom mohli zapsat také tímto způsobem:

```
print("-------  -----  ---- {0}".format("-" * 40))
```

Zde jsme pomocí operátoru řetězcové replikace (`*`) vytvořili vhodný řetězec, který jsme prostě vložili do formátovacího řetězce. Třetí možností by bylo jednoduše zapsat 40 pomlček a použít literálový řetězec.

Kódové body znakové sady Unicode máme uloženy v proměnné code, kterou inicializujeme kódovým bodem pro mezeru (`0x20`). Koncovou proměnnou nastavujeme na nejvyšší dostupný kódový bod znakové sady Unicode, který se liší podle toho, zda byl Python zkompilován tak, aby používal formát UCS-2, nebo UCS-4.

Uvnitř cyklu `while` vezmeme pomocí funkce `chr()` znak ze znakové sady Unicode, který odpovídá kódovému bodu. Funkce `unicodedata.name()` vrací název zadaného znaku ze znakové sady Unicode. Jejím volitelným druhým argumentem je název, který se má použít, pokud pro daný znak není definován žádný název.

Pokud uživatel nezadal nějaké slovo (`word` je `None`) nebo pokud jej zadal a nachází se v kopii názvu znaku Unicode převedeného na malá písmena, pak daný řádek vypíšeme.

I když proměnnou `code` předáváme metodě `str.format()` pouze jednou, používá se při formátování řetězce hned třikrát. Poprvé pro vypsání kódu ve formě celého čísla v poli širokém 7 znaků (výplňový znak je ve výchozím stavu mezera, takže jej nemusíme uvádět), podruhé pro vypsání kódu jako šestnáctkového čísla s velkými písmeny v poli širokém 5 znaků a potřetí pro vypsání znaku Unicode, který odpovídá aktuálnímu kódu (pro tento účel používám formátovací specifikátor „c“) a který zarovnáme na střed do pole s minimální šířkou 3 znaky. Všimněte si, že v první specifikaci formátu nemusíme zadávat typ „d“, což je dáno tím, že se jedná o výchozí typ pro celočíselné argumenty. Druhý argument je název znaku znakové sady Unicode vypsaný způsobem, kdy je první písmeno každého slova velké a všechna ostatní písmena jsou malá.

Nyní již známe všestranné použití metody `str.format()`, a proto ji budeme ve zbývající části této knihy bohatě využívat.

Kódování znaků

Počítače dokážou v podstatě uchovávat pouze bajty, což jsou 8bitové hodnoty, které mají bez znaménka rozsah od `0x00` do `0xFF`. Každý znak tak musí být nějakým způsobem reprezentován pomocí bajtů. V dřevních dobách počítačů vymysleli průkopníci kódovací schémata, která přiřazovala určitý znak určitému bajtu. Například v kódování ASCII je znak A reprezentován hodnotou `0x41`, B hodnotou `0x42` a tak dále. Ve Spojených státech a v západní Evropě se často používalo kódování Latin-1. Jeho znaky z rozsahu 0x20-0x7E jsou stejné jako odpovídající znaky v 7bitovém kódování ASCII, přičemž znaky z rozsahu 0xA0-0xFF se používaly pro písmena s diakritikou a další symboly nezbytné pro ty, kteří používali pro písmena latinku, ale ne anglickou. V průběhu let byla vymyšlena řada dalších kódování, z nichž mnohá se používají dodnes. Nicméně vývoj se pro spoustu z nich zastavil ve prospěch kódování Unicode.

Existence všech těchto odlišných kódování se ukázala jako velice nevhodná, zejména pak při psaní internacionalizovaného softwaru. Jedno řešení, které bylo již téměř celosvětově přijato, je kódování Unicode. Toto kódování přiřazuje každý znak celému číslu (označovanému řečí Unicode jako *kódový bod*), podobně jako předchozí kódování. Avšak kódování Unicode není omezeno na používání jednoho bajtu pro jeden znak, a proto je schopné reprezentovat každý znak v každém jazyku v jediném kódování. To znamená, že na rozdíl od ostatních kódování není kódování Unicode omezeno na jeden jazyk, ale dokáže pracovat se znaky z více jazyků.

Jak se ale kódování Unicode ukládá? V současnosti je definováno více než 100 000 znaků znakové sady Unicode, takže i při použití čísel se znaménky je 32bitové celé číslo více než dostačující pro uložení libovolného kódového bodu v kódování Unicode. Nejjednodušší způsob uložení znaků Unicode je posloupnost 32bitových celých čísel, kde jedno celé číslo představuje jeden znak. To zní docela příhodně, neboť tím bychom dostali přímé mapování znaků na 32bitová čísla, což by vedlo k tomu, že by indexování určitého znaku bylo velice rychlé. Nicméně v praxi nejsou věci tak jednoduché, poněvadž některé znaky Unicode lze reprezentovat jedním či dvěma kódovými body. Například „é“ lze reprezentovat jediným kódovým bodem 0xE9 nebo dvěma kódovými body: 0×65 a 0×301 („e“ a „čárka“).

V současnosti se znaková sada Unicode obvykle ukládá na disk a do paměti pomocí kódování UTF-8, UTF-16 nebo UTF-32. První z nich, UTF-8, je zpětně kompatibilní se 7bitovým kódováním ASCII, protože jeho prvních 128 kódových bodů je reprezentováno jednobajtovými hodnotami, které jsou stejné jako hodnoty 7bitových znaků v kódování ASCII. K reprezentaci všech ostatních znaků Unicode používá kódování UTF-8 pro jeden znak dva, tři nebo více bajtů. Díky tomu je UTF-8 velice kompaktní pro reprezentaci textu, který je celý nebo z velké většiny tvořen znaky anglické abecedy. Knihovna Gtk (používaná mimo jiné okénkovým systémem GNOME) používá kódování UTF-8 a vypadá to, že toto kódování se stává de facto standardním formátem pro ukládání textu ve znakové sadě Unicode do souborů. Kódování UTF-8 je kupříkladu výchozím formátem pro kód jazyka XML a toto kódování dále používá řada soudobých webových stránek.

Formát UCS-2 (který je v moderní formě stejný jako UTF-16) používá i spousta dalších softwarových produktů, jako je kupříkladu Java. Tato reprezentace používá pro jeden znak dva nebo čtyři bajty, přičemž nejčastěji používané znaky jsou reprezentovány dvěma bajty. Reprezentace UTF-32 (označovaná též jako UCS-4) používá pro jeden znak čtyři bajty. Použití kódování UTF-16 nebo UTF-32 pro ukládání do souborů nebo posílání přes síťové připojení však skrývá jisté úskalí. Pokud se totiž data posílají jako celá čísla, tak záleží na uspořádání bajtů (buď od nejvyšší nebo od nejnižší adresy). Jedním z řešení tohoto problému je umístit před data označení pořadí bajtů, aby se příjemce dokázal tomuto pořadí přizpůsobit. Tento problém ovšem kódování UTF-8 nepostihuje, což je další důvod, proč je tak populární.

Python reprezentuje znakovou sadu Unicode pomocí formátu UCS-2 (UTF-16) nebo UCS-4 (UTF-32). V případě formátu UCS-2 používá Python malinko zjednodušenou verzi, která pro jeden znak používá vždy dva bajty, a proto dokáže reprezentovat kódové body po hodnotu 0xFFFF. Při použití formátu UCS-4 dokáže Python reprezentovat všechny kódové body znakové sady Unicode. Maximální kódový bod je uložen v atributu určeném pouze pro čtení s názvem `sys.maxunicode`. Je-li jeho hodnota 65535, pak byl Python zkompilován tak, aby používal formát UCS-2. Je-li větší, můžeme říci, že používá formát UCS-4.

Metoda `str.encode()` vrací posloupnost bajtů (ve skutečnosti jde o objekt typu `bytes`, jemuž se budeme věnovat v lekci 7) zakódovanou podle zadaného argumentu. Pomocí této metody můžeme získat určitý vhled do odlišností mezi jednotlivými kódováními a přesvědčit se, proč může vést nesprávně zvolené kódování k chybám:

```
>>> artist = "Tage Ĺsén"
>>> artist.encode("Latin1")
b'Tage \xc5s\xe9n'
>>> artist.encode("CP850")
b'Tage \x8fs\x82n'
>>> artist.encode("utf8")
b'Tage \xc3\x85s\xc3\xa9n'
>>> artist.encode("utf16")
b'\xff\xfeT\x00a\x00g\x00e\x00 \x00\xc5\x00s\x00\xe9\x00n\x00'
```

Písmeno „b“ před otevíracími uvozovkami říká, že literál není řetězec, ale že je typu `bytes`. Standardně můžeme při vytváření literálů typu `bytes` používat směsici tisknutelných znaků ASCII a bajtů ve formě čísel v šestnáctkové soustavě, před nimiž jsou umístěny znaky „\x“.

Jméno „Tage Åsén“ nemůžeme zakódovat pomocí znakové sady ASCII, protože ta neobsahuje znak „*Å*“ ani žádný jiný znak s diakritikou, takže by došlo k vyvolání výjimky `UnicodeEncodeError`. Kódování Latin-1 (označované též jako ISO-8859-1) je 8bitové kódování, jež obsahuje všechny nezbytné znaky pro uvedené jméno. Na druhou stranu jméno „Pavöl Šemík“ by již takové štěstí nemělo, protože znak „*ö*“ není součástí kódování Latin-1, a proto by jej nešlo úspěšně zakódovat. Obě jména lze samozřejmě zakódovat pomocí znakové sady Unicode. Všimněte si však, že u kódování UTF-16 představují první dva bajty označení pořadí bajtů. Tyto bajty pak použije dekódovací funkce ke zjištění, zda jsou data ve formátu Big nebo Little Endian.

O metodě `str.encode()` si musíme říci ještě několik dalších věcí. U prvního argumentu (název kódování) se rozlišuje velikost písmen, přičemž pomlčky a podtržítka jsou považovány za totéž, takže například „us-ascii“ a „US_ASCII“ jsou stejné názvy. Dále je k dispozici řada aliasů – kupříkladu „latin“, „latin1“, „latin_1“, „ISO-8859-1“, „CP819“ a některé další představují „Latin-1“. Tato metoda může dále přijímat volitelný druhý argument, který metodě říká, jak má zpracovat chyby. Můžeme tak například zakódovat libovolný řetězec do kódování ASCII, pokud jako druhý argument zadáme „ignore“ nebo „replace“, což ovšem znamená možnost ztráty dat. Nechceme-li o data přijít, můžeme zadat „backslashreplace“, což způsobí nahrazení všech znaků nespadajících do znakové sady ASCII příslušnými bajtovými kódy s prefixem „\x“, „\u“ nebo „\U“. Například volání `artist.encode("ascii", "ignore")` vrátí `b'Tage sn'` a volání `artist.encode("ascii", "replace")` vrátí `b'Tage ?s?n'`, zatímco volání `artist.encode("ascii", "backslashreplace")` vrátí `b'Tage \xc5s\xe9n'`. (Řetězec v kódování ASCII můžeme získat také pomocí výrazu `"{0!a}".format(artist)`, který se vyhodnotí na `'Tage \xc5s\xe9n'`.)

Opakem metody `str.encode()` je metoda `bytes.decode()` (a také metoda `bytearray.decode()`), která vrací řetězec s bajty dekódovanými pomocí zadaného kódování:

```
>>> print(b"Tage \xc3\x85s\xc3\xa9n".decode("utf8"))
Tage Åsén
>>> print(b"Tage \xc5s\xe9n".decode("latin1"))
Tage Åsén
```

Rozdíly mezi 8bitovými kódováními Latin-1, CP850 (kódování pro IBM PC) a UTF-8 dávají jasně najevo, že snaha o hádání kódování s největší pravděpodobností není úspěšnou strategií. Kódování UTF-8 se v oblasti holých textových souborů naštěstí stává de facto standardem, takže pozdější generace již možná ani nebudou vědět, že existují také nějaká jiná kódování.

Soubory `.py` s kódem jazyka Python používají kódování UTF-8, takže Python vždy ví, které kódování má u řetězcových literálů použít. To znamená, že do svých řetězců můžeme zapsat libovolné znaky ze znakové sady Unicode, tedy za předpokladu, že je náš editor podporuje.*

Když Python čte data z externích zdrojů, jako jsou sokety, pak nemůže vědět, které kódování používají, a proto vrátí bajty, které pak můžeme adekvátním způsobem dekódovat. U textových souborů používá Python jemnější způsob spočívající v použití místního kódování, není-li kódování stanoveno explicitně.

* Je možné používat i jiná kódování. Přečtěte si tutoriál k jazyku Python s názvem „Source Code Encoding“.

Naštěstí některé souborové formáty svá kódování uvádějí. Například můžeme předpokládat, že soubor XML používá kódování UTF-8, pokud direktiva `<?xml?>` explicitně nespecifikuje odlišné kódování. Při čtení kódu jazyka XML můžeme tedy extrahovat řekněme prvních 1000 bajtů, podívat se na specifikaci kódování, a pokud ji nalezneme, tak dekódovat soubor pomocí uvedeného kódování; v opačném případě přejdeme zpět k dekódování s použitím UTF-8. Tento postup by měl fungovat u libovolného textového souboru s kódem jazyka XML nebo s holým textem, který používá kterékoli z jednobajtových kódování podporovaných Pythonem, kromě kódování založeného na kódu EBCDIC (CP424, CP500) a několika dalších (CP037, CP864, CP865, CP1026, CP1140, HZ, SHIFT-JIS-2004, SHIFT-JISX0213). Tento postup však nebude fungovat u vícebajtových kódování (jako jsou např. UTF-16 a UTF-32). Na webu Python Package Index (viz *http://pypi.python.org/pypi*) jsou k dispozici nejméně dva balíčky Pythonu pro automatickou detekci kódování souboru.

Příklady

V této části využijeme toho, čemu jsme se věnovali v této a předchozí lekci, abychom si ukázali dva malé, ale zato ucelené prográmky, na nichž si procvičíme vše, co jsme se dosud naučili. První program je malinko matematický, se svými 35 řádky je ale docela krátký. Druhý se věnuje zpracování textu a se svými sedmi funkcemi na přibližně 80 řádcích kódu je tak o něco „výživnější".

Program quadratic.py

Kvadratické rovnice jsou rovnice popisující parabolu ve tvaru $ax^2 + bx + c = 0$, kde $a \neq 0$. Kořeny takovéto rovnice se odvozují ze vzorce $x = (\frac{-b \pm \sqrt{b^2-4ac}}{2a})$. Část $b^2 - 4ac$ se označuje jako *diskriminant*. Je-li diskriminant kladný, pak existují dva reálné kořeny, je-li nulový, existuje jeden reálný kořen, a je-li záporný, pak existují dva komplexní kořeny. Napíšeme program, který přijme koeficienty *a, b* a *c* od uživatele (přičemž *b* a *c* mohou být 0) a poté vypočítá a vypíše kořen nebo kořeny.*

Nejdříve se podíváme na ukázkový běh programu a poté si projede jeho kód.

```
quadratic.py
ax² + bx + c = 0
zadejte a: 2.5
zadejte b: 0
zadejte c: -7.25
2.5x² + 0.0x + -7.25 = 0 → x = 1.70293863659 nebo x = -1.70293863659
```

S koeficienty 1,5, -3 a 6 vypadá výstup takto (některá čísla jsou ořezána):

```
1.5x² + -3.0x + 6.0 = 0 → x = (1+1.7320508j) nebo x = (1-1.7320508j)
```

Výstup není tak pěkný, jak by měl být. Kupříkladu místo `+ -3.0x` bychom raději měli mít `- 3.0x` a činitelé s koeficienty s hodnotou 0 by se neměli vůbec zobrazovat. Tyto problémy budete moci napravit ve cvičeních.

---

* Konzola Windows má jen velmi slabou podporu kódování pro UTF-8, a proto se mohou u několika znaků (² a →) používaných v programu quadratic.py vyskytnout problémy. Z tohoto důvodu je k dispozici též program `quadratic_uni.py`, který zobrazí správné symboly na systémech Linux a Mac OS X a na systémech Windows zobrazí alternativní znaky (^2 and →).

Přejděme nyní ke kódu, který začíná třemi příkazy `import`:

```
import cmath
import math
import sys
```

Potřebujeme matematické knihovny pro práci s typem `float` i `complex`, protože kořenové funkce pro reálná a komplexní čísla jsou odlišné. Dále potřebujeme modul `sys`, jenž obsahuje konstantu `sys.float_info.epsilon`, kterou potřebujeme pro porovnávání čísel s pohyblivou řádovou čárkou s nulou.

Kromě toho potřebujeme funkci, která od uživatele získá číslo s pohyblivou řádovou čárkou:

```
def get_float(msg, allow_zero):
    x = None
    while x is None:
        try:
            x = float(input(msg))
            if not allow_zero and abs(x) < sys.float_info.epsilon:
                print("nulu nelze zadat")
                x = None
        except ValueError as err:
           print(err)
    return x
```

Tato funkce bude procházet cyklem, dokud uživatel nezadá platné číslo typu `float` (např. `0.5`, `-9`, `21`, `4.92`); hodnotu `0` přijme pouze tehdy, má-li parametr `allow_zero` hodnotu `True`.

Jakmile je funkce `get_float()` definována, provede se zbývající část kódu. Projdeme si jej ve třech částech a začneme interakcí s uživatelem:

```
print("ax\N{SUPERSCRIPT TWO} + bx + c = 0")
a = get_float("zadejte a: ", False)
b = get_float("zadejte b: ", True)
c = get_float("zadejte c: ", True)
```

Díky funkci `get_float()` je získání koeficientů a, b a c velice jednoduché. Druhý argument představující pravdivostní hodnotu říká, zda lze zadat nulu.

```
x1 = None
x2 = None
discriminant = (b ** 2) - (4 * a * c)
if discriminant == 0:
    x1 = -(b / (2 * a))
else:
    if discriminant > 0:
        root = math.sqrt(discriminant)
    else:  # diskriminant < 0
```

```
        root = cmath.sqrt(discriminant)
    x1 = (-b + root) / (2 * a)
    x2 = (-b - root) / (2 * a)
```

Kód vypadá trošku jinak než vzorec, protože musíme začít výpočtem diskriminantu. Má-li diskriminant hodnotu 0, pak víme, že máme jedno řešení, které můžeme přímo vypočítat. V opačném případě provedeme reálnou a komplexní odmocninu diskriminantu a vypočítáme dva kořeny.

```
equation = ("{0}x\N{SUPERSCRIPT TWO} + {1}x + {2} = 0"
            " \N{RIGHTWARDS ARROW} x = {3}").format(a, b, c, x1)
if x2 is not None:
    equation += " nebo x = {0}".format(x2)
print(equation)
```

Použití metody `str.format()` s rozbalením mapování ➤ 86

Neprovádíme žádné efektní formátování, protože výchozí formát čísel s pohyblivou řádovou čárkou v Pythonu je pro účely tohoto příkladu dostatečný. Na druhou stranu používáme pro několik speciálních znaků názvy znaků ze znakové sady Unicode.

Robustnější alternativa oproti pozičním argumentům, jejichž názvy polí jsou definovány pomocí indexů, spočívá v použití slovníku vráceného funkcí `locals()`, což je technika, s níž jsme se seznámili v dřívější části této lekce:

```
equation = ("{a}x\N{SUPERSCRIPT TWO} + {b}x + {c} = 0"
            " \N{RIGHTWARDS ARROW} x = {x1}").format(**locals())
```

3.1

A pokud používáme Python 3.1, pak bychom mohli vynechat názvy polí a nechat Python, aby sám tato pole naplnit pomocí pozičních argumentů předaných metodě `str.format()`.

```
equation = ("{}x\N{SUPERSCRIPT TWO} + {}x + {} = 0"
            " \N{RIGHTWARDS ARROW} x = {}").format(a, b, c, x1)
```

Jedná se o pohodlné řešení, které ale není tak robustní jako pojmenované parametry, ani tak všestranné, pokud bychom potřebovali použít specifikace formátu. Nicméně v řadě jednoduchých případů je tato syntaxe snadná a užitečná.

Program csv2html.py

Jedním z běžných požadavků je vzít datovou sadu a zobrazit ji uživateli pomocí jazyka HTML. V této části napíšeme program, který přečte soubor v jednoduchém formátu CSV (Comma Separated Value – hodnoty oddělené čárkou) a vypíše tabulku HTML obsahující jeho data. Python nabízí výkonný a sofistikovaný modul `csv` pro práci s CSV a podobnými formáty, my si však napíšeme veškerý kód sami.

Formát CSV, který budeme podporovat, má na jednom řádku jeden záznam, přičemž každý záznam je rozdělen do polí pomocí čárek. Každé pole může být buď řetězec, nebo číslo. Řetězce musejí být uzavřené do jednoduchých nebo dvojitých uvozovek a čísla by měla být v uvozovkách jen v případě, že obsahuje čárky. Čárky mohou být též uvnitř řetězců, kde se nesmějí považovat za oddělovače polí. Předpokládáme, že první záznam obsahuje pole popisků. Námi vytvářený výstup bude mít podobu

tabulky HTML s textem zarovnaným vlevo (výchozí zarovnání v jazyku HTML) a s čísly zarovnanými vpravo, kdy na jednom řádku bude jeden záznam a v jedné buňce jedno pole.

Program musí vypsat otevírací značku tabulky jazyka HTML, poté přečíst každý řádek data, u každého z nich vypsat příslušný řádek tabulky HTML a na konec vypsat ukončovací značku tabulky HTML. Barva pozadí prvního řádku (který zobrazí popisky polí) bude světle zelená a na pozadí datových řádků se bude střídat bílá a světle žlutá. Musíme se také ujistit, že se speciální znaky jazyka HTML („&“, „<“ a „>“) náležitým způsobem zakódovaly, a dále chceme, aby se řetězce zobrazily trošku čistším způsobem.

Zde je úryvek s ukázkovými daty:

```
"ZEMĚ","2000","2001",2002,2003,2004
"ANTIGUA A BARBUDA",0,0,0,0,0
"ARGENTINA",37,35,33,36,39
"BAHAMY, THE",1,1,1,1,1
"BAHRAJN",5,6,6,6,6
```

Za předpokladu, že ukázková data jsou uložena v souboru *data/co2-sample.csv* a že uživatel zadal příkaz `csv2html.py < data/co2-sample.csv > co2-sample.html`, bude mít soubor *co2-sample.html* obsah podobný následujícímu:

```
<table border='1'><tr bgcolor='lightgreen'>
<td>Země</td><td align='right'>2000</td><td align='right'>2001</td>
<td align='right'>2002</td><td align='right'>2003</td>
<td align='right'>2004</td></tr>
...
<tr bgcolor='lightyellow'><td>Argentina</td>
<td align='right'>37</td><td align='right'>35</td>
<td align='right'>33</td><td align='right'>36</td>
<td align='right'>39</td></tr>
...
</table>
```

Výstup jsme malinko upravili a vynechali jsme některé řádky, místo kterých je uveden výpustek. Použili jsme velmi jednoduchou verzi jazyka HTML (HTML 4 Transitional) bez šablony stylů. Obrázek 2.7 ukazuje, jak tento výstup vypadá ve webovém prohlížeči.

Země	2000	2001	2002	2003	2004
Antigua a Barbuda	0	0	0	0	0
Argentina	37	35	33	36	39
Bahamy	1	1	1	1	1
Bahrajn	5	6	6	6	6

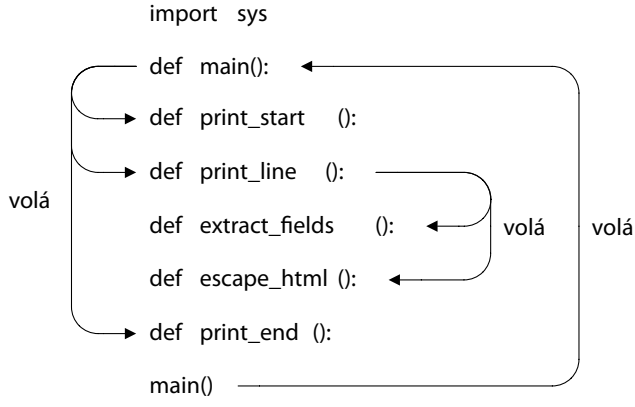
Obrázek 2.7: Tabulka z programu csv2html.py ve webovém prohlížeči

Nyní již tedy víme, jak se program používá a co dělá, takže se můžeme pustit do průzkumu jeho zdrojového kódu. Celý program začíná importem modulu `sys`. My si jej zde neukážeme a od této chvíle si už nebudeme ukazovat žádné importy, pokud nebudou nějak výjimečné nebo hodné výkladu. Posledním příkazem v programu je jediné volání funkce:

```
main()
```

I když Python na rozdíl od některých jiných jazyků žádný vstupní bod nepotřebuje, je docela běžné, že programátoři v Pythonu ve svých programech vytvářejí funkci s názvem `main()`, kterou pak volají pro zahájení činnosti programu. Žádnou funkci nelze zavolat před jejím vytvořením, a proto je nutné zajistit, aby se funkce `main()` zavolala až poté, kdy budou definovány všechny funkce, na kterých závisí. Na pořadí, ve kterém jsou funkce v souboru uvedeny (tj. pořadí, ve kterém jsou vytvářeny), vůbec nezáleží.

V programu csv2html.py je první volanou funkcí funkce `main()`, která zase volá funkce `print_start()` a `print_line()`, přičemž funkce `print_line()` volá funkce `extract_fields()` a `escape_html()`. Struktura našeho programu je znázorněna na obrázku 2.8.



Obrázek 2.8: Struktura programu csv2html.py

Když Python čte soubor, začíná odshora. V tomto příkladu tedy začíná provedením importu, poté vytvoří funkci `main()` a poté vytvoří ostatní funkce v pořadí, ve kterém jsou v souboru uvedeny. Když Python nakonec dospěje k volání funkce `main()` na konci souboru, budou všechny funkce, které `main()` volá (a všechny funkce, které tyto funkce volají), již existovat. Provádění tak, jak o něm běžně uvažujeme, začíná v okamžiku zavolání funkce `main()`.

Nyní se postupně podíváme na každou z funkcí, přičemž začneme funkcí `main()`:

```
def main():
    maxwidth = 100
    print_start()
    count = 0
    while True:
        try:
            line = input()
```

```
            if count == 0:
                color = "lightgreen"
            elif count % 2:
                color = "white"
            else:
                color = "lightyellow"
            print_line(line, color, maxwidth)
            count += 1
        except EOFError:
            break
    print_end()
```

Proměnná `maxwidth` slouží k omezení počtu znaků v buňce. Je-li pole větší než tato hodnota, tak jej ořežeme a za oříznutý text přidáme výpustek. Na funkce `print_start()`, `print_line()` a `print_end()` se podíváme vzápětí. Cyklus `while` prochází přes všechny řádky vstupu, který sice může pocházet i od uživatele píšícího na klávesnici, my však očekáváme, že se bude jednat o přesměrovaný soubor. Nastavíme barvu, kterou chceme použít, a pomocí funkce `print_line()` vypíšeme řádek jako řádek tabulky HTML:

```
def print_start():
    print("<table border='1'>")


def print_end():
    print("</table>")
```

Tyto dvě funkce bychom ani vytvářet nemuseli. Stačilo by jen umístit příslušná volání funkce `print()` do těla funkce `main()`. Naším cílem je však oddělovat logiku programu, což je flexibilnější, i když v takto malém příkladě to nehraje žádnou roli.

```
def print_line(line, color, maxwidth):
    print("<tr bgcolor='{0}'>".format(color))
    fields = extract_fields(line)
    for field in fields:
        if not field:
            print("<td></td>")
        else:
            number = field.replace(",", "")
            try:
                x = float(number)
                print("<td align='right'>{0:d}</td>".format(round(x)))
            except ValueError:
                field = field.title()
                field = field.replace(" A ", " a ")
                if len(field) <= maxwidth:
                    field = escape_html(field)
                else:
                    field = "{0} ...".
```

```
                        format(escape_html(field[:maxwidth]))
                print("<td>{0}</td>".format(field))
    print("</tr>")
```

K rozdělení každého řádku do polí nemůžeme použít příkaz `str.split(",")`, protože čárky se mohou vyskytovat také uvnitř řetězců uzavřených do uvozovek. Naše řešení jsme tedy umístili do funkce `extract_fields()`. Jakmile máme seznam polí (jako řetězců bez obklopujících uvozovek), můžeme je projít a pro každé vytvořit buňku tabulky.

Je-li pole prázdné, vypíšeme prázdnou buňku. Je-li pole uzavřené do uvozovek, pak by mohlo jít o řetězec nebo číslo, které bylo uzavřeno do uvozovek kvůli interním čárkám (např. `"1,566"`). Tuto situaci vyřešíme tak, že vytvoříme kopii pole s odstraněnými čárkami a toto pole se pokusíme převést na hodnotu typu `float`. Je-li převod úspěšný, vypíšeme vpravo zarovnanou buňku s polem zaokrouhleným na nejbližší celé číslo, které vypíšeme jako celé číslo. Pokud převod selže, vypíšeme pole jako řetězec. V takovém případě použijeme metodu `str.title()` k úpravě velikosti písmen a dále nahradíme osamělá písmena `"A"` písmeny `"a"`. Pokud pole není příliš dlouhé, použijeme jej celé. V opačném případě jej ořízneme na velikost definovanou proměnnou `maxwidth` a přidáme výpustek označující oříznutí. V obou případech zakódujeme speciální znaky jazyka HTML, které by mohlo dané pole obsahovat.

```
def extract_fields(line):
    fields = []
    field = ""
    quote = None
    for c in line:
        if c in "\"'":
            if quote is None: # začátek řetězce uzavřeného do uvozovek
                quote = c
            elif quote == c:  # konec řetězce uzavřeného do uvozovek
                quote = None
            else:
                field += c    # další uvozovku uvnitř řetězce v uvozovkách
            continue
        if quote is None and c == ",": # konec pole
            fields.append(field)
            field = ""
        else:
            field += c        # akumulace pole
    if field:
        fields.append(field)  # přidání posledního pole
    return fields
```

Tato funkce přečte zadaný řádek znak po znaku a postupně vytvoří seznam polí, z nichž každé je řetězcem bez uzavíracích uvozovek. Funkce si poradí i s poli, která nejsou uzavřené do uvozovek, dále s poli, která jsou uzavřená do jednoduchých či dvojitých uvozovek, a správně zpracuje také čárky a uvozovky (jednoduché uvozovky v řetězcích uzavřených do dvojitých uvozovek a dvojité uvozovky v řetězcích uzavřených do jednoduchých uvozovek).

```
def escape_html(text):
    text = text.replace("&", "&amp;")
    text = text.replace("<", "&lt;")
    text = text.replace(">", "&gt;")
    return text
```

Tato funkce jednoduše nahradí každý speciální znak jazyka HTML příslušnou entitou jazyka HTML. Nejdříve je samozřejmě nutné nahradit ampersand, ačkoliv na pořadí ostrých závorek už nezáleží. Standardní knihovna Pythonu obsahuje malinko sofistikovanější verzi této funkce. Budete mít šanci se s ní seznámit ve cvičeních a opět se s ní setkáte v lekci 7.

Shrnutí

Na začátku této lekce jsme si ukázali seznam klíčových slov jazyka Python a popsali jsme si pravidla, která Python aplikuje na identifikátory. Ty nejsou díky podpoře znakové sady Unicode v jazyku Python omezeny pouze na určitou podmnožinu znaků z malé znakové sady, jako je ASCII nebo Latin-1.

Dále jsme si popsali datový typ `int` jazyka Python, který se od podobných typů ve většině ostatních jazyků liší v tom, že nemá žádné skutečné omezení velikosti. Celá čísla v jazyku Python mohou být tak velká, jak dovoluje paměť počítače, a není vůbec žádný problém pracovat s čísly se stovkami cifer. Všechny nejzákladnější datové typy jazyka Python jsou neměnitelné, což lze jen zřídkakdy postřehnout, protože díky operátorům rozšířeného přiřazení (`+=`, `*=`, `-=`, `/=` a další) můžeme používat velice přirozenou syntaxi, zatímco interpret jazyka Python vytváří v pozadí výsledné objekty, s nimiž opětně svazuje naše proměnné. Celočíselné literály se obvykle zapisují jako desítková čísla, ovšem pomocí prefixu `0b` lze psát binární literály, pomocí prefixu `0o` osmičkové literály a pomocí prefixu `0x` šestnáctkové literály.

Při dělení dvou celých čísel pomocí operátoru `/` je výsledek vždy typu `float`. To je sice od mnoha jiných běžně používaných jazyků odlišné, ale pomáhá to předcházet některých docela zákeřným chybám, k nimž může docházet, když dělení výsledek tiše ořeže. (Chceme-li přesto použít celočíselné dělení, můžeme sáhnout po operátoru `//`.)

Jazyk Python nabízí datový typ `bool`, který umí uchovávat buď `True`, nebo `False`. Dále Python nabízí logické operátory `and`, `or` a `not`, z nichž oba dva binární operátory (`and` a `or`) používají logiku zkráceného vyhodnocování.

K dispozici jsou tři druhy čísel s pohyblivou řádovou čárkou: `float`, `complex` a `decimal.Decimal`. Nejčastěji se používá typ `float`, který reprezentuje s dvojitou přesností číslo s pohyblivou řádovou čárkou, jehož přesné číselné charakteristiky závisejí na knihovně jazyka C, C# nebo Java, s níž byl Python sestaven. Komplexní čísla jsou reprezentována jako dvě čísla typu `float`. Jedno uchovává reálnou a druhé imaginární hodnotu. Typ `decimal.Decimal` poskytuje modul `decimal`. Čísla tohoto typu mají ve výchozím nastavení 28 desetinných míst. Tuto přesnost lze ale dále zvýšit nebo snížit tak, aby vyhovovala aktuálním potřebám.

Všechny tři typy s pohyblivou řádovou čárkou lze použít s příslušnými vestavěnými matematickými operátory a funkcemi. Kromě toho modul `math` nabízí pestrou škálu trigonometrických, hyper-

bolických a logaritmických funkcí, které lze použít s čísly typu `float`, přičemž modul `cmath` nabízí podobnou skupinu funkcí pro komplexní čísla.

Větší část lekce byla věnována řetězcům. Řetězcové literály jazyka Python lze vytvářet pomocí jednoduchých či dvojitých uvozovek. Můžeme také použít řetězce s trojitými uvozovkami, chceme-li do řetězce přímo zahrnout znaky nových řádků a uvozovky. K vkládání speciální znaků (jako je např. tabulátor a nový řádek) lze použít speciální posloupnosti znaků (`\t` a `\n`). Dále je možné pomocí šestnáctkových kódů nebo specifických názvů vkládat znaky ze znakové sady Unicode. Ačkoliv řetězce podporují stejné porovnávací operátory jako ostatní typy jazyka Python, je třeba si uvědomit, že řazení řetězců obsahujících neanglické znaky může být problematické.

Řetězce jsou posloupnosti, a proto lze pomocí velmi jednoduché, přesto však výkonné syntaxe řezacího operátoru (`[]`) řetězce řezat a krokovat. Řetězce lze také operátorem + spojovat a operátorem * replikovat. Dále můžeme použít verze těchto operátorů s rozšířeným přiřazením (+= a *=), i když častěji se pro spojování řetězců používá metoda `str.join()`. Řetězce mají spoustu dalších metod, včetně metod pro testování vlastností řetězce (např. `str.isspace()` a `str.isalpha()`), pro změnu velikosti písmen (např. `str.lower()` a `str.title()`), pro vyhledávání (např. `str.find()` a `str.index()`) a řady dalších.

Podpora řetězců v jazyku Python je skutečně excelentní; díky niž můžeme texty snadno vyhledávat a extrahovat nebo celé řetězce či části řetězců porovnávat, nahrazovat znaky či podřetězce, rozdělovat řetězce na seznam podřetězců a spojovat seznamy řetězců do jediného řetězce.

Pravděpodobně nejvšestrannější metodou je metoda `str.format()`. Tato metoda se používá pro vytváření řetězců pomocí nahrazovacích polí a proměnných, které se dosadí do těchto polí, a pomocí specifikací formátu, které přesně definují charakteristiky každého pole nahrazovaného nějakou hodnotu. Syntaxe pro názvy nahrazovacích polí nám umožňuje přistupovat k argumentům metody podle jejich pozice nebo jména (pro klíčované argumenty) a pomocí názvu indexu, klíče nebo atributu přistupovat k položce nebo atributu argumentu. Díky specifikaci formátu můžeme stanovit výplňový znak, zarovnání a minimální šířku pole. Kromě toho u čísel můžeme nastavovat, jak se zobrazí znaménko, a u čísel s pohyblivou řádovou čárkou můžeme určit počet desetinných míst a také to, zda se má použít standardní nebo exponenciální notace.

Probírali jsme také spletité téma kódování znaků. Soubory `.py` s kódem jazyka Python používají standardně kódování UTF-8 znakové sady Unicode, a proto mohou mít komentáře, identifikátory a data zapsaná v téměř libovolném lidském jazyku. Pomocí metody `str.encode()` můžeme převést řetězec na posloupnost bajtů s použitím určitého kódování a metodou `bytes.decode()` můžeme posloupnost bajtů v určitém kódování převést zpět na řetězec. Práce s pestrou škálou kódování znaků, které se v současné době používají, může být velmi nepohodlná, avšak kódování UTF-8 se v oblasti holých textových souborů rychle stává de facto standardem (a již nyní je standardem pro soubory XML), takže tento problém by měl v následujících letech slábnout.

Kromě datových typů probíraných v této lekci nabízí jazyk Python dva další vestavěné datové typy `bytes` a `bytearray`, kterým se budeme věnovat v lekci 7. Jazyk Python dále nabízí několik datových typů představujících kolekce, z nichž některé jsou vestavěné a jiné jsou součástí standardní knihovny. V následující lekci se tedy podíváme na nejdůležitější datové typy jazyka Python určené pro práci s kolekcemi.

Cvičení

1. Upravte program `print_unicode.py` tak, aby uživatel mohl na příkazovém řádku zadat několik samostatných slov a aby se vytiskly pouze ty řádky, jejichž název znaku znakové sady Unicode obsahuje všechna slova zadaná uživatelem. To znamená, že můžeme napsat příkazy, jako je tento:

   ```
   print_unicode_ans.py greek symbol
   ```

   Jeden ze způsobů, jak to provést, spočívá v nahrazení proměnné `word` (která obsahovala hodnotu `0`, `None` nebo řetězec) seznamem `words`. Nezapomeňte aktualizovat kromě kódu také informace o použití. Změny zahrnují přidání méně než deseti řádků kódu a změnu maximálně deseti dalších. Řešení najdete v souboru `print_unicode_ans.py`. (Uživatelé systému Windows by měli upravovat soubor `print_unicode_uni.py`; výsledek naleznou v souboru `print_unicode_uni_ans.py`.)

2. Upravte program `quadratic.py` tak, aby se koeficienty s hodnotou 0,0 nevypisovaly a aby se záporné koeficienty vypisovaly jako `- n` a ne `+ -n`. To zahrnuje nahrazení posledních pěti řádků zhruba patnácti řádky. Řešení najdete v souboru `quadratic_ans.py`. (Uživatelé systému Windows by měli upravovat soubor `quadratic_uni.py`, výsledek naleznou v souboru `quadratic_uni_ans.py`.)

3. V programu `csv2html.py` vymažte funkci `escape_html()` a použijte místo ní funkci `xml.sax.saxutils.escape()` z modulu `xml.sax.saxutils`. To je snadné, stačí jen jeden nový řádek (příkaz import), pět řádků vymazat (nechtěná funkce) a jeden řádek změnit (použití funkce `xml.sax.saxutils.escape()` místo `escape_html()`). Řešení je k dispozici v souboru `csv2html1_ans.py`.

4. Znovu upravte soubor csv2html.py a tentokrát přidejte novou funkci s názvem `process_options()`. Tato funkce by se měla volat z funkce `main()` a měla by vracet n-tici dvou hodnot: `maxwidth` (typu `int`) a `format` (typu `str`). Funkce `process_options()` by měla při zavolání nastavit výchozí hodnotu proměnné `maxwidth` na `100` a výchozí hodnotu proměnné `format` na `"%.0f"` – tato hodnota se pak použije jako specifikátor formátu při vypisování čísel.

   Pokud uživatel na příkazovém řádku zadá „-h“ nebo „--help”, vypíše se zpráva s informacemi o použití a vrátí dvojice (`None`, `None`). (V tomto případě by funkce `main()` neměla provést vůbec nic.) V opačném případě by tato funkce měla přečíst argumenty zadané na příkazovém řádku a provést příslušná přiřazení. Když například uživatel zadá „`maxwidth=n`“, pak se do proměnné `maxwidth` přiřadí hodnota n a podobně se nastaví proměnná formát při zadání „`format=s`“. Zde je ukázkový běh zobrazující výpis informací o použití:

   ```
   csv2html2_ans.py -h
   použití:
   csv2html.py [maxwidth=int] [format=str] < infile.csv > outfile.html

   Parametr maxwidth je volitelné celé číslo;
   je-li zadáno, nastaví maximum počet znaků,
   které lze vypsat u řetězcových polí,
   ```

```
jinak se tato pole ořežou na 100 znaků.

Parametr format je formát pro čísla;
není-li zadán, použije se ".0f".
A zde je příkazový řádek s nastavením obou voleb:
csv2html2_ans.py maxwidth=20 format=0.2f < mydata.csv > mydata.html
```

Nezapomeňte upravit funkci `print_line()` tak, aby pro výpis čísel používala nastavený formát. Bude třeba jí předat další argument, přidat jeden řádek a upravit jiný řádek. To bude mít také malý vliv na funkce `main()`. Funkce `process_options()` by měla mít přibližně dvacet pět řádků (včetně asi tak devíti pro zprávu s informacemi o použití). Toto cvičení může být pro nezkušené programátory obtížnější.

K dispozici máte dva soubory s testovacími daty: `data/co2-sample.csv` a `data/co2-fromfossil-fuels.csv`. Řešení je k dispozici v souboru `csv2html2_ans.py`. V lekci 5 uvidíte, jak pomocí modulu `optparse` zjednodušit zpracování argumentů na příkazovém řádku.

LEKCE 3

Datové typy představující kolekce

V této lekci:

- Typy představující posloupnost
- Množinové typy
- Typy představující mapování
- Procházení a kopírování kolekcí

V předchozí lekci jsme se seznámili s nejdůležitějšími základními datovými typy jazyka Python. V této lekci naše programovací dovednosti rozšíříme o možnost shromažďovat datové prvky pomocí datových typů jazyka Python představující kolekce. Budeme se věnovat n-ticím a seznamům a představíme si také nové datové typy představující kolekce, mezi něž patří množiny a slovníky, a všechny si je projdeme pořádně do hloubky.*

Kromě kolekcí se podíváme také na to, jak vytvářet datové prvky, které jsou seskupením jiných datových prvků (podobně jako v případě typu `struct` v jazycích C a C++ nebo `record` v jazyku Pascal); s takovými prvky lze pak v případě potřeby pracovat jako s jednotlivým elementem, zatímco v nich obsažené prvky zůstávají samostatně přístupné. Seskupené prvky můžeme pochopitelně umisťovat do kolekcí jako jakékoli jiné prvky.

Díky možnosti vkládat datové prvky do kolekce je pak mnohem snazší provádět operace, které je třeba aplikovat na všechny tyto prvky. Kromě toho se pak kolekce prvků snadněji čtou ze souborů. V této lekci se budeme průběžně věnovat také úplným základům práce s textovými soubory, přičemž většinu podrobností (včetně ošetřování chyb) si vysvětlíme v lekci 7.

Jakmile probereme jednotlivé datové typy představující kolekce, podíváme se na to, jak kolekce procházet, protože stejná syntaxe se používá pro všechny kolekce jazyka Python. Rozebereme si též problémy a techniky související s kopírováním kolekcí.

Typy představující posloupnost

Typ představující posloupnost je takový typ, který podporuje operátor příslušnosti (`in`), funkci zjišťující velikost (`len()`) a řezání (`[]`) a který je iterovatelný. Jazyk Python nabízí pět vestavěných typů představujících posloupnost: `bytearray` (pole bajtů), `bytes` (bajty), `list` (seznam), `str` (řetězec) a `tuple` (n-tice). První dva typy budeme probírat samostatně v lekci 7. Standardní knihovna poskytuje ještě několik dalších typů představujících posloupnost, z nichž nejpozoruhodnější je typ `collections.namedtuple`. Při procházení vracejí všechny tyto posloupnosti své prvky v určitém pořadí.

Řetězce ➢ 71

V předchozí lekci jsme se věnovali řetězcům. V této části se zaměříme na n-tice, pojmenované n-tice a seznamy.

N-tice

Řezání a krokování řetězců ➢ 74

N-tice je uspořádaná posloupnost nula nebo více odkazů na objekty. N-tice podporují stejnou syntaxi pro řezání a krokování jako řetězce. Díky tomu je extrahování prvků z n-tice velice snadné. Podobně jako řetězce jsou také n-tice neměnitelné, takže nemůžeme nahradit nebo vymazat žádný z jejich prvků. Pokud chceme mít možnost modifikovat uspořádanou posloupnost, musíme místo n-tice použít seznam. Pokud už n-tici máme a chceme ji upravit, pak ji můžeme pomocí převodní funkce `list()` převést na seznam a poté aplikovat změny na výsledný seznam.

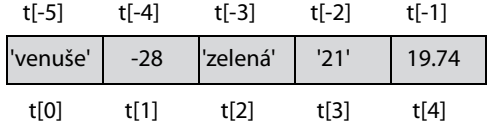
Mělké a hloubkové kopírování ➢ 146

Datový typ `tuple` lze zavolat i jako funkci `tuple()`. Bez argumentů vrátí prázdnou n-tici, s argumentem typu `tuple` vrátí jeho mělkou kopii a argument jakéhokoliv jiného typu se pokusí převést na n-tici. Funkci `tuple()` nelze předat více než jeden argument. N-tice je možné vytvářet i bez funkce `tuple()`. Prázdnou n-tici vytvoříme pomocí prázdných závorek `()` a n-tici s jedním nebo více prvky

* Definice v této lekci týkající se toho, co je typ představující posloupnost nebo typ představující mapování, jsou sice praktické, ale neformální. S formálnějšími definicemi se seznámíme v lekci 8.

vytvoříme pomocí čárek. Někdy je nutné n-tice uzavřít do závorek, abychom předešli syntaktické nejednoznačnosti. Kupříkladu k předání n-tice `1, 2, 3` nějaké funkci musíme napsat `function ((1, 2, 3))`.

Obrázek 3.1 ukazuje n-tici `t = "venuše", -28, "zelená", "21", 19.74` a indexové pozice jednotlivých prvků uvnitř této n-tice. Řetězce se indexují podobným způsobem, avšak zatímco řetězce mají na každé pozici znak, n-tice mají na každé pozici odkaz na objekt.



Obrázek 3.1 Indexové pozice v n-tici

N-tice nabízejí jen dvě metody. Metoda `t.count(x)` vrací počet výskytů objektu `x` v n-tici `t` a metoda `t.index()` vrací pozici nejlevějšího výskytu objektu `x` v n-tici `t` nebo vyvolá výjimku `ValueError`, pokud v n-tici žádný objekt `x` není. (Tyto metody jsou k dispozici také u seznamů.)

Kromě toho lze n-tice použít s operátory + (spojení), * (replikace) a `[]` (řezání) a také s operátory `in` a `not in` pro testování příslušnosti. Ačkoliv n-tice jsou neměnné, můžeme použít i operátory rozšířeného přiřazení += a *=, což je možné díky tomu, že Python v pozadí vytvoří novou n-tici pro uchování výsledku a nastaví odkaz na objekt na levé straně tak, aby ukazoval na tuto nově vytvořenou n-tici. Stejná technika se používá i při aplikaci těchto operátorů na řetězce. N-tice lze porovnávat pomocí standardních porovnávacích operátorů (<, <=, ==, !=, >=, >), přičemž porovnávání se aplikuje na jednotlivé prvky (a rekurzivně pro vnořené prvky, jako jsou n-tice uvnitř n-tic).

Pojďme se podívat na několik příkladů řezání. Začneme extrahováním jednoho prvku a řezu prvků:

```
>>> hair = "černá", "hnědá", "blond", "červená"
>>> hair[2]
'blond'
>>> hair[-3:]    # stejné jako: hair[1:]
('hnědá', 'blond', 'červená')
```

Tyto příkazy fungují stejně u řetězců, seznamů a u jakýchkoli jiných typů představujících posloupnost.

```
>>> hair[:2], "šedá", hair[2:]
(('černá', 'hnědá'), 'šedá', ('blond', 'červená'))
```

Zde jsme se pokusili vytvořit novou pětici, ale obdrželi jsme trojici, která obsahuje dvě dvojice. Příčinou je to, že jsme použili operátor čárka se třemi prvky (n-tice, řetězec a n-tice). Pro získání jediné n-tice se všemi prvky musíme jednotlivé n-tice spojit:

```
>>> hair[:2] + ("šedá",) + hair[2:]
('černá', 'hnědá', 'šedá', 'blond', 'červená')
```

K vytvoření n-tice s jedním prvkem stačí napsat čárku, avšak v tomto případě bychom při vynechání závorek obdrželi chybu `TypeError` (poněvadž Python by si myslel, že se pokoušíme spojit řetězec s n-ticí).

V této knize (od tohoto okamžiku) budeme pro zápis n-tic používat určitý styl. Pokud budeme mít n-tice na levé straně binárního operátoru nebo na pravé straně unárního příkazu, pak závorky vynecháme; ve všech ostatních případech budeme závorky uvádět. Zde je několik příkladů:

```
a, b = (1, 2)                          # levá strana binárního operátoru

del a, b                               # pravá strana unárního operátoru

def f(x):
    return x, x ** 2                   # pravá strana unárního příkazu

for x, y in ((1, 1), (2, 4), (3, 9)):  # levá strana binárního operátoru
    print(x, y)
```

Nejedná se o styl programování, který by bylo nutné bezpodmínečně dodržovat. Například někteří programátoři raději píší závorky pokaždé, což je stejné jako reprezentační forma n-tice, zatímco jiní je používají pouze tam, kde jsou nezbytně nutné.

```
>>> eyes = ("hnědá", "ořechová", "jantarová", "zelená", "modrá", "šedá")
>>> colors = (hair, eyes)
>>> colors[1][3:-1]
('zelená', 'modrá')
```

Zde jsme vnořili dvě n-tice do jiné n-tice. Tímto způsobem lze přímo vytvářet vnořené kolekce libovolné hloubky. Na řez můžeme dle potřeby opakovaně aplikovat řezací operátor `[]`:

```
>>> things = (1, -7.5, ("hrách", (5, "Xyz"), "fronta"))
>>> things[2][1][1][2]
'z'
```

Rozeberme si tento kód krok za krokem. Výraz `things[2]` se vyhodnotí na třetí prvek n-tice (neboť první prvek má index `0`), což je n-tice `("hrách", (5, "Xyz"), "fronta")`. Výraz `things[2][1]` nám dá druhý prvek n-tice `things[2]`, což je opět n-tice `(5, "Xyz")`. A výraz `things[2][1][1]` se vyhodnotí na druhý prvek této n-tice, což je řetězec `"Xyz"`. Nakonec nám výraz `things[2][1][1][2]` vrátí třetí prvek (znak) v řetězci, což je `"z"`.

N-tice jsou schopné uchovávat libovolné prvky libovolného datového typu, včetně typů představujících kolekce, jako jsou n-tice a seznamy, protože ve skutečnosti se neuchovávají samotné objekty, ale odkazy na objekty. Použití podobně složité zanořené datové struktury může být matoucí. Jedno z řešení spočívá v přiřazení jmen určitým indexovým pozicím:

```
>>> MANUFACTURER, MODEL, SEATING = (0, 1, 2)
>>> MINIMUM, MAXIMUM = (0, 1)
>>> aircraft = ("Airbus", "A320-200", (100, 220))
```

```
>>> aircraft[SEATING][MAXIMUM]
220
```

Tento kód je jistě mnohem smysluplnější než zápis `aircraft[2][1]`, vyžaduje však vytvoření spousty proměnných a nevypadá hezky. V následující části se seznámíme s elegantním řešením tohoto problému.

Na prvních dvou řádcích výše uvedeného úryvku kódu jsme v obou příkazech provedli přiřazení do n-tic. Máme-li na pravé straně přiřazení posloupnost (zde máme n-tici) a na levé straně n-tici, pak říkáme, že pravá strana byla *rozbalena* (unpack). Rozbalení posloupnosti lze použít k prohození hodnot:

```
a, b = (b, a)
```

Přesně řečeno, závorky na pravé straně nejsou nutné, ale jak jsme si řekli již dříve, v této knize budeme psát kód takovým stylem, že závorky vynecháme u operandů na levé straně binárních operátorů a u operandů na pravé straně unárních příkazů, přičemž všude jinde budeme závorky psát.

Již jsme se setkali s příklady rozbalení posloupnosti v kontextu cyklů `for ... in`:

```
for x, y in ((-3, 4), (5, 12), (28, -45)):
    print(math.hypot(x, y))
```

Zde procházíme přes n-tici dvojic, které postupně rozbalujeme do proměnných `x` a `y`.

Pojmenované n-tice

Pojmenovaná n-tice se chová stejně jako holá n-tice a vykazuje stejné výkonové charakteristiky. To, co přidává, je možnost odkazovat na prvky v n-tici kromě indexové pozice také jménem, díky čemuž můžeme vytvářet seskupení datových prvků.

Modul `collections` nabízí funkci `namedtuple()`, která se používá k vytváření vlastních datových typů představujících n-tice:

```
Sale = collections.namedtuple("Sale",
                              "productid customerid date quantity price")
```

Prvním argumentem funkce `collections.namedtuple()` je název datového typu n-tice, který chceme vytvořit. Druhý argument je řetězec mezerou oddělených jmen, jedno pro každý prvek, pod kterými budou vystupovat prvky našich vlastních n-tic. První argument a jména ve druhém argumentu musejí být platnými identifikátory jazyka Python. Funkce vrátí naši novou třídu (datový typ), jejímž prostřednictvím můžeme vytvářet pojmenované n-tice. V tomto případě můžeme tedy pracovat s výrazem `Sale` jako s jakoukoli jinou třídou jazyka Python (jako je např. `tuple`) a vytvářet objekt typu `Sale`. (V řeči objektově orientovaného programování je každá takto vytvářená třída podtřídou třídy `tuple`. Objektově orientovanému programování včetně odvozování nových tříd se budeme věnovat v lekci 6.)

Zde je příklad:

```
sales = []
sales.append(Sale(432, 921, "2008-09-14", 3, 79.9))
sales.append(Sale(419, 874, "2008-09-15", 1, 184.9))
```

Zde jsme vytvořili seznam prvků dvou prvků typu `Sale`, tedy, dvou našich vlastních n-tic. Na prvky v těchto n-ticích se můžeme odkazovat pomocí indexových pozic, takže například hodnota prvního prvku je `sales[0][-1]` (tj. `79.9`). Kromě toho však můžeme použít také jména, díky kterým je výsledný kód daleko čistší:

```
total = 0
for sale in sales:
    total += sale.quantity * sale.price
print("Celkem {0:.2f} Kč".format(total))  # vypíše: Celkem 424.6 Kč
```

Čitelnost a pohodlnost nabízená pojmenovanými n-ticemi je často velice užitečná. Vraťme se nyní k příkladu s letadlem z předchozí části (strana 112), který můžeme přepsat do mnohem pěknějšího tvaru:

```
>>> Aircraft = collections.namedtuple("Aircraft",
                                      "manufacturer model seating")
>>> Seating = collections.namedtuple("Seating", "minimum maximum")
>>> aircraft = Aircraft("Airbus", "A320-200", Seating(100, 220))
>>> aircraft.seating.maximum
220
```

Co se týče extrahování prvků z pojmenované n-tice do řetězců, můžeme použít jednu ze tří hlavních technik.

```
>>> print("{0} {1}".format(aircraft.manufacturer, aircraft.model))
Airbus A320-200
```

Zde přistupujeme ke každému prvku n-tice, který nás zajímá, pomocí přístupu k atributům pojmenované n-tice. Díky tomu obdržíme nejkratší a nejjednodušší formátovací řetězec. (A v Pythonu 3.1 bychom jej mohli ještě zmenšit jen na `"{} {}"`.) Avšak tento přístup znamená, že musíme sledovat argumenty předávané metodě `str.format()`, abychom měli přehled o tom, jaké nahrazovací texty se použijí. To je méně čisté ve srovnání s pojmenovanými poli uvedenými přímo ve formátovacím řetězci.

```
"{0.manufacturer} {0.model}".format(aircraft)
```

Zde jsme použili jediný poziční argument a názvy atributů pojmenované n-tice jako názvy polí ve formátovacím řetězci. To je sice ve srovnání se samotnými pozičními argumenty mnohem čistší, je ale škoda, že musíme specifikovat poziční hodnotu (a to i v Pythonu 3.1). Naštěstí existuje ještě hezčí způsob.

Pojmenované n-tice mají několik soukromých metod, což jsou metody, jejichž název začíná podtržítkem. Jedna z nich (`namedtuple._asdict()`) je tak užitečná, že si ji ukážeme v akci.*

```
"{manufacturer} {model}".format(**aircraft._asdict())
```

Soukromá metoda `namedtuple._asdict()` vrací mapování dvojic klíč-hodnota, kde každý klíč je název elementu n-tice a každá hodnota je odpovídající hodnotou. Pomocí rozbalování mapování jsme převáděli mapování do argumentů ve tvaru klíč-hodnotu pro metodu `str.format()`.

Použití metody `str.format()` s rozbalením mapování ➢ 86

I když pojmenované n-tice mohou být velmi užitečné, v lekci se seznámíme s objektově orientovaným programováním, posuneme se dál od jednoduchých pojmenovaných n-tic a naučíme se, jak vytvářet vlastní datové typy, které uchovávají datové prvky a které mají také své vlastní, námi definované metody.

Seznamy

Seznam je uspořádaná posloupnost nula nebo více odkazů na objekty. Seznamy podporují stejnou syntaxi pro řezání a krokování jako řetězce a n-tice, díky které lze snadno extrahovat prvky ze seznamu. Na rozdíl od řetězců a n-tic jsou seznamy měnitelné, takže můžeme nahrazovat a mazat kterýkoli z jejich prvků. Kromě toho je možné vkládat, nahrazovat a mazat řezy v seznamu.

Řezání a krokování řetězců ➢ 74

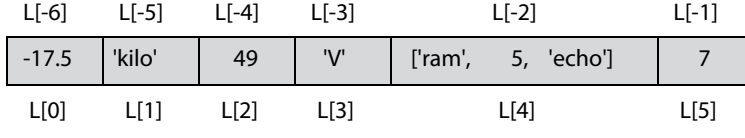
Datový typ `list` lze zavolat i jako funkci `list()`. Bez argumentů vrátí prázdný seznam, s argumentem typu `list` vrátí jeho mělkou kopii a argument jakéhokoliv jiného typu se pokusí převést na seznam. Funkci `list()` nelze předat více než jeden argument. Seznamy je možné vytvářet i bez funkce `list()`. Prázdný seznam vytvoříme pomocí prázdných hranatých závorek `[]` a seznam s jedním nebo více prvky vytvoříme pomocí posloupnosti prvků oddělených čárkou uvnitř hranatých závorek. Další možnost, jak vytvořit seznam, spočívá v použití seznamové komprehenze, což je téma, k němuž se dostaneme za chvíli.

Mělké a hloubkové kopírování ➢ 146

Seznamové komprehenze ➢ 120

Všechny prvky v seznamu jsou skutečně odkazy na objekty, a proto mohou seznamy, stejně jako n-tice, uchovávat prvky libovolného datového typu, včetně typů představujících kolekce, jako jsou seznamy a n-tice. Seznamy lze porovnávat pomocí standardních porovnávacích operátorů (<, <=, ==, !=, >=, >), přičemž porovnávání se aplikuje na jednotlivé prvky (a rekurzivně pro vnořené prvky, jako jsou seznamy nebo n-tice uvnitř seznamů).

Po přiřazení `L = [-17.5, "kilo", 49, "V", ["ram", 5, "echo"], 7]` obdržíme seznam znázorněný na obrázku 3.2.



Obrázek 3.2 Indexové pozice v seznamu

* U soukromých metod, jako je `namedtuple._asdict()`, není zaručeno, že budou k dispozici ve všech verzích Pythonu 3.x, i když metoda namedtuple._asdict() je dostupná v obou verzích Pythonu 3.0 a 3.1.

Zůstaneme u seznamu `L`, na který můžeme použít řezací operátor (klidně i opakovaně) pro přístup k prvkům v seznamu, jak ukazují následující rovnosti:

```
L[0] == L[-6] == -17.5
L[1] == L[-5] == 'kilo'
L[1][0] == L[-5][0] == 'k'
L[4][2] == L[4][-1] == L[-2][2] == L[-2][-1] == 'echo'
L[4][2][1] == L[4][2][-3] == L[-2][-1][1] == L[-2][-1][-3] == 'c'
```

Seznamy lze vnořovat, procházet a řezat, stejně jako n-tice. Všechny příklady s n-ticemi, které jsme si v předchozí části ukázali, by měly ve skutečnosti pracovat naprosto stejně, pokud bychom místo n-tic použili seznamy. Seznamy podporují testování příslušnosti pomocí operátorů `in` a `not int`, spojování s operátorem +, rozšiřování s operátorem += (např. připojení všech prvků v pravém operandu) a replikaci s operátorem `*` a `*=`. Seznamy lze použít také s vestavěnou funkcí `len()` a s příkazem `del`, který si popíšeme v panelu „Mazání prvků pomocí příkazu del“ (strana 117). Kromě toho seznamy nabízejí metody uvedené v tabulce 3.1.

Tabulka 3.1 Metody seznamu

Syntaxe	Popis
L.append(x)	Připojí prvek x na konec seznamu L.
L.count(x)	Vrátí počet výskytů prvku x v seznamu L.
L.extend(m) L += m	Připojí všechny prvky iterovatelného objektu m na konec seznamu L. Totéž provádí operátor +=.
L.index(x, *začátek*, *konec*)	Vrátí indexovou pozici nejlevějšího výskytu prvku x v seznamu L (nebo v řezu *začátek*:*konec* seznamu L). V opačném případě vrátí výjimku `ValueError`.
L.insert(i, x)	Vloží prvek x do seznamu L na indexovou pozici i.
L.pop()	Vrátí a odstraní nejpravější prvek seznamu L.
L.pop(i)	Vrátí a odstraní prvek na indexové pozici i v seznamu L.
L.remove(x)	Odstraní nejlevější výskyt prvku x ze seznamu L nebo vyvolá výjimku `ValueError`, pokud prvek x nenalezne.
L.reverse()	Obrátí seznam L na místě.
L.sort(...)	Seřadí seznam L na místě. Tato metoda přijímá stejné volitelné argumenty *klíč* a *obrátit* jako vestavěná metoda `sorted()`.

`sorted()` ➢ 138, 144

Mazání prvků pomocí příkazu del

I když název příkazu `del` připomíná slovo „delete“ (vymazat), nemusí ve skutečnosti dojít k vymazání žádných dat. Při aplikaci na odkaz na objekt, jenž ukazuje na datový prvek, který není kolekcí, příkaz `del` odváže odkaz na objekt od tohoto datového prvku a tento odkaz vymaže:

```
>>> x = 8143   # vytvoří odkaz na obj. 'x'; vytvoří obj. int s hodnotou 8143
>>> x
8143
>>> del x      # vymaže odkaz na obj. 'x'; obj. int je připraven na úklid
>>> x
Traceback (most recent call last):
...
NameError: name 'x' is not defined
```

Při vymazání odkazu na objekt Python naplánuje, aby byl datový prvek, na který daný odkaz ukazuje, uklizen z paměti, pokud na tento datový prvek již žádný další prvek neodkazuje. Okamžik, kdy dojde k úklidu paměti, pokud k němu vůbec dojde, může být nepředvídatelný (v závislosti na implementaci Pythonu), takže pokud je vyčištění paměti vyžadováno, musíme jej zařídit sami. Python nabízí dvě řešení tohoto nedeterminizmu. Jeden spočívá v použití bloku `try ... finally`, který zajistí, že se vyčištění provedete, a druhý používá příkaz `with`, o kterém se dozvíme v lekci 8.

Když příkaz `del` použijeme na datový typ představující kolekci, jako je n-tice nebo seznam, vymaže se pouze odkaz na kolekci. Kolekce a její prvky (a u prvků, které jsou také kolekcemi, také rekurzivně jejich prvky) se naplánují na uklizení z paměti, pokud na kolekci neukazuje žádný jiný odkaz na objekt.

U měnitelných kolekcí, jako jsou seznamy, lze příkaz `del` aplikovat na jednotlivé prvky nebo řezy, přičemž v obou případech se používá řezací operátor `[]`. Pokud tento prvek nebo prvky odebereme z kolekce a pokud na ně již žádné další odkazy na objekty neukazují, naplánuje se jejich úklid z paměti.

I když můžeme pro přístup k prvkům v seznamu použít řezací operátor, v některých situacích potřebujeme dvě nebo i více částí seznamu najednou. To lze provést rozbalením posloupnosti. Libovolný iterovatelný objekt (seznam, n-tice atd.) můžeme rozbalit pomocí operátoru rozbalení posloupnosti, kterým je asterisk neboli hvězdička (`*`). Při použití s dvěma či více proměnnými na levé straně přiřazení, z nichž jedna má před sebou uveden operátor `*`, se do těchto proměnných přiřadí prvky, přičemž se všechny zbývající přiřadí do proměnné s hvězdičkou:

```
>>> first, *rest = [9, 2, -4, 8, 7]
>>> first, rest
(9, [2, -4, 8, 7])
>>> first, *mid, last = "Karel Filip Artur Jiří Poděbrad".split()
>>> first, mid, last
('Karel', ['Filip', 'Artur', 'Jiří'], 'Poděbrad')
```

```
>>> *directories, executable = "/usr/local/bin/gvim".split("/")
>>> directories, executable
(['', 'usr', 'local', 'bin'], 'gvim')
```

Když se operátor rozbalení posloupnosti použije tímto způsobem, označuje se výraz *rest a podobné výrazy jako *hvězdičkové výrazy* (starred expressions).

Python nabízí další, související koncepci s názvem *hvězdičkové argumenty* (starred arguments). Podívejme se například na následující funkci, která vyžaduje tři argumenty:

```
def product(a, b, c):
    return a * b * c    # * zde představuje operátor násobení
```

Tuto funkci můžeme zavolat buď se třemi argumenty, nebo s použitím hvězdičkových argumentů:

```
>>> product(2, 3, 5)
30
>>> L = [2, 3, 5]
>>> product(*L)
30
>>> product(2, *L[1:])
30
```

V prvním volání předáváme funkci běžným způsobem tři argumenty. Ve druhém volání používáme hvězdičkový argument. V tomto případě dochází k tomu, že se seznam se třemi prvky operátorem `*` rozbalí, takže funkce přijme očekávané tři argumenty. Stejného výsledku bychom dosáhli také při použití n-tice se třemi prvky. A ve třetím volání předáváme první argument normálně a další dva rozbalením dvouprvkového řezu seznamem `L`. Funkce a předávání argumentů budeme podrobně probírat v lekci 4.

V souvislosti s tím, zda je `*` operátorem násobení nebo operátorem rozbalení posloupnosti, neexistuje žádná syntaktická nejednoznačnost. Když se objeví na levé straně přiřazení, jde o operátor rozbalení, a když se objeví kdekoliv jinde (např. při volání funkce), pak jde buď o operátor rozbalení, je-li použit jako unární operátor, nebo o operátor násobení, pokud je použit jako binární operátor.

`range()` ➤ 141

Viděli jsme, že můžeme procházet prvky v seznamu pomocí syntaxe `for item in L:`. Pokud chceme prvky v seznamu měnit, použijeme následující tvar:

```
for i in range(len(L)):
    L[i] = process(L[i])
```

Vestavěná funkce `range()` vrací iterátor, který poskytuje celá čísla. S jedním číselným argumentem n vrátí posloupnost `0, 1, ..., n - 1`.

Tuto techniku bychom mohli použít k inkrementaci všech čísel v seznamu celých čísel:

```
for i in range(len(numbers)):
    numbers[i] += 1
```

Seznamy podporují řezání, a proto lze v několika případech dosáhnout stejného efektu buď pomocí řezání, nebo pomocí metod typu seznamu. Máme-li například seznam `woods = ["Cedr", "Tis", "Borovice"]`, můžeme jej rozšířit kterýmkoli z následujících dvou způsobů:

```
woods += ["Damaroň", "Modřín"]
```

```
woods.extend(["Damaroň", "Modřín"])
```

V obou případech je výsledkem seznam `['Cedr', 'Tis', 'Borovice', 'Damaroň', 'Modřín']`.

Jednotlivé prvky lze přidávat na konec seznamu pomocí metody `list.append()`. Prvky můžeme vkládat na libovolnou pozici v seznamu metodou `list.insert()` nebo přiřazením řezu délky 0. Máme-li například seznam `woods = ["Cedr", "Tis", "Borovice", "Jedle"]`, můžeme vložit nový prvek na indexovou pozici 2 (tj. jako třetí prvek seznamu) jedním ze dvou následujících způsobů:

V obou případech je výsledkem seznam `['Cedr', 'Tis', 'Smrk', 'Borovice', 'Jedle']`.

```
woods[2:2] = ["Smrk"]
```

```
woods.insert(2, "Smrk")
```

Jednotlivé prvky můžeme v seznamu nahrazovat přiřazením do určité indexové pozice, například `woods[2] = "Sekvoj"`. Celé řezy lze nahrazovat přiřazením iterovatelného objektu do daného řezu, například `woods[1:3] = ["Jedle", "Japonský cedr", "Borovice smolná"]`. Řez a iterovatelný objekt nemusejí mít stejnou délku. V každém případě se prvky řezu odstraní a vloží se prvky iterovatelného objektu. Tím může dojít ke zmenšení seznamu, má-li iterovatelný objekt méně prvků než řez, který nahrazuje, nebo k jeho zvětšení, má-li iterovatelný objekt více prvků než daný řez.

Aby bylo skutečně jasné, jak probíhá přiřazení iterovatelného objektu zvolenému řezu, podíváme se ještě na jeden příklad. Představte si, že máme seznam `L = ["A", "B", "C", "D", "E", "F"]` a že přiřadíme iterovatelný objekt (v tomto případě seznam) jistému řezu pomocí kódu `L[2:5] = ["X", "Y"]`. Nejdříve se tento řez odstraní, takže na pozadí se ze seznamu stane `['A', 'B', 'F']`. A poté se všechny iterovatelné prvky vloží na počáteční pozici řezu, takže výsledkem je seznam `['A', 'B', 'X', 'Y', 'F']`.

Prvky můžeme odstranit i několika dalšími způsoby. Pomocí metody `list.pop()` zavolané bez argumentů můžeme odstranit nejpravější prvek v seznamu, který nám navíc metoda vrátí. Podobně můžeme pomocí metody `list.pop()` zavolané s celočíselným indexem odstranit (a získat) prvek na zadané indexové pozici. Další možností je zavolat metodu `list.remove()`, které předáme prvek, který se má odstranit. Jednotlivé prvky nebo řezy prvků můžeme odstraňovat též příkazem `del` (např. `del woods[4]`). Řezy lze také odstraňovat tak, že jim přiřadíme prázdný seznam, takže následující dva úryvky jsou ekvivalentní:

```
woods[2:4] = []
```

```
del woods[2:4]
```

V úryvku na levé straně jsme řezu přiřadili iterovatelný objekt (prázdný seznam), takže se řez nejdříve odstraní, a protože je vkládaný iterovatelný objekt prázdný, k žádnému vložení nedojde.

Když jsme poprvé probírali řezání a krokování, jednalo se o řetězce, u nichž není krokování moc zajímavé. Avšak v případě seznamů nám krokování umožňuje přistupovat ke každému n-tému prvku, což může být často užitečné. Představte si například, že máme seznam `x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` a že chceme nastavit všechny prvky na lichých pozicích (tj. `x[1]`, `x[3]` atd.) na hodnotu `0`. Ke každému druhému prvku můžeme přistupovat pomocí krokování, například `x[::2]`. Tím ale obdržíme prvky na pozicích 0, 2, 4 a tak dále. To můžeme napravit zadáním výchozího začátečních-

Řezání a krokování řetězců ➤ 74

ho indexu, takže nyní máme `x[1::2]`, čímž obdržíme řez s prvky, které požadujeme. Pro nastavení každého prvku v řezu na hodnotu 0 potřebujeme seznam nul, kterých musí být naprosto stejný počet, jako je prvků v řezu.

Zde je kompletní řešení: `x[1::2] = [0] * len(x[1::2])`. Nyní seznam `x` obsahuje `[1, 0, 3, 0, 5, 0, 7, 0, 9, 0]`. K vytvoření seznamu s potřebným počtem nul podle délky (tj. počtu prvků) řezu jsme použili operátor replikace (`*`). Zajímavé je, že při přiřazení seznamu `[0, 0, 0, 0, 0]` do krokovaného řezu Python správně nahradí hodnotu `x[1]` první nulou, hodnou `x[3]` druhou nulou a tak pořád dál.

sorted() ➤ 138

Seznamy lze obracet a seřazovat stejným způsobem jako kterýkoliv jiný iterovatelný objekt pomocí vestavěných funkcí `reversed()` a `sorted()`, které budeme probírat v části „Operace a funkce pro iterovatelné objekty“ (strana 138). Také seznamy nabízejí ekvivalentní metody `list.reverse()` a `list.sort()`, které pracují místně nad příslušným seznamem (takže nic nevracejí) s tím, že metoda `list.sort()` přijímá stejné volitelné argumenty jako funkce `sorted()`. Často potřebujeme seřadit seznam řetězců bez ohledu na velikost písmen. Například seznam `woods` bychom mohli seřadit takto: `woods.sort(key=str.lower)`. Pomocí argumentu `key` stanovíme funkci, která se má aplikovat na každý prvek a jejíž návratová hodnota se použije k porovnávání při seřazování. Jak jsme si řekli již v předchozí lekci věnované porovnávání řetězců (strana 73), u jazyků jiných než angličtina může být porovnávání řetězců způsobem, který je smysluplný pro člověka, docela náročné.

Co se vkládání prvků týče, seznamy si vedou nejlépe, když se prvky přidávají nebo odstraňují na konci (`list.append()`, `list.pop()`). K nejhoršímu výkonu dochází tehdy, když v seznamu hledáme prvky, například pomocí metody `list.remove()` či `list.index()`, nebo když používáme operátor `in` pro testování příslušnosti. Pokud požadujeme rychlé vyhledávání nebo testování příslušnosti, pak může být vhodnější sáhnout po množině (typ `set`) nebo slovníku (typ `dict`), které budeme probírat v pozdější části této lekce. Seznamy mohou také nabídnout rychlé hledání, pokud je budeme udržovat seřazené. Řadicí algoritmus Pythonu je obzvláště dobře optimalizován pro řazení částečně seřazených seznamů, přičemž k hledání prvků používá binární vyhledávání (poskytované modulem `bisect`). (V lekci 6 vytvoříme vlastní třídu reprezentující skutečně seřazený seznam.)

Seznamové komprehenze

Malé seznamy se často vytvářejí pomocí seznamových literálů, ale delší seznamy se obvykle vytvářejí kódem programu. Pro seznam celých čísel můžeme použít funkci `list(range(n))` nebo jen funkci `range()`, pokud nám stačí iterátor celých čísel. U seznamů jiných typů se nejčastěji používá cyklus `for... in`. Představte si, že chceme například vytvořit seznam přestupných roků v zadaném rozmezí. Kód můžeme začít takto:

range() ➤ 141

```
leaps = []
for year in range(1900, 1940):
if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
    leaps.append(year)
```

Když funkci `range()` předáme dva celočíselné argumenty n a m, vytvoří takto získaný iterátor celá čísla `n, n + 1, ..., m - 1`.

Pokud bychom znali přesný rozsah předem, mohli bychom samozřejmě použít seznamový literál, například `leaps = [1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936]`.

Seznamová komprehenze je výraz a cyklus s volitelnou podmínkou uzavřený do hranatých závorek, přičemž cyklus se používá ke generování prvků pro seznam a podmínka může odfiltrovat nechtěné prvky. Nejjednodušší tvar seznamové komprehenze vypadá takto:

```
[prvek for prvek in iterovatelný_objekt]
```

Tento výraz vrací seznam všech prvků v iterovatelném objektu a je sémanticky shodný s výrazem `list(iterovatelný_objekt)`. Seznamové komprehenze činí zajímavějšími a výkonnějšími to, že můžeme použít výrazy a připojit podmínku. Tím se dostáváme ke dvěma obecným syntaxím pro seznamové komprehenze:

```
[výraz for prvek in iterovatelný_objekt]
[výraz for prvek in iterovatelný_objekt if podmínka]
```

Druhá syntaxe je ekvivalentní zápisu:

```
temp = []
for prvek in iterovatelný_objekt:
    if podmínka:
        temp.append(výraz)
```

Výraz v této syntaxi často obsahuje uvedený `prvek`. Seznamová komprehenze samozřejmě na rozdíl od verze s cyklem `for ... in` žádnou proměnnou `temp` nepotřebuje.

Nyní můžeme přepsat kód pro generování přestupných roků pomocí seznamové komprehenze. Kód napíšeme postupně, ve třech krocích. Nejdříve budeme generovat seznam, který má všechny roky v zadaném rozsahu:

```
leaps = [y for y in range(1900, 1940)]
```

Stejného výsledku bychom dosáhli také pomocí příkazu `leaps = list(range(1900, 1940))`. Nyní potřebujeme přidat jednoduchou podmínku, abychom získali jen každý čtvrtý rok:

```
leaps = [y for y in range(1900, 1940) if y % 4 == 0]
```

Nakonec můžeme zapsat kompletní verzi:

```
leaps = [y for y in range(1900, 1940)
         if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]
```

Pomocí seznamové komprehenze jsme v tomto případě zredukovali kód ze čtyř na dva řádky, což je sice malá úspora, která však může být ve velkých projektech podstatným přínosem.

Vzhledem k tomu, že seznamové komprehenze vytvářejí seznamy neboli iterovatelné objekty, a vzhledem k tomu, že syntaxe pro seznamové komprehenze vyžaduje iterovatelný objekt, můžeme seznamové komprehenze vnořovat. Jedná se o ekvivalenci vnořování cyklů `for... in`. Pokud například chceme vygenerovat všechna možná označení oblečení pro zadané skupiny pohlaví, velikostí a barev,

ale bez označení pro plnoštíhlé ženy, které módní průmysl neustále ignoruje, mohli bychom použít následující cykly `for... in`:

```
codes = []
for sex in "MF":            # Male (muž), Female (žena)
    for size in "SMLX":     # Small, Medium, Large, eXtra large
        if sex == "F" and size == "X":
            continue
        for color in "BGW": # Black (černá), Gray (šedá), White (bílá)
            codes.append(sex + size + color)
```

Tento kód vytvoří seznam s 21 prvky (`['MSB', 'MSG', ..., 'FLW']`). Stejného výsledku můžeme docílit pomocí seznamové komprehenze jen na dvou řádcích:

```
codes = [s + z + c for s in "MF" for z in "SMLX" for c in "BGW"
         if not (s == "F" and z == "X")]
```

Zde každý prvek v seznamu vytváříme výrazem `s + z + c`. Kromě toho jsme použili malinko odlišnou logiku pro seznamovou komprehenzi, v níž neplatnou kombinaci pohlaví a velikosti přeskakujeme v nejvnitřnějším cyklu, zatímco vnořené verze cyklů `for...in` přeskakují neplatné kombinace ve svém prostředním cyklu. Jakoukoliv seznamovou komprehenzi můžeme přepsat pomocí jednoho či více cyklů `for...in`.

Generátory ➢ 332

Je-li generovaný seznam velmi rozsáhlý, může být efektivnější místo vytváření celého seznamu najednou generovat každý prvek až v okamžiku, kdy jej potřebujeme. Toho nelze dosáhnout pomocí seznamové komprehenze, ale pomocí generátorů, kterým se budeme věnovat v lekci 8.

Množinové typy

Množinový typ je datový typ představující kolekci, který podporuje operátor příslušnosti (`in`) funkci zjišťující velikost (`len()`) a který je iterovatelný. Množinové typy dále nabízejí metodu `set.isdisjoint()` a podporují porovnávání a také bitové operátory (které se v kontextu množin používají pro sjednocení, průnik atd.). Jazyk Python nabízí dva vestavěné množinové typy: měnitelný typ `set` (množina) a neměnitelný typ `frozenset` (zmrazená množina). Při průchodu množinou procházíme prvky v nahodilém pořadí.

Do množiny lze přidávat pouze *hashovatelné* objekty. Hashovatelné objekty jsou takové objekty, které mají speciální metodu `__hash__()`, jejíž návratová hodnota je v průběhu života objektu vždy stejná a kterou lze porovnávat na rovnost pomocí speciální metody `__eq__()`. (Speciální metody, tj. metody, jejichž název začíná a končí dvěma podtržítky, budeme probírat v lekci 6.)

Všechny vestavěné neměnitelné datové typy, jako je `float`, `frozenset`, `int`, `str` a `tuple`, jsou hashovatelné, a lze je tedy přidávat do množin. Vestavěné měnitelné datové typy, jako je `dict`, `list` a `set`, nejsou hashovatelné, protože jejich hashová hodnota se mění v závislosti na prvcích, které obsahují, a proto je nelze přidávat do množin.

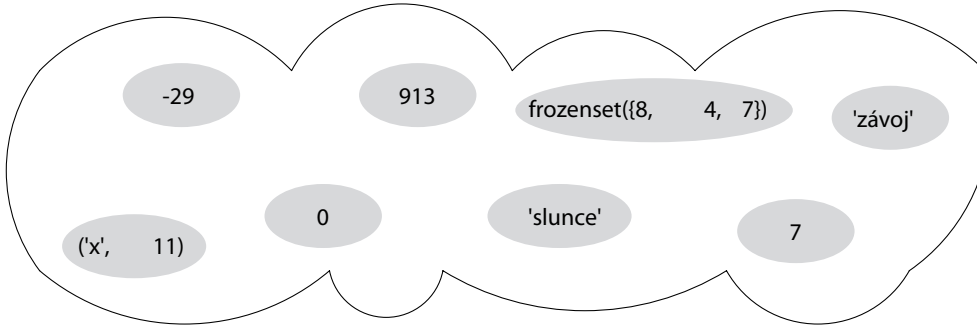
Množinové typy lze porovnávat pomocí standardních porovnávacích operátorů (<, <=, ==, !=, >=, >). Všimněte si, že ačkoliv operátory == a =! mají svůj obvyklý význam, kdy se porovnávání aplikuje po jednotlivých prvcích

(a rekurzivně pro vnořené prvky, jako jsou n-tice nebo zmrazené množiny uvnitř množin), ostatní porovnávací operátory provádějí podmnožinová a nadmnožinová porovnání, o čemž se za okamžik přesvědčíme.

Množiny

Množina je neuspořádaná kolekce nula či více odkazů na objekty, které ukazují na hashovatelné objekty. Množiny jsou měnitelné, a proto můžeme snadno přidávat nebo odebírat prvky. Jsou však také neuspořádané, takže nevědí, co je indexová pozice, a proto je nelze řezat ani krokovat. Obrázek 3.3 znázorňuje množinu vytvořenou následujícím úryvkem kódu:

```
S = {7, "závoj", 0, -29, ("x", 11), "slunce", frozenset({8, 4, 7}), 913}
```



Obrázek 3.3: Množina je neuspořádaná kolekce jedinečných prvků

Datový typ `set` lze zavolat i jako funkci `set()`. Bez argumentů vrátí prázdnou množinu, s argumentem typu `set` vrátí jeho mělkou kopii a argument jakéhokoliv jiného typu se pokusí převést na množinu. Funkci `set()` nelze předat více než jeden argument. Neprázdné množiny je možné vytvářet i bez funkce `set()`, ale prázdná množina musí být vytvořená pomocí funkce `set()`, a ne pomocí prázdných složených závorek.* Množinu s jedním nebo více prvky vytvoříme pomocí posloupnosti prvků oddělených čárkou uvnitř složených závorek. Další možnost, jak vytvořit množinu, spočívá v použití množinové komprehenze, což je téma, k němuž se dostaneme za okamžik.

Mělké a hloubkové kopírování ➤ 146

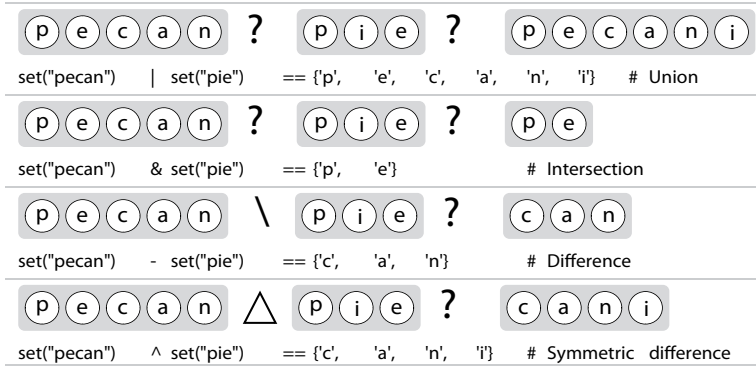
Množinové komprehenze ➤ 126

Množiny vždy obsahují jedinečné prvky. Přidání duplikátních prvků je sice bezpečné, ale bezúčelné. Kupříkladu tyto tři množiny jsou stejné: `set("apple")`, `set("aple")` a `{'e', 'p', 'a', 'l'}`. Vzhledem k této skutečnosti se množiny často používají k eliminaci duplicit. Pokud je například `x` seznam řetězců, budou po provedení příkazu `x = list(set(x))` všechny řetězce v `x` jedinečné (a v nahodilém pořadí).

Množiny podporují vestavěnou funkci `len()` a rychlé testování příslušnosti pomocí operátorů `in` a `not in`. Kromě toho nabízejí obvyklé množinové operátory, které jsou znázorněny na obrázku 3.4.

```
set("pecan") | set("pie") == {'p', 'e', 'c', 'a', 'n', 'i'} # Sjednocení
set("pecan") & set("pie") == {'p', 'e'}                     # Průnik
set("pecan") - set("pie") == {'c', 'a', 'n'}                # Rozdíl
set("pecan") ^ set("pie") == {'c', 'a', 'n', 'i'}           # Symetrický rozdíl
```

* Prázdné složené závorky ({}) se používají pro vytváření prázdného slovníku, kterému se budeme věnovat v následující části.



Obrázek 3.4: Standardní množinové operátory

Úplný seznam množinových metod a operátorů uvádí tabulka 3.2. Všechny „aktualizační" metody (`set.update()`, `set.intersection_update()` atd.) přijímají jako svůj argument libovolný iterovatelný objekt, avšak ekvivalentní operátory (`|=`, `&=` atd.) vyžadují, aby oba operandy byly množiny.

Tabulka 3.2: Seznam množinových metod a operátorů

Syntaxe	Popis
s.add(x)	Přidá prvek x do množiny s, pokud dosud v množině s není.
s.clear()	Odstraní všechny prvky z množiny s.
s.copy()	Vrátí mělkou kopii množiny s*.
s.difference(t) s – t	Vrátí novou množinu, jež má všechny prvky, které jsou v množině s a nejsou v množině t*.
s.difference_ update(t) s -= t	Odstraní z množiny s všechny prvky, které jsou v množině t.
s.discard(x)	Odstraní prvek x z množiny s, pokud leží v množině s (viz též metoda `set.remove()`).
s.intersection(t) s & t	Vrátí novou množinu, jež má všechny prvky, kterou jsou současně v obou množinách s a t*.
s.intersection_ update(t) s &= t	Upraví množinu s tak, aby obsahovala průnik s množinou t.
s.isdisjoint(t)	Vrátí hodnotu `True`, pokud množiny s a t nemají žádný společný prvek*.
s.issubset(t) s <= t	Vrátí hodnotu `True`, pokud je množina s ekvivalentní množině t nebo pokud je její podmnožinou. K otestování, zda je s vlastní podmnožinou množiny t se používá s < t*.
s.issuperset(t) s >= t	Vrátí hodnotu `True`, pokud je množina s ekvivalentní množině t nebo pokud je její nadmnožinou. K otestování, zda je s vlastní nadmnožinou množiny t, se používá s < t*.
s.pop()	Odstraní a vrátí náhodný prvek z množiny s nebo vyvolá výjimku `KeyError`, je-li s prázdná.

Mělké a hloubkové kopírování ➤ 146

Syntaxe	Popis
s.remove(x)	Odstraní prvek x z množiny s nebo vyvolá výjimku KeyError, není-li x v množině s (viz též metoda set.discard()).
s.symmetric_difference(t) s ^ t	Vrátí novou množinu, jež obsahuje všechny prvky, které jsou v množině s, a všechny prvky, které jsou v množině t, ale bez těch prvků, které jsou v obou množinách*.
s.symmetric_difference_update(t) s ^= t	Upraví množinu s tak, aby obsahovala symetrický rozdíl sebe a množiny t.
s.union(t) s | t	Vrátí novou množinu, jež obsahuje všechny prvky v množině s a všechny prvky v množině t, které nejsou v množině s*.
s.update(t) s |= t	Přidá všechny prvky v množině t, které nejsou v množině s, do množiny s.

Množiny se běžně používají v situaci, kdy potřebujeme rychle otestovat příslušnost. Můžeme například chtít vypsat uživateli zprávu s informacemi o použití, pokud na příkazovém řádku nezadá žádný z argumentů nebo pokud zadá argument „-h“ nebo „--help“:

```
if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
```

Dalším běžným použitím pro množiny je zajištění, aby se nezpracovávala duplicitní data. Představte si, že máme nějaký iterovatelný objekt (např. seznam) obsahující IP adresy ze souborů protokolu webového serveru a že chceme nad každou jedinečnou adresou provést určité zpracování. Za předpokladu, že IP adresy jsou hashovatelné a jsou v iterovatelné proměnné ips a že máme definovanou funkci, kterou chceme volat pro každou adresu a která se jmenuje process_ip(), provedou následující úryvky kódu to, co potřebujeme, i když s malinko odlišným chováním:

```
seen = set()
for ip in ips:
    if ip not in seen:
        seen.add(ip)
        process_ip(ip)
```

```
for ip in set(ips):
    process_ip(ip)
```

V úryvku na levé straně přidáváme IP adresu do množiny seen a zpracováváme ji jen tehdy, pokud jsme ji nezpracovali již dříve. V opačném případě ji ignorujeme. V úryvku na pravé straně získáváme pouze jedinečné IP adresy, které ihned zpracováváme. Rozdíl je jednak v tom, že v úryvku na levé straně vytváříme množinu seen, kterou na pravé straně nepotřebujeme, a pak také v tom, že na levé straně zpracováváme IP adresy v pořadí, ve kterém je obdržíme z iterovatelné proměnné ips, kdežto na pravé straně je zpracováváme v nahodilém pořadí.

Kód na pravé straně se snadněji píše, pokud je však uspořádání prvků v iterovatelné proměnné ips důležité, pak musíme buď použít postup na levé straně, nebo změnit první řádek pravé strany na něco, jako je sorted(set(ips)), je-li to dostatečné pro získání požadovaného uspořádání. Teoreticky může být postup na pravé straně pomalejší, je-li počet prvků v ips velmi velký, protože zde nevytváříme množinu inkrementálně, ale naráz.

* Tuto metodu a její operátor (má-li nějaký) lze použít také u množin typu frozenset.

Množiny se používají také k eliminaci nechtěných prvků. Máme-li například seznam názvů souborů a přitom nechceme žádné soubory typu `makefile` (třeba proto, že nejsou psány ručně, ale jsou generované), pak můžeme napsat:

```
filenames = set(filenames)
for makefile in {"MAKEFILE", "Makefile", "makefile"}:
    filenames.discard(makefile)
```

Tento kód odstraní všechny soubory `makefile`, které jsou v seznamu se standardně používanými velikostmi písmen. Pokud v seznamu `filenames` žádné soubory `makefile` nejsou, nestane se vůbec nic. Stejného výsledku docílíme na jednom řádku pomocí operátoru množinového rozdílu (`-`):

```
filenames = set(filenames) - {"MAKEFILE", "Makefile", "makefile"}
```

Kromě toho můžeme odstraňovat prvky také pomocí metody `set.remove()`, i když tato metoda vyvolá výjimku `KeyError`, pokud prvek, který se má odstranit, v množině není.

Množinové komprehenze

Kromě vytváření množin pomocí funkce `set()` nebo pomocí množinového literálu můžeme využít také *množinové komprehenze*. Množinová komprehenze je výraz a cyklus s volitelnou podmínkou uzavřený do složených závorek. Podporovány jsou podobně jako u seznamových komprehenzí dvě syntaxe:

```
{výraz for prvek in iterovatelný_objekt}
{výraz for prvek in iterovatelný_objekt if podmínka}
```

Tyto syntaxe můžeme použít k realizaci filtrovacího efektu (za předpokladu, že nezáleží na pořadí). Zde je příklad:

```
html = {x for x in files if x.lower().endswith((".htm", ".html"))}
```

Máme-li v seznamu `files` názvy souborů, pak tato množinová komprehenze vytvoří množinu `html` uchovávající pouze ty názvy souborů, které končí příponou `.htm` nebo `.html` bez ohledu na velikost písmen.

Podobně jako u seznamových komprehenzí mohou také samotné iterovatelné objekty použité v množinové komprehenzi být množinovou komprehenzí (nebo jakýmkoli jiným druhem komprehenze), díky čemuž lze vytvářet docela sofistikované množinové komprehenze.

Zmrazené množiny

Mělké a hloubkové kopírování ➤ 146

Zmrazená množina je množina, kterou nelze po jejím vytvoření měnit. Proměnnou ukazující na zmrazenou množinu samozřejmě můžeme opětovně svázat s něčím jiným. Zmrazené množiny lze vytvářet jen pomocí datového typu `frozenset` zavolaného jako funkce. Bez argumentů vrátí prázdnou zmrazenou množinu, s argumentem typu `frozenset` vrátí jeho mělkou kopii a argument jakéhokoliv jiného typu se pokusí převést na zmrazenou množinu. Funkci `frozenset()` nelze předat více než jeden argument.

Zmrazené množiny jsou neměnitelné, a proto podporují pouze ty metody a operátory, které vytvářejí výsledky bez vlivu na samotnou zmrazenou množinu nebo na množiny, na které se aplikují. Tabulka 3.2 (strana 124) uvádí seznam všech množinových metod, přičemž zmrazené množiny podporují metody `frozenset.copy()`, `frozenset.difference()` (`-`), `frozenset.intersection()` (`&`), `frozenset.isdisjoint()`, `frozenset.issubset()` (`<=` a také `<` pro vlastní podmnožiny), `frozenset.union()` (`|`) a `frozenset.symmetric_difference()` (`^`), které jsme v této tabulce označili hvězdičkou.

Pokud použijeme binární operátor s množinou a zmrazenou množinou, bude datový typ výsledku stejný jako datový typ operandu na levé straně. Takže pokud je `f` zmrazená množina a `s` množina, pak `f & s` vytvoří zmrazenou množinu a `s & f` vytvoří množinu. V případě operátorů == a `!=` na pořadí operandů nezáleží, a proto výraz f == s vrátí hodnotu `True`, obsahují-li obě množiny stejné prvky.

Dalším důsledkem neměnitelnosti zmrazených množin je to, že splňují kritérium hashovatelnosti pro množinové prvky. Množiny a zmrazené množiny tedy mohou obsahovat zmrazené množiny.

Více příkladů s množinami si ukážeme v následující části a také v části s příklady této lekce.

Typy představující mapování

3.x

Typ představující *mapování* je takový typ, který podporuje operátor příslušnosti (`in`) a funkci pro zjištění velikost (`len()`) a který je iterovatelný. Mapování jsou kolekce prvků ve tvaru klíč-hodnota a nabízejí metody pro přístup k prvkům a k jejich klíčům a hodnotám. Při procházení poskytují typy představující neuspořádané mapování své prvky v nahodilém pořadí. Python 3.0 nabízí dva typy představující neuspořádané mapování: vestavěný typ `dict` a typ ze standardní knihovny `collections.defaultdict`. V Pythonu 3.1 byl zaveden nový typ představující uspořádané mapování s názvem `collections.OrderedDict`. Jedná se o slovník, který má stejné metody a vlastnosti (tj. stejné rozhraní API) jako vestavěný typ `dict`, své prvky však uchovává v stejném pořadí, ve kterém byly *vloženy*. * Termínem *slovník* budeme označovat libovolný z těchto typů v situacích, kdy nebude podstatné, o který konkrétní typ se jedná.

Hashovatelné objekty ➢ 122

Jako klíče slovníků lze použít pouze hashovatelné objekty, což zahrnuje všechny neměnitelné datové typy, jako je `float`, `frozenset`, `int`, `str` a `tuple`, a vylučuje všechny měnitelné typy, jako je `dict`, `list` a `set`. Na druhou stranu každá hodnota spojená s klíčem může být odkazem na objekt ukazujícím na objekt libovolného typu, včetně čísel, řetězců, seznamů, množin slovníků, funkcí a tak dále.

Slovníkové typy lze porovnávat pomocí standardních operátorů porovnávajících rovnost (== a `!=`), které se aplikují na jednotlivé prvky (a rekurzivně na vnořené prvky, jako jsou n-tice nebo slovníky uvnitř slovníků). Porovnávání s použitím ostatních porovnávacích operátorů (`<`, `<=`, `>=`, `>`) není podporováno, protože u neuspořádaných kolekcí, jako jsou slovníky, nemají žádný smysl.

* Zkratka API znamená aplikační programovací rozhraní (Application Programming Interface), což je obecný termín používaný pro označení veřejných metod a vlastností poskytovaných třídami a také parametrů a návratových hodnot funkcí a metod. Například dokumentace Pythonu dokumentuje rozhraní API poskytované Pythonem.

Slovníky

Typ `dict` představuje neuspořádanou kolekci nula či více dvojic klíč-hodnota, přičemž klíče jsou odkazy na objekty, které ukazují na hashovatelné objekty, a hodnoty jsou odkazy na objekty ukazující na objekty libovolného typu. Slovníky jsou měnitelné, můžeme u nich tedy snadno přidávat a odebírat prvky. Nejsou však uspořádané, takže nevědí, co je indexová pozice, a proto je nelze řezat ani krokovat.

Mělké a hloubkové kopírování ➤ 146

Datový typ `dict` lze zavolat i jako funkci `dict()`. Bez argumentů vrátí prázdnou množinu a s argumentem typu představujícího mapování vrátí jeho mělkou kopii (např. je-li argument nějaký slovník). Kromě toho je možné použít argument představující posloupnost, ovšem za předpokladu, že každý prvek v posloupnosti je sám posloupností dvou objektů, z nichž první se použije jako klíč a druhý jako jeho hodnota. Dále je možné u slovníků, v nichž jsou klíče platnými identifikátory jazyka Python, použít klíčované argumenty, přičemž pro příslušné klíče se použije hodnota odpovídajícího klíče. Slovníky lze též vytvářet pomocí složených závorek (`{}`). Prázdné složené závorky vytvoří prázdný slovník a neprázdné složené závorky musejí obsahovat jeden nebo více čárkou oddělených prvků, z nichž každý je tvořen klíčem, dvojtečkou a hodnotou. Další možnost, jak vytvořit slovník, spočívá v použití slovníkové komprehenze, což je téma, k němuž se dostaneme za chvíli.

Klíčové argumenty ➤ 173

Slovníkové komprehenze ➤ 134

Zde je několik příkladů ilustrujících nejrůznější syntaxe, přičemž všechny vytvářejí stejný slovník:

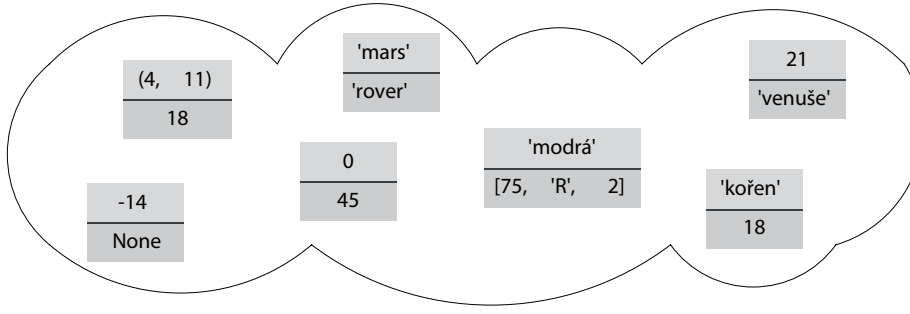
```
d1 = dict({"id": 1948, "name": "Myčka", "size": 3})
d2 = dict(id=1948, name="Myčka", size=3)
d3 = dict([("id", 1948), ("name", "Myčka"), ("size", 3)])
d4 = dict(zip(("id", "name", "size"), (1948, "Myčka", 3)))
d5 = {"id": 1948, "name": "Myčka", "size": 3}
```

zip() ➤ 142

Slovník `d1` jsme vytvořili pomocí slovníkového literálu a slovník `d2` jsme vytvořili pomocí klíčovaných argumentů. Slovníky `d3` a `d4` byly vytvořeny z posloupností a slovník `d5` byl vytvořen ze slovníkového literálu. Vestavěná funkce `zip()`, kterou jsme použili k vytvoření slovníku `d4`, vrací seznam n-tic, z nichž první obsahuje první prvky každého z iterovatelných argumentů funkce, druhý obsahuje druhé prvky a tak dále. Syntaxe s klíčovanými argumenty (použití k vytvoření slovníku `d2`) je obvykle nejkompaktnější a nejpohodlnější, ovšem za předpokladu, že klíče jsou platnými identifikátory.

Obrázek 3.5 znázorňuje slovník vytvořený následujícím úryvkem kódu:

```
d = {"kořen": 18, "modrá": [75, "R", 2], 21: "venuše", -14: None,
     "mars": "rover", (4, 11): 18, 0: 45}
```



Obrázek 3.5: Slovník je neuspořádaná kolekce prvků (klíč, hodnota) s jedinečnými klíči

Klíče slovníku jsou jedinečné, takže pokud přidáme prvek ve tvaru klíč-hodnota, jehož klíč je stejný jako některý ze stávajících klíčů, dojde k nahrazení hodnoty klíče novou hodnotou. Hranaté závorky se používají pro přístup k jednotlivým hodnotám. Například u slovníku na obrázku 3.5 výraz `d["root"]` vrátí `18`, `d[21]` vrátí řetězec `"venuše"` a `d[91]` způsobí vyvolání výjimky `KeyError`.

Hranaté závorky lze použít také k přidávání a mazání prvků slovníku. K přidání prvku se používá operátor = (např. `d["X"] = 59`) a k vymazání prvku se používá příkaz `del` (např. `del d["mars"]` vymaže ze slovníku prvek, jehož klíč má hodnotu „mars“, nebo vyvolá výjimku `KeyError`, pokud žádný z prvků slovníku tento klíč nemá). Prvky můžeme ze slovníku také odstranit (a získat) pomocí metody `dict.pop()`.

Slovníky podporují vestavěnou funkci `len()` a díky svým klíčům nabízejí rychlé testování příslušnosti prostřednictvím operátorů `in` a `not in`. Všechny slovníkové metody uvádí tabulka 3.3.

Tabulka 3.3: Slovníkové metody

Syntaxe	Popis
d.clear()	Odstraní všechny prvky ze slovníku `d`.
d.copy()	Vrátí mělkou kopii slovníku `d`.
d.fromkeys(s, v)	Vrátí slovník typu `dict`, jehož klíče jsou tvořeny prvky z posloupnosti `s` a hodnoty jsou buď `None`, nebo mají hodnotu `v`, je-li tato hodnota zadána.
d.get(k)	Vrátí hodnotu spojenou s klíčem `k` nebo `None`, pokud klíč `k` ve slovníku `d` není.
d.get(k, v)	Vrátí hodnotu spojenou s klíčem `k` nebo hodnotu `v`, pokud klíč `k` ve slovníku `d` není.
d.items()	Vrátí pohled* tvořený všemi dvojicemi (klíč, hodnota) ze slovníku `d`.
d.keys()	Vrátí pohled* tvořený všemi klíči ze slovníku `d`.
d.pop(k)	Vrátí hodnotu spojenou s klíčem `k` a odstraní prvek, jehož klíčem je `k`, nebo vyvolá výjimku `KeyError`, není-li klíč `k` ve slovníku `d`.
d.pop(k, v)	Vrátí hodnotu spojenou s klíčem `k` a odstraní prvek, jehož klíčem je `k`, nebo vrátí hodnotu `v`, není-li klíč `k` ve slovníku `d`.
d.popitem()	Vrátí a odstraní libovolnou dvojici (klíč, hodnota) ze slovníku `d` nebo vyvolá výjimku `KeyError`, je-li slovník `d` prázdný.
d.setdefault(k, v)	Funguje stejně jako metoda `dict.get()`, tedy až na to, že pokud klíč `k` není ve slovníku `d`, vloží se ke klíči `k` nový prvek s hodnotou `None` nebo `v`, je-li `v` zadáno.
d.update(a)	Přidá do slovníku `d` každou dvojici (klíč, hodnota) z `a`, která není ve slovníku `d`, a pro každý klíč, který je v `d` i `a`, nahradí odpovídající hodnotu v `d` hodnotou v `a`. Parametrem `a` může být slovník, iterovatelný objekt s dvojicemi (klíč, hodnota) nebo klíčované argumenty.
d.values()	Vrátí pohled tvořený všemi hodnotami ze slovníku `d`.

Mělké a hloubkové kopírování ➤ 146

Slovníky mají klíče i hodnoty, a proto můžeme slovníky procházet podle prvků (klíč, hodnota), podle hodnot nebo podle klíčů. Zde jsou například dva ekvivalentní postupy k procházení podle dvojic (klíč, hodnota):

* Slovníkové pohledy si lze představit jako iterovatelné objekty (a také se takto používají). Podrobně si je vysvětlíme v dalším textu.

```
for item in d.items():
    print(item[0], item[1])
```

```
for key, value in d.items():
    print(key, value)
```

Procházení hodnot slovníku je velmi podobné:

```
for value in d.values():
    print(value)
```

K procházení klíčů slovníku můžeme použít metodu `dict.keys()` nebo můžeme prostě považovat slovník za iterovatelný objekt, který se prochází přes klíče, což demonstrují následující dva ekvivalentní úryvky kódu:

```
for key in d:
    print(key)
```

```
for key in d.keys():
    print(key)
```

Pro změnu hodnot ve slovníku se obvykle prochází přes klíče slovníku a hodnoty se mění pomocí operátoru s hranatými závorkami. Zde je kupříkladu způsob, jakým bychom mohli inkrementovat každou hodnotu ve slovníku `d`, ovšem za předpokladu, že všechny hodnoty jsou čísla:

```
for key in d:
    d[key] += 1
```

Metody `dict.items()`, `dict.keys()` a `dict.values()` vracejí *slovníkové pohledy*. Slovníkový pohled je v podstatě iterovatelný objekt určený pouze ke čtení, který má uchovávat prvky, klíče nebo hodnoty slovníku v závislosti na pohledu, o který požádáme.

Obecně lze říci, že s pohledy můžeme pracovat jako s iterovatelnými objekty. Nicméně jsou zde dvě věci, které pohledy od běžných iterovatelných objektů odlišují. Je-li totiž slovník, na který se pohled odkazuje, změněn, odrazí se tyto změny také v tomto pohledu. Další věcí je to, že pohledy s klíči a prvky podporují operace podobné množinovým operacím. Máme-li slovníkový pohled `v` a množinu nebo slovníkový pohled `x`, pak můžeme použít následující operace:

```
v & x # Průnik
v | x # Sjednocení
v - x # Rozdíl
v ^ x # Synamtrický rozdíl
```

Pomocí operátoru příslušnosti (`in`) můžeme zjistit, zda je ve slovníku určitý klíč (např. `x in d`), a pomocí operátoru průniku můžeme zjistit, které klíče ze zadané množiny se nacházejí ve slovníku:

```
d = {}.fromkeys("ABCD", 3) # d == {'A': 3, 'B': 3, 'C': 3, 'D': 3}
s = set("ACX")             # s == {'A', 'C', 'X'}
matches = d.keys() & s     # matches == {'A', 'C'}
```

Všimněte si, že v komentářích úryvku kódu jsme použili abecední pořadí, což nemá jiný význam než zlepšit čitelnost, protože slovníky a množiny jsou neuspořádané.

Slovníky se často používají pro uchovávání počtu jedinečných prvků. Jedním z příkladů může být sčítání počtu výskytů každého jedinečného slova v souboru. Zde je kompletní program (`uniquewords1.py`), který

u všech souborů uvedených na příkazovém řádku vypíše v abecedním pořadí všechna v nich obsažená slova a počet jejich výskytů:

```
import string
import sys

words = {}
strip = string.whitespace + string.punctuation + string.digits + "\"'"
for filename in sys.argv[1:]:
    for line in open(filename):
        for word in line.lower().split():
            word = word.strip(strip)
            if len(word) > 2:
                words[word] = words.get(word, 0) + 1
for word in sorted(words):
    print("'{0}' se vyskytuje {1}krát".format(word, words[word]))
```

Na začátku programu vytváříme prázdný slovník s názvem `words`. Poté spojením několika užitečných řetězců z modulu `string` vytváříme řetězec, který obsahuje všechny znaky, které chceme ignorovat. Procházíme všechny názvy souborů zadané na příkazovém řádku a také všechny řádky každého z nich. Vysvětlení funkce open() najdete v panelu „Čtení a zápis textových souborů". Kódování neuvádíme (protože nevíme, které kódování jednotlivé soubory používají), a proto necháme Python, aby každý soubor otevřel s použitím výchozího místního kódování. Každý řádek převedený na malá písmena rozdělíme do slov a poté z obou konců každého slova odstraníme znaky, které chceme ignorovat. Je-li výsledné slovo alespoň tři znaky dlouhé, aktualizujeme náš slovník.

Čtení a zápis textových souborů

Soubory se otevírají pomocí vestavěné funkce `open()`, která vrací objekt představující soubor (pro textové soubory vrací objekt typu `io.TextIOWrapper`). Funkce open() přijímá jeden povinný argument (název souboru, který může obsahovat cestu) a až šest volitelných argumentů. Dva z nich si zde ve stručnosti popíšeme. Druhým argumentem je *režim*, který se používá ke specifikaci, zda se má zadaný soubor považovat za textový nebo binární a zda se má soubor otevřít pro čtení, zápis, připojování nebo kombinaci těchto režimů.

Lekce 7 (Práce se soubory) ➤ 280

Pro textové soubory používá Python kódování, které je závislé na používané platformě. Všude, kde je to možné, je nejvhodnější stanovit kódování pomocí argumentu funkce `open()` s názvem `encoding`. Syntaxe pro běžné otevírání souborů tedy vypadají takto:

Kódování znaků ➤ 95

```
fin = open(filename, encoding="utf8")       # pro čtení textu
fout = open(filename, "w", encoding="utf8") # pro zápis textu
```

Výchozí hodnotou režimu funkce `open()` je „čtení textu", takže pokud pro argument `encoding` použijeme místo pozičního klíčovaný argument, můžeme při otevírání souboru pro čtení vynechat ostatní volitelé poziční argumenty. A podobně při otevírání souboru pro zápis stačí, když zadáme pouze argumenty, které chceme skutečně použít. (Předávání argumentů budeme probírat v lekci 4.)

Jakmile je soubor otevřen pro čtení v textovém režimu, můžeme pomocí metody `read()` objektu souboru celý soubor načíst do jediného řetězce nebo pomocí metody `readlines()` objektu souboru do seznamu řetězců. Pro čtení řádek po řádku se velmi často pracuje s objektem souboru jako s iterátorem:

```
for line in open(filename, encoding="utf8"):
    process(line)
```

Tento zápis funguje díky tomu, že objekt souboru lze procházet stejně jako posloupnost, přičemž každým dalším prvkem je řetězec obsahující následující řádek ze souboru. Řádky, které takto načteme, obsahují ukončovací znak „\n".

Pokud zadáme režim „w", soubor se otevře v režimu „zápisu textu". Do souboru zapisujeme pomocí metody `write()` objektu souboru, která jako svůj argument přijímá jediný řetězec. Každý zapsaný řádek by měl končit znakem „\n". Python při čtení a zápisu automaticky převádí znak „\n" na ukončovací znak dané platformy a zpět.

Jakmile dokončíme práci s objektem souboru, můžeme zavolat jeho metodu `close()`, která způsobí, že se jakýkoli nedokončený zápis dokončí. Malé programy napsané v Pythonu se často voláním metody `close()` neobtěžují, protože Python ji volá automaticky v okamžiku, kdy objekt souboru opustí svůj obor platnosti. Dojde-li k nějakému problému, projeví se vyvoláním výjimky.

Syntaxi `words[word] += 1` použít nemůžeme, protože by při první výskytu nového slova došlo k vyvolání výjimky `KeyError`. Koneckonců nelze přece inkrementovat hodnotu prvku, který ve slovníku zatím neexistuje. Používáme tedy promyšlenější přístup. Voláme metodu `dict.get()` s výchozí hodnotou `0`. Je-li dané slovo již ve slovníku, vrátí metoda `dict.get()` s ním spojené číslo, k němuž přičteme 1 a nastavíme je jako novou hodnotu tohoto prvku. Pokud dané slovo ve slovníku není, metoda `dict.get()` vrátí zadanou výchozí hodnotu `0`, k níž přičteme jedničku a nastavíme jako hodnotu nového prvku, jehož klíčem je řetězec uložený v proměnné `word`. Pro lepší pochopení uvádíme dva úryvky kódu, které provádějí stejnou věc, i když kód používající metodu `dict.get()` je efektivnější:

```
words[word] = words.get(word, 0) + 1
```

```
if word not in words:
    words[word] = 0
words[word] += 1
```

V následující podčásti, v níž se budeme věnovat výchozím slovníkům, se seznámíme s alternativním řešením.

Jakmile naplníme slovník všech slov, pustíme se do procházení jeho klíčů (tj. slov) v uspořádaném pořadí, přičemž vypíšeme každé slovo a počet jeho výskytů.

Díky metodě `dict.get()` můžeme snadno aktualizovat hodnoty slovníku, ovšem za předpokladu, že hodnoty jsou prosté prvky jako čísla nebo řetězce. Co ale dělat v případě, kdy je každá hodnota sama kolekcí? Řešení si ukážeme na programu, který načte soubory HTML zadané na příkazovém řádku a vypíše seznam všech jedinečných webů, na které se tyto soubory odkazují, společně se seznamem odkazujících souborů uvedených pod názvem každého webu. Svou strukturou je tento program (`external_sites.py`) velice podobný programu s jedinečnými slovy, který jsme si právě prostudovali. Zde je hlavní část kódu:

```
sites = {}
for filename in sys.argv[1:]:
    for line in open(filename):
        i = 0
        while True:
            site = None
            i = line.find("http://", i)
            if i > -1:
                i += len("http://")
                for j in range(i, len(line)):
                    if not (line[j].isalnum() or line[j] in ".-"):
                        site = line[i:j].lower()
                        break
                if site and "." in site:
                    sites.setdefault(site, set()).add(filename)
                i = j
            else:
                break
```

Začínáme vytvořením prázdného slovníku. Poté procházíme všechny soubory uvedené na příkazovém řádku a všechny řádky v každém z nich. Musíme vzít v potaz skutečnost, že každý řádek může odkazovat na libovolný počet webů, kvůli čemuž voláme metodu `str.find()` až do okamžiku, kdy volání selže. Pokud nalezneme řetězec „http://“, inkrementujeme proměnnou i (naše počáteční indexová pozice) o délku řetězce „http://“ a poté se podíváme na každý následující znak, dokud nenarazíme na takový, který není platný pro název webu. Nalezneme-li web (pouze pokud obsahuje tečku, což je minimální a jednoduchá kontrola), přidáme jej do slovníku.

Syntaxi `sites[site].add(filename)` použít nemůžeme, protože tím by při prvním výskytu nového webu došlo k vyvolání výjimky `KeyError`. Koneckonců nelze přece přidávat prvek do množiny, která je hodnotou prvku, který ve slovníku zatím neexistuje. Proto musíme použít jiný přístup. Metoda `dict.setdefault()` vrací odkaz na objekt, který je prvkem ve slovníku se zadaným klíčem (první argument). Pokud takový prvek neexistuje, metoda vytvoří nový prvek se zadaným klíčem a nastaví jeho hodnotu buď na `None`, nebo na zadanou výchozí hodnotu (druhý argument). V tomto případě předáváme výchozí hodnotu `set()`, tedy prázdnou množinu, takže volání `dict.setdefault()` vždy vrátí odkaz na objekt ukazující na prvek, který již existoval nebo který byl nově vytvořen. (Není-li zadaný klíč hashovatelný, tak samozřejmě dojde k vyvolání výjimky `TypeError`.)

V tomto příkladu ukazuje vrácený odkaz na objekt vždy na množinu (při prvním použití určitého klíče neboli webu ukazuje na prázdnou množinu), do níž poté přidáme název souboru, který odkazuje na daný web. Díky tomu, že používáme množinu, máme zajištěno, že i kdyby se soubor na nějaký web odkazoval opakovaně, zaznamenáme pro takový web název souboru pouze jednou.

Zde jsou dva ekvivalentní úryvky kódu, které přehledným způsobem objasňují fungování metody `dict.setdefault()`:

```
sites.setdefault(site,
                 set()).add(fname)
```

```
if site not in sites:
    sites[site] = set()
sites[site].add(fname)
```

Pro úplnost uvádíme ještě zbývající část programu:

```
for site in sorted(sites):
    print("na {0} se odkazuje soubor:".format(site))
    for filename in sorted(sites[site], key=str.lower):
        print("    {0}".format(filename))
```

sorted() ➤ 138, 144

Každý web se vypíše se soubory, které na něj odkazují. Tyto soubory se vypíší pod web s určitým odsazením. Volání funkce `sorted()` ve vnějším cyklu `for ... in` seřadí všechny klíče slovníku. Kdykoliv je slovník použit v kontextu, který vyžaduje iterovatelný objekt, použijí se jeho klíče. Pokud chceme procházet prvky (klíč, hodnota) nebo hodnoty, můžeme použít metodu `dict.items()` nebo `dict.values()`. Vnitřní cyklus `for ... in` prochází seřazené názvy souborů z množiny názvů souborů aktuálního webu.

Rozbalení mapování ➤ 177

I když slovník webů bude nejspíše obsahovat spoustu prvků, řada jiných slovníků obsahuje pouze několik málo prvků. U malých slovníků můžeme vypsat jejich obsah pomocí jejich klíčů jakožto názvů polí a prostřednictvím rozbalení mapování převést pro metodu `str.format()` prvky slovníku ve tvaru klíč-hodnota na argumenty ve tvaru klíč-hodnota.

Použití metody str.format() s rozbalením mapování ➤ 86

```
>>> greens = dict(zelená="#0080000", olivová="#808000", citrusová="#00FF00")
>>> print("{zelená} {olivová} {citrusová}".format(**greens))
#0080000 #808000 #00FF00
```

Zde má použití rozbalení mapování (`**`) naprosto stejný efekt jako zápis `.format(zelená=greens.zelená, olivová=greens.olivová, citrusová=greens.citrusová)`, ovšem zápis první varianty je snazší a čistší. Všimněte si, že nezáleží na tom, zda má slovník více klíčů, než potřebujeme, protože se použijí pouze ty klíče, jejichž jména se objevují ve formátovacím řetězci.

Slovníkové komprehenze

Slovníková komprehenze je výraz a cyklus s volitelnou podmínkou uzavřený do složených závorek, velice podobný množinové komprehenzi. K dispozici jsou podobně jako u seznamových a množinových komprehenzí dvě syntaxe:

```
{klíčový_výraz: hodnotový_výraz for klíč, hodnota in iterovatelný_objekt}
{klíčový_výraz: hodnotový_výraz for klíč, hodnota in iterovatelný_objekt
 if podmínka}
```

moduly os a os.path ➤ 219

Následující kód ukazuje, jak bychom mohli pomocí slovníkové komprehenze vytvořit slovník, v němž je každý klíč název souboru v aktuálním adresáři a každá hodnota velikostí tohoto souboru v bajtech:

```
file_sizes = {name: os.path.getsize(name) for name in os.listdir(".")}
```

Funkce `os.listdir()` modulu `os` („operační systém“) vrací seznam souborů a adresářů na zadané cestě, i když nikdy do seznamu nepřidává „.“ ani „..“. Funkce `os.path.getsize()` vrací velikost zadaného souboru v bajtech. Adresáře a všechny záznamy nepředstavující soubory můžeme vynechat přidáním podmínky:

```
file_sizes = {name: os.path.getsize(name) for name in os.listdir(".")
              if os.path.isfile(name)}
```

Funkce `os.path.isfile()` modulu `os.path` vrací hodnotu `True`, pokud zadaná cesta představuje soubor. V opačném případě (tj. v případě adresářů, linků a dalších nesouborových prvků) vrací hodnotu `False`.

Slovníkovou komprehenzi lze použít také k vytvoření převráceného slovníku. Máme-li například slovník `d`, pak můžeme vytvořit nový slovník, jehož klíče, resp. hodnoty, jsou hodnoty, resp. klíče, slovníku `d`:

```
inverted_d = {v: k for k, v in d.items()}
```

Výsledný slovník lze převrátit zpět na původní slovník, pokud jsou všechny hodnoty původního slovníku jedinečné. Není-li ovšem některá z hodnot hashovatelná, pak převrácení zpět selže s výjimkou `TypeError`.

Iterovatelné objekty ve slovníkové komprehenzi mohou být stejně jako v případě slovníkových a množinových komprehenzí další komprehenzí, takže lze vytvářet všechny druhy vnořených komprehenzí.

Výchozí slovníky

Výchozí slovníky jsou slovníky – mají všechny operátory a metody jako běžné slovníky. Co však výchozí slovníky odlišuje od běžných slovníků, je způsob jejich práce s chybějícími klíči. Ve všech ostatních směrech se chovají naprosto stejně jako normální slovníky. (V objektově orientovaném smyslu je `defaultdict` podtřídou třídy `dict`. Objektově orientovanému programování včetně odvozování tříd se budeme věnovat v lekci 6.)

Pokud při přístupu k slovníku použijeme neexistující („chybějící“) klíč, dojde k vyvolání výjimky `KeyError`. To je užitečné, protože často chceme vědět, zdali klíč, který očekáváme, je skutečně přítomen. Avšak v některých situacích chceme, aby byl přítomen každý klíč, který použijeme, i když to znamená, že při prvním přístupu ke klíči dojde k vložení určitého prvku do slovníku.

Představte si například, že máme slovník `d`, který pod klíčem `m` *nemá* žádný prvek. Pak kód x = d[m] vyvolá výjimku `KeyError`. Pokud by ale slovník `d` byl vhodně vytvořeným výchozím slovníkem, pak bychom v případě, že se prvek pod klíčem `m` ve výchozím slovníku nachází, obdrželi odpovídající hodnotu. Pokud by ale `m` nebyl klíč výchozího slovníku, vytvořil by se pod klíčem `m` nový prvek s výchozí hodnotou a my bychom obdrželi hodnotu tohoto nově vytvořeného prvku.

Dříve jsme napsali malý program, který počítal jedinečná slova v souborech zadaných na příkazovém řádku. Slovník slov jsme vytvořili takto:

Program `unique-words1.py` ➤ 130

```
words = {}
```

Každý klíč ve slovníku `words` byl určitým slovem a každá hodnota číslem uchovávajícím počet výskytů příslušného slova ve všech načtených souborech. Níže uvedeným způsobem jsme tuto hodnotu inkrementovali pokaždé, když jsme narazili na příslušné slovo:

```
words[word] = words.get(word, 0) + 1
```

Museli jsme použít metodu `dict.get()`, abychom zohlednili situace, kdy na slovo narazíme poprvé (v této situaci potřebujeme vytvořit nový prvek s hodnotou 1) a kdy na něj narazíme znovu (v této situaci potřebujeme přidat 1 ke stávající hodnotě slova).

Když vytváříme výchozí slovník, můžeme mu předat tzv. *tovární funkci* (faktory function). Tovární funkce je funkce, která při svém zavolání vrátí objekt určitého typu. Všechny vestavěné datové typy jazyka Python lze použít jako tovární funkce. Kupříkladu datový typ `str` lze zavolat jako funkci `str()`, která bez argumentů vrátí objekt představující prázdný řetězec. Tovární funkce předaná výchozímu slovníku se použije ke tvorbě výchozích hodnot pro chybějící klíče.

Je třeba poznamenat, že *název* funkce je odkaz na objekt ukazující na funkci. Chceme-li tedy předat funkci jako parametr, stačí zapsat její název. Když použijeme funkci s kulatými závorkami, říkají tyto závorky Pythonu, že chceme danou funkci zavolat.

Program `uniquewords2.py` má oproti původnímu programu `uniquewords1.py` ještě jeden řádek (`import collections`) a řádky pro tvorbu a aktualizaci slovníku má zapsané jiným způsobem. Výchozí slovník vytváří takto:

```
words = collections.defaultdict(int)
```

Výchozí slovník `words` nikdy nevyvolá výjimku `KeyError`. Pokud bychom napsali `x = words["xyz"]` a přitom by žádný prvek s klíčem `"xyz"` neexistoval, výchozí slovník by okamžitě vytvořil nový prvek s klíčem `"xyz"` a hodnotou `0` (zavoláním funkce `int()`), která by se přiřadila do proměnné `x`.

```
words[word] += 1
```

Metodu `dict.get()` tedy již vůbec nepotřebujeme. Místo ní můžeme prostě inkrementovat hodnotu prvku. Při prvním přístupu k určitému slovu se vytvoří nový prvek s hodnotou `0` (k níž okamžitě přičteme 1) a při každém dalším přístupu se k aktuální hodnotě přičítá 1.

Prohlídku všech vestavěných datových typů jazyka Python představujících kolekce a několika datových typů představujících kolekce ze standardní knihovny máme nyní úspěšně za sebou. V následující části se podíváme na některé problémy, které se týkají všech datových typů určených pro práci s kolekcemi.

3.1 Uspořádané slovníky

Typ `collections.OrderedDict` představují uspořádaný slovník. Tento typ byl zaveden v Pythonu 3.1 jakožto implementace návrhu PEP 372. Uspořádané slovníky lze použít jako přímou náhradu za neuspořádaný seznam, protože podporují stejné rozhraní API. Rozdíl mezi těmito dvěma typy slovníků spočívá v tom, že uspořádané slovníky uchovávají své prvky v pořadí, ve které do nich byly vloženy, což je vlastnost, která může být velice výhodná.

Je třeba poznamenat, že pokud uspořádanému slovníku při vytváření předáte slovník typu `dict` nebo klíčované argumenty, bude výsledné pořadí prvků nahodilé. To je dáno tím, že Python předává klíčované argumenty pomocí standardního neuspořádaného slovníku. K podobnému efektu dochází při použití metody `update()`. Z těchto důvodů by se při vytváření uspořádaného slovníku nebo při volání metody `update()` neměly používat klíčované argumenty ani neuspořádaný slovník. Pokud ovšem při vytváření uspořádaného slovníku použijeme seznam nebo n-tici s dvojicemi

klíčhodnota, zůstane pořadí prvků zachováno (poněvadž všechny prvky se předají jako jediný element – seznam nebo n-tice).

Uspořádaný slovník můžeme pomocí seznamu s dvojicemi prvků vytvořit takto:

```
d = collections.OrderedDict([('z', -4), ('e', 19), ('k', 7)])
```

Jako argument jsme použili jediný seznam, a proto bude pořadí klíčů zachováno. Uspořádané seznamy se však častěji vytvářejí inkrementálním způsobem:

```
tasks = collections.OrderedDict()
tasks[8031] = "Backup"
tasks[4027] = "Scan Email"
tasks[5733] = "Build System"
```

Pokud bychom stejným způsobem vytvořili neuspořádaný slovník a požádali o jeho klíče, bylo by pořadí těchto klíčů nahodilé. Avšak v případě uspořádaného slovníku se můžeme spolehnout na to, že pořadí vrácených klíčů bude stejné, v jakém jsme tyto klíče do slovníku vložili. Pokud bychom tedy napsali `list(d.keys())`, pak máme jistotu, že obdržíme seznam `['z', 'e', 'k']`, a pokud bychom napsali `list(tasks.keys())`, pak víme, že obdržíme seznam `[8031, 4027, 5733]`.

Další pěknou vlastností uspořádaných slovníků je to, že pokud změníme *hodnotu* nějakého prvku, což znamená, pokud vložíme prvek s existujícím klíčem, pak se pořadí nijak nezmění. Pokud bychom tedy napsali `tasks[8031] = "Denní záloha"` a poté požádali o seznam klíčů, obdrželi bychom úplně stejný seznam s prvky uspořádanými naprosto stejně jako předtím.

Pokud bychom chtěli přesunout nějaký prvek na konec, musíme jej smazat a pak znovu vložit. K odstranění a získání posledního prvku klíč-hodnota můžeme v uspořádaném slovníku také zavolat metodu `popitem()`. Tu můžeme zavolat jako `popitem(last=False)`, při čemž se odstraní a vrátí první prvek.

Další trošku specializovanější použití uspořádaných slovníků spočívá ve vytváření *seřazených* slovníků. Máme-li slovník d, pak jej můžeme převést na seřazený slovník takto: `d = collections.OrderedDict(sorted(d.items()))`. Všimněte si, že pokud bychom chtěli vložit nějaké další klíče, vložily by se na konec slovníku. Pokud bychom v takovém případě chtěli zachovat slovník seřazený, museli bychom jej opětovně vytvořit provedením stejného kódu, kterým jsme jej vytvořili poprvé. Vkládání a opětovné vytváření není tak neefektivní, jak to může na první pohled vypadat, poněvadž řadicí algoritmus Pythonu je vysoce optimalizovaný, zejména pro částečně seřazená data. Na druhou stranu je ale třeba počítat s tím, že se jedná o potenciálně dražší operaci.

Obecně platí, že používání uspořádaného slovníku k vytvoření seřazeného slovníku dává smysl pouze tehdy, pokud očekáváme, že budeme slovník procházet vícekrát, a pokud neočekáváme, že budeme po vytvoření seřazeného slovníku vkládat další prvky (nebo že jich budeme vkládat mnoho). (Implementaci skutečně seřazeného slovníku, který automaticky udržuje své klíče seřazené, si ukážeme v lekci 6 na straně 269.

Procházení a kopírování kolekcí

Jakmile máme kolekce datových prvků, je potřebné mít nějakou možnost, jak tyto prvky procházet. V první podčásti této části se seznámíme s některými iterátory jazyka Python a také s operátory a funkcemi, které se iterátorů týkají.

Dalším běžným požadavkem je kopírování kolekcí. S tímto tématem souvisí několik zákeřností, což je dáno způsobem, jakým Python používá odkazy na objekty (kvůli efektivitě). Ve druhé podčásti této části se tedy podíváme, jak můžeme kolekce kopírovat a jak u toho docílit takového chování, jaké chceme.

Operace a funkce pro iterátory a iterovatelné objekty

__iter__() ➤ 267

Iterovatelný datový typ je takový typ, který umí postupně vracet každý z jeho prvků. Jakýkoliv objekt, který má metodu `__iter__()`, a jakákoliv posloupnost (tj. objekt, která má metodu `__getitem__()` přijímající celočíselné argumenty začínající od nuly) jsou iterovatelné objekty, které umí poskytovat *iterátor*. Iterátor je objekt, který nabízí metodu `__next__()`, která vrací vždy následující prvek nebo vyvolá výjimku `StopIteration`, pokud už žádné další prvky nejsou k dispozici. Tabulka 3.4 uvádí operátory a funkce, které lze použít s iterovatelnými objekty.

Tabulka 3.4: Operace a funkce pro iterovatelné objekty

Syntaxe	Popis
s + t	Vrátí posloupnost, která je spojením posloupností `s` a `t`.
s * n	Vrátí posloupnost, která je `n`-násobným (`n` je typu `int`) spojením posloupnosti `s`.
x in i	Vrátí hodnotu `True`, pokud je prvek `x` v iterovatelném objektu `i`. K obrácenému testu se používá operátor `not in`.
all(i)	Vrátí hodnotu `True`, pokud se každý prvek v iterovatelném objektu i vyhodnotí na hodnotu `True`.
any(i)	Vrátí hodnotu `True`, pokud se libovolný prvek v iterovatelném objektu `i` vyhodnotí na hodnotu `True`.
enumerate(i, *`start`*)	Běžně se používá v cyklech `for ... in` k poskytnutí posloupnosti n-tice (index, prvek) s indexy začínajícími od `0` nebo od hodnoty *`start`* (viz následující text).
len(x)	Vrací „délku“ `x`. Pokud je `x` kolekcí, jedná se o počet prvků. Pokud je `x` řetězec, jedná se o počet znaků.
max(i, *`klíč`*)	Vrací největší prvek v iterovatelném objektu `i` nebo prvek s největší hodnotou *`klíč`* (*`prvek`*), je-li zadána funkce *`klíč`*.
min(i, *`klíč`*)	Vrací nejmenší prvek v iterovatelném objektu `i` nebo prvek s nejmenší hodnotou *`klíč`* (*`prvek`*), je-li zadána funkce *`klíč`*.
range(*`start`*, *`stop`*, `krok`)	Vrátí celočíselný iterátor. S jedním argumentem (*`stop`*) jde iterátor od `0` po hodnotu *`stop`* `- 1`. Se dvěma argumenty (*`start`*, *`stop`*) jde iterátor od hodnoty *`start`* po hodnotu *`stop`* `- 1`. Se třemi argumenty jde od hodnoty *`start`* po hodnotu *`stop`* `- 1` po krocích velikosti *`krok`*.

Syntaxe	Popis
reversed(i)	Vrátí iterátor, který vrací prvky z iterátoru `i` v opačném pořadí.
sorted(i, *klíč*, *obrátit*)	Vrátí seznam prvků z iterátoru i v seřazeném pořadí. Parametr klíč se používá k řazení typu DSU (Decorate, Sort, Undecorate – dekoruj, seřaď, odeber dekoraci). Má-li parametr obrátit hodnotu `True`, provede se seřazení v obráceném pořadí.
sum(i, *start*)	Vrátí součet prvků v iterovatelném objektu `i` plus *start* (výchozí hodnota je 0). Iterovatelný objekt `i` může obsahovat také řetězce.
zip(i1, ..., iN)	Vrátí iterátor n-tic pomocí iterátorů `i1` až `iN` (viz následující text).

Pořadí, ve kterém se prvky vrátí, závisí na použitém iterovatelném objektu. V případě seznamů a n-tic, se prvky vrátí v sekvenčním pořadí od prvního prvku (indexová pozice 0), ovšem některé iterátory, jako jsou kupříkladu slovníkové a množinové iterátory, vracejí prvky v nahodilém pořadí.

Vestavěná funkce `iter()` vykazuje dvě docela odlišná chování. Když jí předáme nějaký datový typ představující kolekci nebo nějakou posloupnost, tak vrátí iterátor pro zadaný objekt (nebo vyvolá výjimku `TypeError`, nelze-li objekt procházet). Toto použití bývá obvyklé při tvorbě vlastních datových typů představujících kolekce, v ostatních situacích je však jen zřídkakdy potřebné. K druhému chování funkce `iter()` dochází tehdy, když této funkci předáváme volatelný objekt (tj. funkci nebo metodu) a indikátor konce. V tomto případě se zadaná funkce zavolá jednou při každé iteraci a přitom se buď vrátí její návratová hodnota, nebo dojde k vyvolání výjimky `StopIteration`, pokud se návratová hodnota rovná indikátoru konce.

Při použití cyklu `for prvek in iterovatelný_objekt` dochází ve skutečnosti k tomu, že Python pro získání iterátoru volá `iter(iterovatelný_objekt)`. Poté se v každé iteraci cyklu pro získání následujícího prvku zavolá metoda `__next__()` tohoto iterátoru, a jakmile se vyvolá výjimka `StopIteration`, dojde k jejímu zachycení a ukončení cyklu. Další možností, jak získat následující prvek iterátoru, je zavolat vestavěnou funkci `next()`. Zde jsou dva ekvivalentní úryvky kódu (násobení hodnot v seznamu), přičemž jeden používá cyklus `for ... in` a druhý explicitní iterátor:

```
product = 1
for i in [1, 2, 4, 8]:
    product *= i
print(product) # prints: 64
```

```
product = 1
i = iter([1, 2, 4, 8])
while True:
    try:
        product *= next(i)
    except StopIteration:
        break
print(product) # prints: 64
```

Jakýkoliv (končený) iterovatelný objekt `i` lze převést zavoláním `tuple(i)` na n-tici nebo `list(i)` na seznam.

Funkcionální styl programování ➢ 381

Funkce `all()` a `any()` je možné použít na iterátory a často se používají při programování ve funkcionálním stylu. Zde je několik příkladů, které ukazují praktické použití funkcí `all()`, `any()`, `len()`, `min()`, `max()` a `sum()`:

```
>>> x = [-2, 9, 7, -4, 3]
>>> all(x), any(x), len(x), min(x), max(x), sum(x)
(True, True, 5, -4, 9, 13)
>>> x.append(0)
>>> all(x), any(x), len(x), min(x), max(x), sum(x)
(False, True, 6, -4, 9, 13)
```

Ze všech těchto funkcí se nejčastěji používá funkce `len()`.

Funkce enumerace přijímá iterátor a vrací enumerátorový objekt. S tímto objektem lze pracovat podobně jako s iterátorem a v každé iteraci vrací n-tici se dvěma prvky, z nichž první je číslo iterace (standardně začíná od 0) a druhý je další prvek z iterátoru, na němž byla funkce `enumerate()` zavolána. Pojďme se podívat na použití funkce `enumerate()` v kontextu malinkého, ale uceleného prográmku.

Program `grepword.py` vezme slovo a jeden nebo více názvů souborů z příkazového řádku a vypíše název souboru, číslo řádku a řádek pokaždé, když daný řádek obsahuje zadané slovo.* Zde je ukázkový běh programu:

```
grepword.py Dom data/forenames.txt
data/forenames.txt:615:Dominykas
data/forenames.txt:1435:Dominik
data/forenames.txt:1611:Domhnall
data/forenames.txt:3314:Dominic
```

Datové soubory `data/forenames.txt` a `data/surnames.txt` obsahují neseřazený seznam jmen a příjmení, z nichž každé je na jednom řádku.

Kromě příkazu `import sys` má program pouze následujících deset řádků:

```
if len(sys.argv) < 3:
    print("použití: grepword.py slovo soubor1 [soubor2 [... souborN]]")
    sys.exit()

word = sys.argv[1]
for filename in sys.argv[2:]:
    for lino, line in enumerate(open(filename), start=1):
        if word in line:
            print("{0}:{1}:{2:.40}".format(filename, lino,
                                           line.rstrip()))
```

* V lekci 10 uvidíme dvě další implementace tohoto programu s názvy `grepword-p.py` a `grepwordt.py`, které rozdělují práci na více procesů a více vláken.

Začneme kontrolou, zda máme k dispozici alespoň dva argumenty příkazového řádku. Pokud ne, vypíšeme zprávu s informacemi o použití a ukončíme program. Funkce `sys.exit()` provede okamžité čisté ukončení, při kterém uzavře všechny případné otevřené soubory. Tato funkce přijímá volitelný argument typu `int`, který se předá volajícímu shellu.

Předpokládáme, že prvním argumentem je slovo, které uživatel hledá, a že ostatními argumenty jsou názvy souborů, ve kterých se má toto slovo hledat. Vědomě jsme zavolali funkci open() bez stanovení kódování, protože uživatel může použít zástupné symboly pro zadání libovolného počtu souborů, z nichž každý může mít jiné kódování, takže v tomto případě necháme Python, aby použil kódování závislé na používané platformě.

Panel „Čtení a zápis textových souborů“ ➢ 131

Objekt souboru vrácený funkcí `open()` v textovém režimu lze použít jako iterátor, který v každé iteraci vrací jeden řádek souboru. Předáním tohoto iterátoru funkci `enumerate()` získáme enumerátorový iterátor, který v každé iteraci vrací číslo iterace (v proměnné `lino`, která uchovává číslo řádku) a řádek ze souboru. Je-li slovo, které uživatel hledá, na daném řádku, vypíšeme název souboru, číslo řádku a prvních 40 znaků řádku bez koncového bílého místa (tedy např. bez „\n“). Funkce `enumerate()` přijímá volitelný klíčovaný argument `start`, jehož výchozí hodnota je `0`. Tento argument jsme nastavili na hodnotu 1, protože čísla řádků v textových souborech se standardně počítají od 1.

Docela často nepotřebujeme enumerátor, ale spíše iterátor, který vrací po sobě jdoucí celá čísla. A právě k tomuto účelu slouží funkce `range()`. Pokud potřebujeme seznam nebo n-tici celých čísel, pak můžeme pomocí příslušné převodní funkce převést iterátor vrácený funkcí `range()`. Zde je několik příkladů:

```
>>> list(range(5)), list(range(9, 14)), tuple(range(10, -11, -5))
([0, 1, 2, 3, 4], [9, 10, 11, 12, 13], (10, 5, 0, -5, -10))
```

Funkce `range()` se nejčastěji používá ke dvěma účelům: pro vytváření seznamů nebo n-tic celých čísel a k poskytování pokračování cyklu v cyklech `for ... in`. Například následující dva ekvivalentní příklady zajišťují, aby všechny prvky seznamu `x` byly nezáporné:

```
for i in range(len(x)):
    x[i] = abs(x[i])
```

```
i = 0
while i < len(x):
    x[i] = abs(x[i])
    i += 1
```

Pokud měl seznam `x` původně prvky `[11, -3, -12, 8, -1]`, pak bude mít po provedení kteréhokoli z uvedených úryvků kódu prvky `[11, 3, 12, 8, 1]`.

Iterovatelný objekt můžeme rozbalit pomocí operátoru `*`, a proto můžeme rozbalit také iterátor vrácený funkcí `range()`. Máme-li například funkci s názvem `calculate()`, která přijímá čtyři argumenty, pak můžeme pro její zavolání s argumenty 1, 2, 3, a 4 použít kterýkoli z následujících způsobů:

```
calculate(1, 2, 3, 4)
t = (1, 2, 3, 4)
calculate(*t)
calculate(*range(1, 5))
```

Ve všech třech voláních předáváme čtyři argumenty. Ve druhém volání rozbalujeme n-tici se 4 prvky a ve třetím rozbalujeme iterátor vrácený funkcí `range()`.

Nyní se podíváme na několik malých, ale ucelených programů, pomocí nichž si upevníme některé znalosti několika témat, kterým jsme se dosud věnovali, a poprvé budeme zapisovat do souboru. Program `generate_test_names1.py` načte soubor s křestními jmény a soubor s příjmeními, vytvoří dva seznamy a poté vytvoří soubor test-names1.txt a zapíše do něj 100 náhodných jmen.

Použijeme funkci `random.choice()`, která přijímá náhodný prvek z posloupnosti, takže se mohou vyskytnout duplicitní jména. Nejdříve se podíváme na funkci, která vrací seznamy jmen, a poté se podíváme na zbývající část programu.

```
def get_forenames_and_surnames():
    forenames = []
    surnames = []
    for names, filename in ((forenames, "data/forenames.txt"),
                            (surnames, "data/surnames.txt")):
        for name in open(filename, encoding="utf8"):
            names.append(name.rstrip())
    return forenames, surnames
```

Rozbalení n-tice ➤ 113

Ve vnějším cyklu `for ... in` procházíme přes dvě n-tice se dvěma prvky, z nichž každou rozbalujeme do dvou proměnných. I když mohou být tyto dva seznamy docela velké, jejich předání z funkce je efektivní, protože Python používá odkazy na objekty, takže jediné, co funkce skutečně vrací, je n-tice dvou odkazů na objekty.

Uvnitř programů Pythonu se standardně vždy používají cesty ve stylu Unixu, poněvadž je můžeme psát bez nutnosti používat dvě zpětná lomítka, a navíc fungují na všech platformách (včetně Windows). Pokud máme cestu, kterou chceme předložit uživateli, třeba v proměnné `path`, můžeme vždy importovat modul os a zavolat `path.replace("/", os.sep)` pro nahrazení lomítek oddělovači adresářů specifickými pro používanou platformu.

```
forenames, surnames = get_forenames_and_surnames()
fh = open("test-names1.txt", "w", encoding="utf8")
for i in range(100):
    line = "{0} {1}\n".format(random.choice(forenames),
                              random.choice(surnames))
    fh.write(line)
```

Panel „Čtení a zápis textových souborů“ ➤ 131

Po získání obou seznamů otevřeme výstupní soubor pro zápis a objekt souboru uložíme do proměnné `fh` („file handle“). Poté procházíme cyklem 100krát po sobě a při každé iteraci vytvoříme řádek, který bude zapsán do souboru, přičemž nezapomeneme připojit na konec každého řádku znak nového řádku. Proměnnou cyklu `i` vůbec nepoužíváme, je zde uvedena jen kvůli správně zapsané syntaxi cyklu `for ... in`. Předchozí úryvek kódu, funkce `get_forenames_and_surnames()` a příkaz import tvoří celý program.

V programu `generate_test_names1.py` spojujeme prvky ze dvou samostatných seznamů dohromady do řetězců. Další způsob kombinování prvků ze dvou či více seznamů (nebo jiných iterovatelných

objektů) spočívá v použití funkce `zip()`. Funkce `zip()` přijímá jeden nebo více iterovatelných objektů a vrací iterátor, který vrací n-tice. První n-tice má první prvky z každého iterovatelného objektu, druhá n-tice má druhé prvky z každého iterovatelného objektu a tak dále. Toto zpracování se zastaví, jakmile je jeden ze zadaných iterovatelných objektů vyčerpán. Zde je příklad:

```
>>> for t in zip(range(4), range(0, 10, 2), range(1, 10, 2)):
...     print(t)
(0, 0, 1)
(1, 2, 3)
(2, 4, 5)
(3, 6, 7)
```

I když každý z iterátorů vrácených druhým a třetím voláním funkce `range()` může vytvořit pět prvků, první dokáže vytvořit pouze čtyři, což omezuje počet prvků, které může funkce `zip()` vrátit na čtyři n-tice.

Zde je modifikovaná verze programu pro generování testovacích jmen. Tentokrát každé jméno zabírá 25 znaků a za nimi následuje náhodný rok. Program se nazývá `generate_test_names2.py` a výstup zapisuje do souboru `test-names2.txt`. Funkci `get_forenames_and_surnames()` a volání `open()` jsme si neukázali, protože kromě výstupního souboru jsou stejné jako dříve.

```
limit = 100
years = list(range(1970, 2013)) * 3
for year, forename, surname in zip(
        random.sample(years, limit),
        random.sample(forenames, limit),
        random.sample(surnames, limit)):
    name = "{0} {1}".format(forename, surname)
    fh.write("{0:.<25}.{1}\n".format(name, year))
```

Začínáme nastavením omezení na počet jmen, která chceme generovat. Poté vytvoříme seznam let, který obsahuje roky od 1970 až po 2012 včetně, a pak tento seznam třikrát replikujeme tak, aby se ve výsledném seznamu každý rok vyskytoval třikrát. To je nezbytné, protože funkce `random.sample()`, kterou používáme (místo funkce `random.choice()`) přijímá iterovatelný objekt a počet prvků, které má vytvořit. Jedná se o číslo, které nesmí být menší než počet prvků, které dokáže daný iterovatelný prvek vrátit. Funkce `random.sample()` vrací iterátor, který vytvoří nejvýše stanovený počet prvků ze zadaného iterovatelného prvku, a to bez opakování. Tato verze programu tedy generuje vždy jedinečná jména.

V cyklu `for ... in` rozbalíme každou n-tici vrácenou funkcí `zip()`. Chceme omezit délku každého jména na 25 znaků. K tomu musíme nejdříve vytvořit řetězec s úplným jménem a poté nastavit jeho maximální šířku druhém volání při metody `str.format()`. Každé jméno zarovnáme vlevo a u jmen kratších než 25 znaků doplníme tečky. Další tečka navíc zajistí, že jména, která zabírají celou šířku pole, budou i tak oddělena od roku tečkou.

Rozbalení n-tice ➤ 113

str.format() ➤ 83

Tuto podčást uzavřeme zmínkou o dvou dalších funkcích souvisejících s iterovatelnými objekty: `sorted()` a `reversed()`. Funkce `sorted()` vrací seznam se seřazenými prvky a funkce `reversed()`

vrací jednoduše iterátor, který probíhá v opačném pořadí vůči iterátoru zadanému jako argument. Zde je příklad použití funkce `reversed()`:

```
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
>>> list(reversed(range(6)))
[5, 4, 3, 2, 1, 0]
```

Funkce `sorted()` je sofistikovanější, jak ukazují následující příklady:

```
>>> x = []
>>> for t in zip(range(-10, 0, 1), range(0, 10, 2), range(1, 10, 2)):
...     x += t
>>> x
[-10, 0, 1, -9, 2, 3, -8, 4, 5, -7, 6, 7, -6, 8, 9]
>>> sorted(x)
[-10, -9, -8, -7, -6, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sorted(x, reverse=True)
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -6, -7, -8, -9, -10]
>>> sorted(x, key=abs)
[0, 1, 2, 3, 4, 5, 6, -6, -7, 7, -8, 8, -9, 9, -10]
```

V předchozím úryvku kódu vrací funkce zip n-tice se třemi prvky: `(-10, 0, 1)`, `(-9, 2, 3)` a tak dále. Operátor += rozšiřuje seznam, což znamená, že připojí každý prvek v zadané posloupnosti k seznamu.

První volání funkce `sorted()` vrátí kopii seznamu s použitím standardního seřazení. Druhé volání vrátí kopii seznamu seřazeného v obráceném pořadí. V posledním volání funkce `sorted()` specifikujeme „klíčovou" funkci, k níž se za okamžik vrátíme.

Všimněte si, že funkce jazyka Python jsou objekty jako každé jiné, a proto je lze předávat jako argumenty jiným funkcím a také přímo ukládat do kolekcí. Vzpomeňte si, že název funkce je odkaz na objekt této funkce. Jsou to právě závorky, které následují za názvem, co říká Pythonu, aby funkci zavolal.

Když předáme klíčovou funkci (v tomto případě funkci `abs()`), zavolá se jednou pro každý prvek v seznamu (přičemž se jí předá tento prvek jako její jediný argument), čímž se vytvoří „dekorovaný" seznam. Poté se tento dekorovaný seznam seřadí a vrátí se již bez dekorace jako výsledek. Jako klíčovou funkci můžeme klidně použít naši vlastní funkci, jak si ukážeme za chvíli.

Můžeme například seřadit seznam řetězců bez ohledu na velikost písmen tak, že jako klíčovou funkci stanovíme metodu `str.lower()`. Máme-li seznam `x` s prvky `["Sloup", "Yard", "Cibule", "sekaná", "kedlub"]`, můžeme je seřadit bez ohledu na velikost písmen pomocí řazení DSU (Decorate, Sort, Undecorate) na jediném řádku kódu tak, že zadáme klíčovou funkci nebo explicitně provedeme kroky DSU, což ukazují tyto dva ekvivalentní úryvky kódu:

```
x = sorted(x, key=str.lower)
```

```
temp = []
for item in x:
    temp.append((item.lower(), item))
x = []
for key, value in sorted(temp):
    x.append(value)
```

Oba uvedené úryvky kódu vytvářejí nový seznam: `["Cibule", "kedlub", "sekaná", "Sloup", "Yard"]`, ačkoliv výpočty, které provádějí, nejsou identické, protože úryvek na pravé straně vytváří dočasný seznam `temp`.

Řadicím algoritmem Pythonu je adaptivní, stabilní řazení slučováním (merge sort), které je rychlé a zároveň chytré a navíc je zvláště dobře optimalizované pro částečně seřazené seznamy, což je velmi častý případ.* Adaptivní znamená, že se řadicí algoritmus přizpůsobuje aktuálním okolnostem, například tím, že využívá částečně seřazená data. Stabilní znamená, že prvky, které jsou z hlediska pořadí na stejné pozici, se navzájem nepřesouvají (koneckonců to přece není třeba). A řazení slučováním je obecný název pro použití řadícího algoritmu. Při řazení kolekcí celých čísel, řetězců nebo ostatních jednoduchých typů se použije jejich operátor „menší než" (`<`). Python také dokáže seřadit kolekce, které obsahují jiné kolekce, přičemž pracuje rekurzivně do libovolné hloubky:

```
>>> x = list(zip((1, 3, 1, 3), ("pram", "dorie", "kayak", "canoe")))
>>> x
[(1, 'pram'), (3, 'dorie'), (1, 'kayak'), (3, 'canoe')]
>>> sorted(x)
[(1, 'kayak'), (1, 'pram'), (3, 'canoe'), (3, 'dorie')]
```

Python seřadil seznam těchto n-tic tak, že porovnává první prvky každé n-tice, a pokud jsou stejné, tak porovnává druhé prvky. N-tice se tak seřadí podle celých čísel a hned poté podle řetězců. Definováním jednoduché klíčové funkce si můžeme vynutit, aby řazení proběhlo podle řetězců a až poté podle celých čísel:

```
def swap(t):
    return t[1], t[0]
```

Funkce `swap()` přijímá n-tici se dvěma prvky a vrací novou n-tici se dvěma prvky s prohozenými argumenty. Za předpokladu, že jsme definici funkce `swap()` zapsali do editoru IDLE, můžeme napsat následující kód:

```
>>> sorted(x, key=swap)
[(3, 'canoe'), (3, 'dorie'), (1, 'kayak'), (1, 'pram')]
```

Seznamy lze řadit také na místě pomocí metody `list.sort()`, která přijímá stejné volitelné argumenty jako funkce `sorted()`.

* Algoritmus vytvořil Tim Peters. Zajímavé vysvětlení společně s diskuzí ohledně tohoto algoritmu najdete to souboru `listsort.txt`, který je součástí zdrojového kódu Pythonu.

Řazení můžeme aplikovat pouze na kolekce, v nichž lze všechny prvky navzájem porovnávat:

```
sorted([3, 8, -7.5, 0, 1.3])              # vrátí: [-7.5, 0, 1.3, 3, 8]
sorted([3, "francouzák", -7.5, 0, 1.3])   # vyvolá výjimku TypeError
```

I když první seznam obsahuje čísla různých typů (`int` a `float`), tyto typy lze navzájem porovnávat, takže seřazení tohoto seznamu proběhne bez problémů. Ale druhý seznam obsahuje řetězec, který nelze rozumě porovnat s číslem, a proto dojde k vyvolání výjimky `TypeError`. Pokud bychom chtěli seřadit seznam, který obsahuje celá čísla, čísla s pohyblivou řádovou čárkou a řetězce, které obsahují čísla, můžeme jako klíčovou funkci stanovit funkci `float()`:

```
sorted(["1.3", -7.5, "5", 4, "-2.4", 1], key=float)
```

Toto volání vrátí seznam `[-7.5, '-2.4', 1, '1.3', 4, '5']`. Všimněte si, že hodnoty seznamu se nezměnily, takže řetězce zůstaly řetězci. Pokud některý z řetězců není možné převést na číslo (např. „francouzák“), vyvolá se výjimka `ValueError`.

Kopírování kolekcí

Odkazy na objekty ➤ 26

Python používá odkazy na objekty, a proto při použití operátoru přiřazení (=) nedojde k žádnému kopírování. Je-li operand na pravé straně literál (např. řetězec nebo číslo), nastaví se operand na levé straně na odkaz na objekt, který ukazuje objekt v paměti, jenž uchovává hodnotu tohoto literálu. Je-li operand na pravé straně odkaz na objekt, nastaví se operand na levé straně na odkaz na objekt, který ukazuje na stejný objekt jako operand na pravé straně. Jedním z důsledků tohoto přístupu je to, že přiřazení je velice efektivní.

Když přiřazujeme rozsáhlé kolekce, jako jsou dlouhé seznamy, jsou úspory skutečně zřetelné. Zde je příklad:

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs
>>> beatles, songs
(['Because', 'Boys', 'Carol'], ['Because', 'Boys', 'Carol'])
```

Zde se vytvořil nový odkaz na objekt (`beatles`) a oba odkazy na objekty ukazují na stejný seznam – k žádnému kopírování nedošlo.

Seznamy jsou měnitelné, takže na ně můžeme aplikovat změnu:

```
>>> beatles[2] = "Cayenne"
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Cayenne'])
```

Změnu jsme aplikovali s použitím proměnné `beatles`, což je ale odkaz na objekt ukazující na stejný seznam, na který ukazuje proměnná `songs`. Jakékoli změny provedené prostřednictvím jednoho z odkazů na objekty se tedy projeví také u druhého. To je obvykle chování, které požadujeme, protože kopírování rozsáhlých kolekcí je potenciálně drahé. Kromě toho to znamená, že můžeme například předat seznam nebo jiný měnitelný datový typ představující kolekci jako argument funkci a uvnitř této funkce kolekci upravit. Takto upravená kolekce pak bude přístupná po dokončení volání této funkce.

Nicméně v některých situacích potřebujeme samostatnou kopii kolekce (nebo jiného měnitelného objektu). Když z posloupnosti bereme řez (např. `songs[:2]`), jedná se vždy o nezávislou kopii prvků. Celou posloupnost můžeme tedy zkopírovat takto:

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs[:]
>>> beatles[2] = "Cayenne"
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Carol'])
```

U slovníků a množin lze kopírování provádět pomocí metod `dict.copy()` a `set.copy()`. Kromě toho máme k dispozici modul `copy`, který nabízí funkci `copy.copy()`, jež vrací kopii zadaného objektu. Další možnost, jak kopírovat vestavěné typy představující kolekci, spočívá v použití typu jako funkce, které předáme jako argument kolekci, která se má zkopírovat. Zde je několik příkladů:

```
copy_of_dict_d = dict(d)
copy_of_list_L = list(L)
copy_of_set_s = set(s)
```

Všimněte si, že všechny tyto kopírovací techniky jsou *mělké*, což znamená, že se zkopírují pouze odkazy na objekty, a ne samotné objekty. Pro neměnitelné datové typy, jako jsou čísla a řetězce, to má stejný efekt jako kopírování (až na to, že je to mnohem efektivnější), ale pro měnitelné datové typy, jako jsou vnořené kolekce, to znamená, že na objekty, na které ukazují, se odkazuje původní i zkopírovaná kolekce. Tuto situaci demonstruje následující úryvek kódu:

```
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = x[:]          # mělká kopie
>>> x, y
([53, 68, ['A', 'B', 'C']], [53, 68, ['A', 'B', 'C']])
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['Q', 'B', 'C']])
```

Když se seznam `x` kopíruje mělce, zkopíruje se odkaz na vnořený seznam `["A", "B", "C"]`. To znamená, že `x` a `y` mají jako svůj třetí prvek odkaz na objekt, který ukazuje na tento seznam, takže libovolné změny tohoto vnořeného seznamu se projeví v obou seznamech `x` a `y`. Pokud skutečně potřebujeme nezávislé kopie libovolně vnořených kolekcí, pak musíme provést hloubkovou kopii:

```
>>> import copy
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = copy.deepcopy(x)
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['A', 'B', 'C']])
```

Zde jsou seznamy `x` a `y` a seznam prvků, který obsahují, zcela nezávislé. Je třeba poznamenat, že od této chvíle budeme používat oba termíny: kopie a mělká kopie. Pokud budeme mít na mysli hloubkovou kopii, pak to explicitně uvedeme.

Příklady

Dokončili jsme přehled datových typů představujících kolekce, které jsou vestavěné v jazyku Python, a také tři další ze standardní knihovny (`collections.namedtuple`, `collections.defaultdict` a `collections.OrderedDict`). Python dále nabízí typ `collections.deque`, což je fronta se dvěma konci, přičemž řada další typů představujících kolekce je k dispozici od třetích stran a na stránce Python Package Index (*http://pypi.python.org/pypi*). Nyní se podíváme na několik malinko delších příkladů, které se společně dotýkají mnoha témat probíraných v této a předchozí lekci.

První program má přibližně sedmdesát řádků a týká se zpracování textu. Druhý program je zhruba devadesát řádků dlouhý a má matematickou příchuť. Tyto programy využívají slovníky, seznamy, pojmenované n-tice, množiny a velkým dílem také metodu `str.format()` z předchozí lekce.

Program generate_usernames.py

Představte si, že připravujeme nový počítačový systém a potřebujeme vygenerovat uživatelská jména pro všechny pracovníky v naší organizaci. K dispozici máme holý textový soubor s daty (v kódování UTF-8), v němž každý řádek představuje záznam a pole jsou oddělena dvojtečkou. Každý záznam obsahuje informace o jednom členu personálu, což zahrnuje pole představující jedinečné ID pracovníka, křestní jméno, prostřední jméno (které může být prázdné), příjmení a název oddělení. Zde je ukázka několika řádků z datového souboru `data/users.txt`:

```
1601:Albert:Lukas:Montgomery:Legal
3702:Albert:Lukas:Montgomery:Sales
4730:Nadelle::Landale:Warehousing
```

Program musí načíst všechny datové soubory zadané na příkazovém řádku, z každého řádku (záznamu) extrahovat pole a vrátit data s vhodným uživatelským jménem. Každé uživatelské jméno musí být jedinečné a musí být založeno na jméno pracovníka. Výstup musí mít podobu textu odeslaného do konzoly a musí být abecedně seřazený podle příjmení a jména:

```
Jméno                            ID   Uživ. jm.
-------------------------------- ------ ---------
Milner, Kieara.................. (4454) kmilner
Montgomery, Albert L............ (1601) almontgo
Montgomery, Albert L............ (3702) almontgo1
```

Každý záznam má přesně pět polí, na která se sice můžeme odkazovat číslem, my však budeme raději používat jména, aby byl výsledný kód čistý:

```
ID, FORENAME, MIDDLENAME, SURNAME, DEPARTMENT = range(5)
```

V Pythonu se identifikátory zapsané velkými písmeny standardně považují za konstanty.

Dále potřebujeme vytvořit typ představující pojmenovanou n-tici pro uchovávání dat každého uživatele:

```
User = collections.namedtuple("User",
                              "username forename middlename surname id")
```

Na použití konstant a pojmenované n-tice `User` se podíváme ve zbývající části kódu.

Celková logika programu je zachycena ve funkci `main()`:

```
def main():
    if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
        print("použití: {0} soubor1 [soubor2 [... souborN]]".format(sys.argv[0]))
    sys.exit()

    usernames = set()
    users = {}
    for filename in sys.argv[1:]:
        for line in open(filename, encoding="utf8"):
            line = line.rstrip()
            if line:
                user = process_line(line, usernames)
                users[(user.surname.lower(), user.forename.lower(),
                       user.id)] = user
    print_users(users)
```

Pokud uživatel nezadá na příkazovém řádku žádný název souboru nebo pokud na příkazový řádek napíše „-h“ nebo „--h”, tak vypíšeme zprávu s informacemi o použití a ukončíme program.

U každého načteného řádku odřízneme případné koncové bílé místo (např. „\n“) a zpracujeme pouze ty řádky, které nejsou prázdné. To znamená, že pokud datový soubor obsahuje prázdné řádky, pak budou bezpečně ignorovány.

V množině `usernames` máme uložena všechna dosud přidělená uživatelská jména, čímž máme zajištěno, že nevytvoříme žádné duplicity. Samotná data uchováváme ve slovníku `users` s tím, že každý uživatel (člen personálu) je uložen jako prvek slovníku, jehož klíč je n-ticí tvořenou příjmením, jménem a identifikačním číslem uživatele a jehož hodnota je pojmenovanou n-ticí typu `User`. Díky tomu, že pro klíče slovníku používáme n-tici obsahující příjmení, jméno a ID, obdržíme po zavolání funkce `sorted()` na slovník iterovatelný objekt, který bude seřazený přesně tak, jak potřebujeme (tj. podle příjmení, jména a ID), aniž bychom museli použít klíčovou funkci.

```
def process_line(line, usernames):
    fields = line.split(":")
    username = generate_username(fields, usernames)
    user = User(username, fields[FORENAME], fields[MIDDLENAME],
                fields[SURNAME], fields[ID])
    return user
```

Vzhledem k tomu, že datový formát každého záznamu je takto jednoduchý a že jsme z řádku již odřízli koncové bílé místo, můžeme pole extrahovat prostým rozdělením řetězce podle dvojteček. Pole a uživatelská jména předáme funkci `generate_username()` a poté vytvoříme instanci pojmenované n-tice `User`, kterou vzápětí vrátíme volající funkci (`main()`), jež vloží uživatele do slovníku `users`, který je již připraven k výpisu do konzoly.

Pokud bychom nevytvořili vhodné konstanty uchovávající indexové pozice, museli bychom používat číselné indexy:

```
user = User(username, fields[1], fields[2], fields[3], fields[0])
```

Ačkoliv tento zápis je očividně kratší, jedná se o špatný přístup. Za prvé, pro ty, kteří budou kód v budoucnu udržovat, není jasné, co jednotlivá pole představují, a za druhé, tento přístup je zranitelný vůči změnám formátu datového souboru. Pokud se pořadí nebo počet polí v záznamu změní, pak tento kód havaruje všude, kde jej budeme chtít použít. S použitím pojmenovaných konstant nám při změnách struktury záznamu bude stačit změnit pouze hodnoty konstant a všechna místa, na kterých tyto konstanty používáme, budou nadále fungovat.

```
def generate_username(fields, usernames):
    username = ((fields[FORENAME][0] + fields[MIDDLENAME][:1] +
                 fields[SURNAME]).replace("-", "").replace("'", ""))
    username = original_name = username[:8].lower()
    count = 1
    while username in usernames:
        username = "{0}{1}".format(original_name, count)
        count += 1
    usernames.add(username)
    return username
```

První pokus o vytvoření uživatelského jména spočívá ve spojení prvního písmene křestního jména, prvního písmene prostředního jména a celého příjmení, přičemž se z výsledného řetězce vymažou všechny případné spojovníky nebo jednoduché uvozovky. Kód pro získání prvního písmene prostředního jména je docela zákeřný. Pokud bychom použili výraz `fields[MIDDLENAME][0]`, obdrželi bychom u prázdných prostředních jmen výjimku `IndexError`. Avšak při použití řezu získáme první písmeno, pokud je k dispozici, nebo prázdný řetězec.

V dalším kroku převedeme uživatelské jméno na malá písmena a zkrátíme jeho délku osm znaků, je-li delší. Pokud se toto uživatelské jméno již používá (tj. nachází se v množině `usernames`), pokusíme se na jeho konec připojit „1“, a pokud se již používá i toto uživatelské jméno, zkusíme připojit „2“ a tak pořád dál, dokud nezískáme takové, které se dosud nepoužívá. Poté uživatelské jméno přidáme do množiny `usernames` a vrátíme jej volajícímu.

```
def print_users(users):
    namewidth = 32
    usernamewidth = 9

    print("{0:<{nw}} {1:^6} {2:{uw}}".format(
          "Jméno", "ID", "Uživ. jm.", nw=namewidth, uw=usernamewidth))
```

```
    print("{0:-<{nw}} {0:-<6} {0:-<{uw}}".format(
          "", nw=namewidth, uw=usernamewidth))

    for key in sorted(users):
        user = users[key]
        initial = ""
        if user.middlename:
            initial = " " + user.middlename[0]
        name = "{0.surname}, {0.forename}{1}".format(user, initial)
        print("{0:.<{nw}} ({1.id:4}) {1.username:{uw}}".format(
              name, user, nw=namewidth, uw=usernamewidth))
```

Jakmile byly zpracovány všechny záznamy, zavoláme funkci `print_users()`, které předáme jako parametr slovník `users`.

První příkaz `print()` vypíše názvy sloupců a druhý vypíše pod každý název pomlčky. Volání metody `str.format()` u druhého příkazu je trošku zákeřnější. Řetězec, který předáváme k vypsání, je `""`, což je prázdný řetězec. Pomlčky totiž získáme tak, že vypíšeme prázdný řetězec, který se doplní pomlčkami do zadané šířky.

`str.format()` ➤ 83

V dalším kroku vypíšeme pomocí cyklu `for ... in` údaje o každém uživateli, přičemž extrahujeme klíče pro prvky slovníku každého uživatele v seřazeném pořadí. Proměnnou `user` vytváříme z toho důvodu, abychom nemuseli v celé funkci neustále psát `users[key]`. U prvního volání metody `str.format()` v cyklu nastavujeme proměnnou `name` na jméno uživatele ve formě křestního jména, příjmení (a volitelně iniciálky). K prvkům v pojmenované n-tici `user` přistupujeme prostřednictvím jména. Jakmile máme jméno uživatele v jednom řetězci, vypíšeme údaje o uživateli, přičemž omezíme každý sloupec (jméno, ID, uživatelské jméno) na požadovanou šířku.

Kompletní program (který se od toho, co jsme si ukázali, liší pouze v tom, že na začátku obsahuje pár řádků s komentáři a několik příkazů `import`) se nachází v souboru `generate_usernames.py`. Struktura tohoto programu (načtení datového souboru, zpracování každého záznamu, zapsání výstupu) se používá velice často a my se s ní opět setkáme v následujícím příkladu.

Program statistics.py

Představte si, že máme skupinu datových souborů obsahujících čísla související s nějakým námi provedeným zpracováním a že chceme vytvořit nějaké základní statistiky, abychom získali určitý druh přehledu nad svými daty. Každý soubor používá holý text (kódování ASCII) s jedním nebo více čísely na řádku (oddělenými bílým místem).

Zde je ukázka toho, jak by měl vypadat výstup:

```
počet      = 183
střed      = 130.56
medián     = 43.00
modus      = [5.00, 7.00, 50.00]
sm. odch.  = 235.01
```

Zde jsme načetli 183 čísel (5, 7 a 50 se vyskytují nejčastěji), která vykazují směrodatnou odchylku 235,01.

Samotné statistické údaje uchováváme v pojmenované n-tici s názvem `Statistics`:

```
Statistics = collections.namedtuple("Statistics",
                                    "mean mode median std_dev")
```

Funkce `main()` slouží též jako přehled struktury programu:

```
def main():
    if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
        print("použití: {0} soubor1 [soubor2 [... souborN]]".format(
            sys.argv[0]))
        sys.exit()

    numbers = []
    frequencies = collections.defaultdict(int)
    for filename in sys.argv[1:]:
        read_data(filename, numbers, frequencies)
    if numbers:
        statistics = calculate_statistics(numbers, frequencies)
        print_results(len(numbers), statistics)
    else:
        print("nebyla nalezena žádná čísla")
```

Všechna čísla ze všech souborů uchováváme v seznamu `numbers`. K výpočtu modu („nejčastěji se vyskytujících“) čísel potřebujeme vědět, kolikrát se každé číslo vyskytuje, a proto vytvoříme výchozí slovník používající tovární funkci `int()`, což nám umožní sledovat počty jednotlivých čísel.

Procházíme každý soubor a čteme jeho data. Seznam a výchozí slovník předáváme jako dodatečné parametry, aby je mohla funkce `read_data()` aktualizovat. Jakmile načteme všechna data a alespoň některá čísla se načtou úspěšně, zavoláme funkci `calculate_statistics()`. Ta vrátí pojmenovanou n-tici typu `Statistics`, kterou poté použijeme k vypsání výsledků.

```
def read_data(filename, numbers, frequencies):
    for lino, line in enumerate(open(filename, encoding="ascii"),
                                start=1):
        for x in line.split():
            try:
                number = float(x)
                numbers.append(number)
                frequencies[number] += 1
            except ValueError as err:
                print("{filename}:{lino}: skipping {x}: {err}".format(
                    **locals()))
```

Každý řádek rozdělujeme podle bílých míst a každý prvek se pokusíme převést na hodnotu typu `float`. Je-li převod úspěšný (v případě celých čísel a čísel s pohyblivou řádovou čárkou v desítkové i exponenciální notace), přidáme číslo do seznamu `numbers` a aktualizujeme výchozí slovník `frequencies`. (Pokud bychom použili typ `dict`, museli bychom pro aktualizaci použít kód `frequencies[number] = frequencies.get(number, 0) + 1`.)

Pokud převod selže, vypíšeme číslo řádku (počínaje řádkem 1, což je tradiční u textových souborů), text, který jsme se pokoušeli převést, a chybový text výjimky `ValueError`. Místo pozičních argumentů (např. `.format(filename, lino` atd.) nebo explicitně pojmenovaných argumentů (např. `format (filename=filename, lino=lino` atd.) získáváme názvy a hodnoty místních proměnných zavoláním funkce `locals()` a metodě `str.format()` předáváme tyto pojmenované argumenty klíč-hodnota pomocí rozbalení mapování.

Použití metody `str.format()` s rozbalením mapování ➢ 86

```
def calculate_statistics(numbers, frequencies):
    mean = sum(numbers) / len(numbers)
    mode = calculate_mode(frequencies, 3)
    median = calculate_median(numbers)
    std_dev = calculate_std_dev(numbers, mean)
    return Statistics(mean, mode, median, std_dev)
```

Tato funkce se používá pro shromáždění všech statistických údajů. Střed („průměr“) lze vypočítat velice snadno, a proto jej zde počítáme přímo. Pro ostatní statistické hodnoty voláme specializované funkce a na konci vracíme objekt pojmenované n-tice `Statistics`, který obsahuje čtyři vypočítané statistiky.

```
def calculate_mode(frequencies, maximum_modes):
    highest_frequency = max(frequencies.values())
    mode = [number for number, frequency in frequencies.items()
            if frequency == highest_frequency]
    if not (1 <= len(mode) <= maximum_modes):
        mode = None
    else:
        mode.sort()
    return mode
```

Nejčastěji se vyskytujících čísel může být více, a proto tato funkce vyžaduje, aby volající kromě slovníku `frequencies` stanovil také maximální počet přijatelných modů. (Volajícím je funkce `calculate_statistics()`, která stanoví maximálně tři mody.)

K nalezení nejvyšší hodnoty ve slovníku `frequencies` použijeme funkci `max()`. Poté použijeme seznamovou komprehenzi k vytvoření seznamu těch modů, jejichž četnost odpovídá nejvyšší hodnotě. Porovnáváme operátorem `==`, protože všechny četnosti jsou celá čísla.

Je-li počet modů 0 nebo větší než maximum přijatelných modů, vrátíme hodnotu `None`. V opačném případě vrátíme seřazený seznam modů.

```
def calculate_median(numbers):
    numbers = sorted(numbers)
```

```
    middle = len(numbers) // 2
    median = numbers[middle]
    if len(numbers) % 2 == 0:
        median = (median + numbers[middle - 1]) / 2
    return median
```

Medián („střední hodnota“) je hodnota, která se vyskytuje uprostřed, jsou-li čísla seřazena. Výjimkou je situace, kdy je počet čísel sudý. V takovém případě spadá střed mezi dvě čísla, a proto je medián středem těchto dvou čísel.

Začneme seřazením čísel od nejmenšího po největší. Poté použijeme ořezávací (celočíselné) dělení k nalezení indexové pozice prostředního čísla, které extrahujeme a uložíme jako medián. Je-li počet čísel sudý, vypočítáme medián jako střed dvou prostředních čísel.

```
def calculate_std_dev(numbers, mean):
    total = 0
    for number in numbers:
        total += ((number - mean) ** 2)
    variance = total / (len(numbers) - 1)
    return math.sqrt(variance)
```

Směrodatná odchylka je mírou rozptylu vyjadřuje, jak dalece se čísla odlišují od středu. Tato funkce vypočítá směrodatnou odchylku pomocí vzorce $s = \sqrt{\frac{\Sigma(x - \bar{x})^2}{n-1}}$, kde x je každé číslo, $\bar{x}$ je střed a n je počet čísel.

```
def print_results(count, statistics):
    real = "9.2f"

    if statistics.mode is None:
        modeline = ""
    elif len(statistics.mode) == 1:
        modeline = "modus      = {0:{fmt}}\n".format(
                   statistics.mode[0], fmt=real)
    else:
        modeline = ("modus      = [" + ", ".join(["{0:.2f}".format(m)
                   for m in statistics.mode]) + "]\n")
    print("""\
počet      = {0:6}
střed      = {mean:{fmt}}
medián     = {median:{fmt}}
{1}\
sm. odch.  = {std_dev:{fmt}}""".format(
    count, modeline, fmt=real, **statistics._asdict()))
```

str.format() ➤ 83

Větší část této funkce je zaměřena na naformátování seznamů modů do řetězce `modeline`. Pokud nejsou k dispozici žádné mody, řádek s modem se vůbec nevypíše. Pokud existuje jeden modus, obsahuje seznam `mode` pouze jeden prvek (`mode[0]`), který vypíšeme pomocí stejného formátu, jaký používá-

me pro ostatní statistické údaje. Pokud máme více modů, vypíšeme je jako seznam, přičemž každý naformátujeme náležitým způsobem. K tomu využíváme seznamovou komprehenzi, která vytvoří seznam řetězců s mody, které poté spojíme do seznamu, kde budou odděleny čárkou. Výpis na konci je díky použití pojmenované n-tice a její metody `_asdict()` společně s rozbalením mapování velice snadný. To nám umožňuje přistupovat k statistickým údajům v objektu `statistics` pomocí názvů, a ne pomocí číselných indexů a díky trojitým uvozovkám jazyka Python můžeme rozvrhnout text tak, aby se vypsal srozumitelným způsobem. Vzpomeňte si, že pokud k předání argumentů metodě `str.format()` použijeme rozbalení mapování, můžeme tak učinit pouze jednou a pouze na konci.

Pojmenovaná n-tice ➤ 113

Použití metody `str.format()` s rozbalením mapování ➤ 86

Zde je třeba upozornit na jednu záludnost. Mody vypisujeme jako formátovací prvek `{1}`, za kterým následuje zpětné lomítko. Zpětné lomítko potlačí nový řádek, takže pokud je `mode` prázdný řetězec, prázdný řádek se neobjeví. A právě kvůli potlačení nového řádku musíme umístit symbol „\n“ na konec řetězce `modeline`, je-li neprázdný.

Shrnutí

V této lekci jsme se věnovali všem vestavěným typům jazyka Python představujícím kolekce a také několika typům představujícím kolekce ze standardní knihovny. Probírali jsme typy představující posloupnost (`tuple`, `collections.namedtuple` a `list`), které podporují stejnou syntaxi pro řezání a krokování jako řetězce. Podívali jsme se též na použití operátoru rozbalení posloupnosti (`*`) a stručně jsme se zmínili o hvězdičkových argumentech při volání funkce. Kromě toho jsme se věnovali množinovým typů (`set` a `frozenset`) a typům představujícím mapování (`dict` a `collections.defaultdict`).

Viděli jsme, jak pomocí pojmenovaných n-tic standardní knihovny Pythonu vytvářet jednoduché vlastní datové typy představující n-tice, k jejichž prvkům můžeme přistupovat přes indexovou pozici nebo pohodlněji přes jméno. Viděli jsme také, jak vytvářet „konstanty“ pomocí proměnných, jejichž identifikátory jsou tvořeny jen velkými písmeny.

V souvislosti se seznamy jsme viděli, že s nimi lze provádět úplně vše co s n-ticemi. Seznamy jsou navíc měnitelné, a proto nabízejí podstatně více funkčnosti než n-tice. To zahrnuje metody, které modifikují seznam (např. `list.pop()`), a možnost mít řezy na levé straně přiřazení sloužící pro vkládání, nahrazování a mazání prvků. Seznamy jsou ideální pro uchovávání posloupnosti prvků, zejména tehdy, když potřebujeme rychlý přístup přes indexovou pozici.

Když jsme probírali typy `set` (množina) a `frozenset` (zmrazená množina), řekli jsme si, že mohou obsahovat pouze hashovatelné prvky. Množiny poskytují rychlé testování příslušnosti a jsou užitečné pro odfiltrování duplicitních dat.

Slovníky jsou v určitých ohledech podobné množinám. Jejich klíče totiž musejí být hashovatelné a jedinečné stejně jako prvky v množině. Avšak slovníky uchovávají dvojice klíč-hodnota, jejichž hodnota může být libovolného typu. Věnovali jsme se také metodám `dict.get()` a `dict.setdefault()` a ukázali jsme si, že alternativou k používání těchto metod jsou výchozí slovníky. Slovníky podobně jako množiny poskytují velmi rychlé testování příslušnosti a urychlují přístup přes klíče.

Seznamy, množiny a slovníky nabízejí kompaktní syntaxe pro komprehenze, s jejichž pomocí lze vytvářet kolekce těchto typů z iterovatelných objektů (které samy mohou být komprehenzemi)

a s možností připojit podmínky. Funkce `range()` a `zip()` se často používají při vytváření kolekcí, ať už v tradičních cyklech `for ... in` nebo v komprehenzích.

Prvky lze z měnitelných typů představujících kolekce mazat pomocí příslušných metod (např. `list.pop()` a `set.discard()`) nebo pomocí příkazu `del` (např. příkaz `del d[k]` vymaže prvek s klíčem `k` ze slovníku `d`).

Odkazy na objekty používané v jazyku Python činí přiřazení extrémně efektivní. Na druhou stranu to ale znamená, že se objekty při použití přiřazovacího operátoru (=) nezkopírují. Viděli jsme rozdíly mezi mělkým a hloubkovým kopírováním a později jsme si ukázali, jak lze seznamy mělce kopírovat pomocí řezu celého seznamu (`L[:]`) a jak lze slovníky mělce kopírovat pomocí metody `dict.copy()`. Libovolný kopírovatelný objekt je možné zkopírovat prostřednictvím funkcí z modulu `copy`. Funkce `copy.copy()` provádí mělkou kopii a funkce `copy.deepcopy()` provádí hloubkovou kopii.

Seznámili jsme se s vysoce optimalizovanou funkci Pythonu s názvem `sorted()`. Tato funkce se používá ve spoustě programů napsaných v jazyku Python, protože Python nenabízí žádné datové typy představující skutečně uspořádanou kolekci. Když tedy potřebujeme procházet kolekci v seřazeném pořadí, sáhneme po metodě `sorted()`.

Vestavěné datové typy jazyka Python představující kolekce (n-tice, seznamy, množiny, zmrazené množiny a slovníky) jsou samy o sobě pro většinu úloh dostatečné. Nicméně ve standardní knihovně je k dispozici ještě několik dalších typů představujících kolekce a spoustu jich nabízejí balíčky třetích stran.

Často potřebujeme číst kolekce ze souborů nebo je do souborů zapisovat. V této lekci jsme se v našem velmi stručném textu ohledně práce se soubory zaměřili jen na čtení a zapisování řádků textu. Práci se soubory se budeme podrobně věnovat v lekci 7 a s dodatečnými prostředky zajišťujícími persistenci dat se seznámíme v lekci 12.

V následující lekci se blíže podíváme na řídicí struktury jazyka Python a představíme si jednu, se kterou jsme se dosud nesetkali. Kromě toho se podrobněji podíváme na zpracování výjimek a na několik dalších příkazů, jako je například příkaz `assert`, kterému jsme se dosud nevěnovali. Dále budeme probírat tvorbu vlastních funkcí a podíváme se zvláště na neuvěřitelně všestranné možnosti práce s argumenty v jazyku Python.

Cvičení

1. Upravte externí program `external_sites.py` tak, aby používal výchozí slovník. Jedná se o snadnou změnu vyžadující dodatečný příkaz import a úpravu pouhých dvou dalších řádků. Řešení najdete v souboru `external_sites_ans.py`.

2. Upravte program `uniquewords2.py` tak, aby nevypisoval slova v abecedním pořadí, ale aby je seřadil podle četnosti jejich výskytu. Bude nutné projít všechny prvky slovníku a vytvořit malou dvouřádkovou funkci pro extrahování hodnoty každého prvku, kterou předáte jako klíčovou funkci funkce `sorted()`. Dále bude nutné náležitým způsobem změnit volání funkce `print()`. Není to sice obtížné, ale malinko zákeřné. Řešení najdete v souboru `uniquewords_ans.py`.

3. Upravte program `generate_usernames.py` tak, aby vypisoval údaje dvou uživatelů na jednom řádku s tím, že jména omezíte na 17 znaků, po každých 64 řádcích vypíšete znak posunu na další stránku (Form Feed) a na začátku každé stránky se vypíše záhlaví sloupců. Výsledný výstup by měl vypadat takto:

```
Jméno               ID   Uživ. jm.  Jméno               ID   Uživ. jm.
----------------- ------ ---------  ----------------- ------ ---------
Battaile, Kierah. (6201) kbattail   Blakey, Kelci-Lou (0641) kblakey
Blenkinsop, Keiri (4541) kblenkin   Clifford, Rebbekk (6263) rcliffor
Collyer, Grey.... (6285) gcollyer   Cowthwaite, Asir. (8866) acowthwa
```

Tento úkol je opravdu náročný. Budete muset uchovávat záhlaví sloupců v proměnných, abyste jej mohli v případě potřeby vypsat. Dále budete muset upravit specifikace formátu tak, aby vyhovovaly užším jménům. Jedna z možností, jak realizovat stránkování, spočívá v zapsání všech vypisovaných prvků do seznamu, který potom projdete pomocí krokování, díky kterému získáte prvky pro levou a pravou stranu, a pomocí funkce zip(), díky které je zase spárujete. Řešení najdete v souboru `generate_usernames_ans.py`. Delší ukázkový datový soubor máte k dispozici v souboru `data/users2.txt`.

LEKCE 4

Řídicí struktury a funkce

V této lekci:

- Řídicí struktury
- Zpracování výjimek
- Vlastní funkce

V prvních dvou částech této lekce se zaměříme na řídicí struktury jazyka Python, přičemž v první části se podíváme na větvení a cykly a ve druhé na zpracování výjimek. S většinou řídicích struktur a základů zpracování výjimek jsme se seznámili v lekci 1, avšak zde své znalosti prohloubíme a přidáme k nim další syntaxe řídicích struktur. Kromě toho se naučíme, jak vyvolávat výjimky a jak vytvářet své vlastní výjimky.

Třetí a nejobsáhlejší část je věnována tvorbě vlastních funkcí, přičemž detailně prostudujeme extrémně všestranné zpracování argumentů v jazyku Python. Díky vlastním funkcím můžeme požadovanou funkčnost zabalit a parametrizovat, což vede ke snížení velikosti našeho kódu, protože nemusíme psát znovu stejný kód, který navíc můžeme opětovně používat. (V následující lekci se podíváme, jak můžeme vytvářet své vlastní moduly, abychom tak mohli své funkce využít i v jiných programech.)

Řídicí struktury

Jazyk Python poskytuje podmíněné větvení prostřednictvím příkazů `if` a cykly prostřednictvím příkazů `while` a `for ... in`. Python nabízí také *podmíněné výrazy*, což je jistý druh příkazu `if`, který je odpovědí Pythonu na ternární operátor (`?:`) používaný v jazycích ve stylu jazyka C.

Podmíněné větvení

Jak jsme viděli v lekci 1, obecná syntaxe pro příkazy podmíněného větvení vypadá v jazyku Python takto:

```
if logický_výraz1:
    sada1
elif logický_výraz2:
    sada2
...
elif logický_výrazN:
    sadaN
else:
    sada_else
```

Klauzulí `elif` může být nula nebo více a poslední klauzule `else` je volitelná. Pokud chceme vzít v potaz určitý případ, u kterého však nechceme provést žádnou akci, pak můžeme místo příslušné sady použít klíčové slovo `pass` (které znamená „nedělej nic“).

V některých případech můžeme zkrátit příkaz `if ... else` na jediný *podmíněný výraz*. Syntaxe pro podmíněný výraz vypadá následovně:

```
výraz1 if logický_výraz else výraz2
```

Pokud se *`logický_výraz`* vyhodnotí na `True`, bude výsledkem podmíněného výrazu *`výraz1`*. V opačném případě bude výsledkem *`výraz2`*.

Programátoři často přiřazují nějaké proměnné výchozí hodnotu, kterou pak v případě potřeby změní, třeba na základě požadavku uživatele nebo podle platformy, na které daný program běží. Zde je řešení pomocí konvenčního příkazu `if`:

```
offset = 20
if not sys.platform.startswith("win"):
    offset = 10
```

Proměnná `sys.platform` uchovává název aktuální platformy (např. „win32“ nebo „linux2“). Stejného výsledku dosáhneme pomocí podmíněného výrazu na jediném řádku:

```
offset = 20 if sys.platform.startswith("win") else 10
```

Žádné závorky zde nepotřebujeme, pokud je však použijeme, vyhneme se zákeřným chybám. Představte si například, že chceme nastavit proměnnou `width` na 100 plus 10, má-li proměnná `margin` hodnotu `True`. Celý příkaz bychom mohli zapsat takto:

```
width = 100 + 10 if margin else 0     # ŠPATNĚ!
```

Upozornění: Tento příkaz ovšem funguje správně jen tehdy, když má proměnná `margin` hodnotu `True`. Pokud je ale její hodnota `False`, pak se proměnná `width` nenastaví na `100`, ale na `0`. To je dáno tím, že Python považuje `100 + 10` součást *`výrazu1`* podmíněného příkazu. Řešení spočívá v použití závorek:

```
width = 100 + (10 if margin else 0)
```

Díky závorkám bude kód také čitelnější pro lidské oko.

Podmíněné výrazy můžeme použít pro zlepšení zpráv vypisovaných pro uživatele. Například při oznamování počtu zpracovaných souborů můžeme místo „0 soubor(y/ů)“, „1 soubor(y/ů)“ a tak podobně použít několik podmíněných výrazů:

```
print("{0} soubor{1}".format((count if count != 0 else "žádný"),
                             ("y" if count in range(2,5) else
                                 ("" if count in range (0,2) else "ů"))))
```

Tento příkaz vypíše „žádný soubor“, „1 soubor“, „2 soubory“, „5 souborů“ a tak podobně, což působí mnohem profesionálněji.

Cykly

Jazyk Python poskytuje cykly `while` a `for ... in`, které oproti tomu, co jsme si ukázali v lekci 1, nabízejí mnohem sofistikovanější syntaxi.

Cyklus while

Zde je kompletní obecná syntaxe cyklu `while`:

```
while logický_výraz:
    sada_while
else:
    sada_else
```

Klauzule `else` je volitelná. Dokud má *`logický_výraz`* hodnotu `True`, provádí se sada bloku `while`. Pokud má *`logický_výraz`* hodnotu `False` (ať už zpočátku nebo po některé z iterací), cyklus se ukončí a provede se sada klauzule `else`, je-li uvedena. Pokud se uvnitř sady bloku `while` provede příkaz `continue`, vrátí se řízení ihned na začátek cyklu a znovu se vyhodnotí *`logický_výraz`*. Pokud se cyklus neukončí normálním způsobem, pak se případná klauzule `else` přeskočí.

Volitelná klauzule `else` je pojmenována poněkud matoucím způsobem, protože její sada se provede vždy, když se příslušný cyklus ukončí normálním způsobem. V případě, když se cyklus přeruší příkazem `break` nebo `return` (je-li cyklus ve funkci či metodě) nebo když dojde k vyvolání výjimky, pak se sada klauzule `else` neprovede. (Pokud dojde k výjimce, přeskočí Python klauzuli `else` a vyhledá vhodnou obsluhu výjimky – tomuto tématu se budeme věnovat v následující části.) Na druhou stranu je chování klauzule `else` stejné u cyklů `while`, u cyklů `for ... in` a u bloků `try ... except`.

Podívejme se na ukázku klauzule `else` v akci. Metody `str.index()` a `list.index()` vracejí indexovou pozici zadaného řetězce či prvku nebo vyvolají výjimku `ValueError`, není-li řetězec či prvek nalezen. Metoda `str.find()` provádí totéž, avšak v případě chyby nevyvolá výjimku, ale vrátí index -1. Pro seznamy žádná podobná metoda neexistuje, můžeme ji ale napsat s využitím cyklu `while`:

```
def list_find(lst, target):
    index = 0
    while index < len(lst):
        if lst[index] == target:
            break
        index += 1
    else:
        index = -1
    return index
```

Tato funkce hledá v zadaném seznamu cílový prvek. Je-li nalezen, příkaz `break` ukončí cyklus, což způsobí vrácení příslušné indexové pozice. Není-li nalezen, cyklus se dokončí a ukončí normálním způsobem. Po normálním ukončení se provede sada `else`, v níž se indexová pozice nastaví na hodnotu -1.

Cyklus for

Také syntaxe cyklu `for` obsahuje stejně jako v případě cyklu `while` volitelnou klauzuli `else`:

```
for výraz in iterovatelný_objekt:
    sada_for
else:
    sada_else
```

Výrazem je buď obyčejná proměnná, nebo posloupnost proměnných uvedených obvykle ve formě n-tice. Pokud se ve výrazu použije n-tice nebo seznam, pak se každý prvek rozbalí do příslušných prvků.

Pokud se uvnitř sady bloku `for ... in` provede příkaz `continue`, vrátí se řízení ihned na začátek cyklu a spustí se nová iterace. Pokud cyklus doběhne do svého konce, pak provede případná klau-

zule `else`. V případě, když se cyklus přeruší příkazem `break` nebo `return` (je-li cyklus ve funkci či metodě) nebo když dojde k vyvolání výjimky, pak se sada klauzule `else` neprovede. (Pokud dojde k výjimce, přeskočí Python klauzuli `else` a vyhledá vhodnou obsluhu výjimky – tomuto tématu se budeme věnovat v následující části.)

Zde je verze funkce `list_find()` s cyklem `for ... in` a podobně jako v případě cyklu `while` také s klauzulí `else`:

enume-
rate()
➢ 140

```
def list_find(lst, target):
    for index, x in enumerate(lst):
        if x == target:
            break
    else:
        index = -1
    return index
```

Jak je z tohoto úryvku kódu patrné, proměnné vytvořené uvnitř výrazu cyklu `for ... in` existují i po ukončení cyklu. Podobně jako všechny lokální proměnné přestanou existovat na konci svého uzavírajícího oboru platnosti.

Zpracování výjimek

Python signalizuje chyby a výjimečné stavy vyvoláváním výjimek, i když některé knihovny Pythonu od třetích stran používají poněkud staromódnější techniky, jako jsou „chybové“ návratové hodnoty.

Zachytávání a vyvolávání výjimek

Výjimky se zachytávají pomocí bloků `try ... except`, jejichž obecná syntaxe vypadá takto:

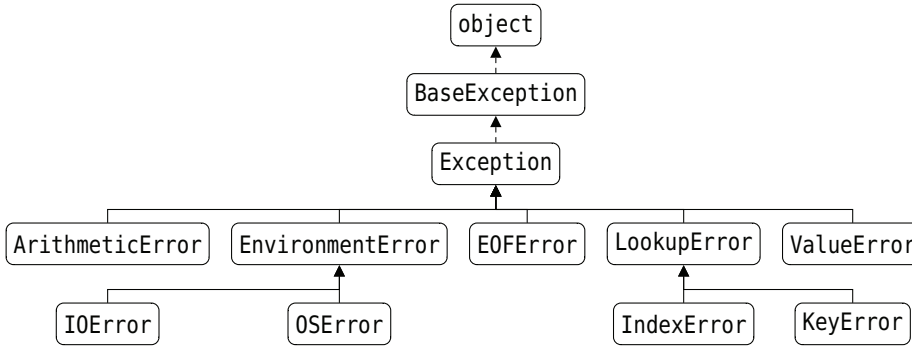
```
try:
    sada_try
except skupina_výjimek1 as proměnná1:
    sada_except1
...
except skupina_výjimekN as proměnnáN:
    sada_exceptN
else:
    sada_else
finally:
    sada_finally
```

Alespoň jeden blok `except` je povinný, bloky `esle` a `finally` jsou však volitelné. Sada bloku `else` se provede při normálním dokončení bloku `try`, při vzniku výjimky se však neprovede. Je-li uveden blok `finally`, pak se vždy nakonec provede.

Každá skupina výjimek klauzule `except` může být jedinou výjimkou nebo n-ticí výjimek uzavřenou do závorek. U každé skupiny je část `as` *`proměnná`* volitelná. Je-li uvedena, pak zadaná proměnná obsahuje vyvolanou výjimku, k níž lze takto přistupovat v sadě bloku `except`.

Pokud v sadě bloku `try` dojde k výjimce, pak se postupně zkouší jednotlivé klauzule `except`. Pokud se výjimka shoduje se skupinou výjimek, provede se odpovídající sada. Shoda se skupinou výjimek vyžaduje, aby výjimka byla stejného (či některého) typu jako typy výjimek uvedené ve skupině nebo stejného (či některého) typu jako podtřídy typů výjimek v dané skupině.*

Pokud například dojde při vyhledávání ve slovníku k výjimce `KeyError`, pak dojde ke shodě s první klauzulí `except`, která obsahuje třídu `Exception`, protože `KeyError` je (nepřímou) podtřídou třídy `Exception`. Není-li v žádné skupině uvedena třída `Exception` (což je obvyklé), ale jedna obsahuje třídu `LookupError`, pak dojde ke shodě se třídou `KeyError`, protože třída `KeyError` je podtřídou třídy `LookupError`. A pokud žádná skupina neobsahuje třídu `Exception` ani `LookupError`, ale jedna obsahuje třídu `KeyError`, pak dojde ke shodě s touto skupinou. Obrázek 4.1 znázorňuje část hierarchie výjimek.



Obrázek 4.1: Část hierarchie výjimek v Pythonu

Zde je ukázka nesprávného použití:

```
try:
    x = d[5]
except LookupError:     #     ŠPATNÉ POŘADÍ
    print("Došlo k chybě při hledání")
except KeyError:
    print("Neplatný klíč")
```


Upozornění: Pokud slovník d nemá prvek s klíčem 5, pak nechceme zachytit obecnější výjimku `LookupError`, ale tu nejspecifičtější (`KeyError`). Zde se však blok `except KeyError` nikdy neprovede. Pokud dojde k výjimce `KeyError`, nastane shoda s blokem `except LookupError`, protože `LookupError` je bázovou třídou třídy `KeyError`, což znamená, že třída `LookupError` je v hierarchii výjimek nad třídou `KeyError`. Pokud tedy používáme více bloků `except`, musíme je vždy uspořádat od nejspecifičtějších (nejnižší pozice v hierarchii) až po ty nejméně specifické (nejvyšší pozice v hierarchii).

* Jak uvidíme v lekci 6, v objektově orientovaném programování se běžně pracuje s hierarchií tříd, což znamená, že jedna třída (datový typ) dědí od jiné. V jazyku Python je na začátku této hierarchie třída `object`. Každá jiná třída dědí od této třídy nebo od jiné třídy, která dědí od třídy `object`. Podtřída je třída, která dědí od jiné třídy, takže všechny třídy Pythonu (kromě třídy `object`) jsou podtřídami, protože všechny jsou odvozeny od třídy `object`.

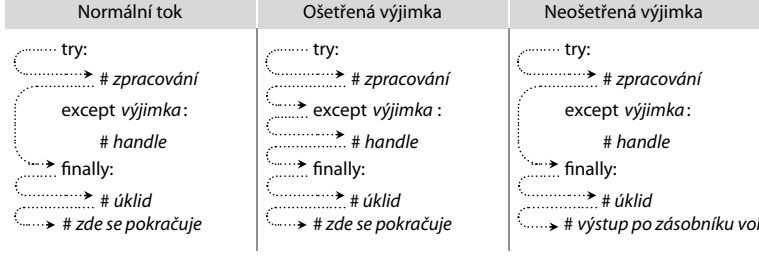
```
try:
    x = d[k / n]
except Exception:     #     ŠPATNÝ POSTUP
    print("Něco se stalo")
```

Upozornění: Všimněte si, že použití třídy `Exception` není obvykle vhodným řešením, poněvadž tím dojde ke shodě se všemi výjimkami, což by snadno mohlo zamaskovat logické chyby v našem kódu. V tomto příkladu je naším cílem zachycení výjimky `KeyError`, pokud by ale proměnná n měla hodnotu 0, pak bychom neúmyslně (a mlčky) zachytili výjimku `ZeroDivisionError`.

Upozornění: Můžeme také napsat `except:`, což znamená, že neuvedeme vůbec žádnou skupinu výjimek. Takto uvedený blok `except` zachytí libovolnou výjimku, včetně těch, které jsou odvozeny od `BaseException`, ale ne od `Exception` (na obrázku 4.1 uvedeny nejsou). Jedná se zde o stejný problém jako při použití `except Exception`, avšak ještě horší, a proto byste tento zápis neměli nikdy za normálních okolností používat.

Pokud se žádný blok `except` s výjimkou neshoduje, pokračuje Python vzhůru zásobníkem volání a hledá vhodnou obsluhu výjimky. Pokud žádnou nenalezne, ukončí program a vypíše výjimku a obsah zásobníku volání do konzoly.

Pokud nedojde k žádné výjimce, provede se volitelný blok `else`. V každém případě (tj. pokud nedojde k výjimce, pokud dojde k výjimce, která je zachycena, nebo pokud dojde k výjimce, která je předána dál podle zásobníku volání) se *vždy* provede sada volitelného bloku `finally`. Pokud nedojde k žádné výjimce nebo pokud dojde k výjimce, která je zachycena jedním z bloků `except`, pak se sada bloku `finally` provede na konci. Pokud ale dojde k výjimce, která se neshoduje s žádným blokem `except`, provede se nejdříve sada bloku `finally` a až poté se výjimka předá dál podle zásobníku volání. Tímto způsobem zaručené provádění může být velice užitečné v situaci, kdy potřebujeme zajistit, aby se správně uvolnily přidělené prostředky. Obrázek 4.2 zachycuje řízení toků v obecných blocích `try ... except ... finally`.



Obrázek 4.2: Řízení toku v blocích try ... except ... finally

Zde je finální verze funkce `list_find()`, tentokráte s využitím zachytávání výjimek:

```
def list_find(lst, target):
    try:
        index = lst.index(target)
    except ValueError:
        index = -1
    return index
```

Zde jsme v podstatě použili blok `try ... except` k převedení výjimky na návratovou hodnotu. Stejný postup můžeme použít také k zachycení jednoho typu výjimky a vyvolání jiného, což je technika, s níž se seznámíme za okamžik.

Jazyk Python dále nabízí jednodušší blok `try ... finally`, který je někdy užitečný:

```
try:
    sada_try
finally:
    sada_finally
```

Bez ohledu na to, co se v sadě bloku `try` stane (vyjma pádu programu nebo počítače!), se vždy provede sada bloku `finally`. Podobného efektu jako při použití bloku `try ... finally` lze dosáhnout také pomocí příkazu `with` společně se správcem kontextu (viz Lekce 8).

Bloky `try ... except ... finally` se obvykle používají pro ošetření chyb při práci se soubory. Kupříkladu program `noblanks.py` načte seznam názvů souborů z příkazového řádku a pro každý z nich vytvoří další soubor se stejným názvem, ale s příponou změněnou na `.nb` a se stejným obsahem, který je zbaven prázdných řádků. Zde je funkce tohoto programu s názvem `read_data()`:

```
def read_data(filename):
    lines = []
    fh = None
    try:
        fh = open(filename, encoding="utf8")
        for line in fh:
            if line.strip():
                lines.append(line)
    except (IOError, OSError) as err:
        print(err)
        return []
    finally:
        if fh is not None:
            fh.close()
    return lines
```

Objekt souboru `fh` jsme nastavili na `None`, protože je možné, že volání funkce `open()` selže, do se `fh` nic nepřiřadí (takže zůstane jako `None`) a vyvolá se výjimka. Pokud dojde k některé z námi uvedených výjimek (`IOError` nebo `OSError`), vrátíme po vypsání chybové zprávy prázdný seznam. Všimněte si, že se před vrácením seznamu provede sada bloku `finally`, takže soubor bude bezpečně uzavřen (pokud byl nejdříve úspěšně otevřen).

Chyby za běhu programu ➤ 402

Dále si všimněte, že pokud dojde k chybě kódování, tak se i přesto, že příslušnou výjimku (`UnicodeDecodeError`) nezachytíme, soubor bezpečně uzavře. V takovýchto případech se provede sada bloku `finally` a poté se výjimka předá dál podle zásobníku volání – žádná návratová hodnota zde není, protože funkce se ukončí v důsledku neošetřené výjimky. A vzhledem k tomu, že pro zachycení výjimek souvisejících s chybou kódování zde není žádný vhodný blok `except`, dojde k ukončení programu a vypsání zásobníku volání.

Klauzuli `except` bychom mohli napsat také malinko úsporněji:

```
except EnvironmentError as err:
    print(err)
    return []
```

Tento zápis funguje díky tomu, že `EnvironmentError` je bázovou třídou pro obě třídy `IOError` a `OSError`.

V lekci 8 si ukážeme ještě kompaktnější styl programování, který zajišťuje bezpečné uzavření souborů a nevyžaduje blok `finally`.

Správci kontextu ➤ 357

Vyvolávání výjimek

Výjimky poskytují užitečné prostředky pro změnu toku programu. K tomuto účelu můžeme využít vestavěné výjimky nebo si můžeme vytvořit výjimky vlastní a v případě potřeby vyvolat kteroukoli z těchto výjimek. Pro vyvolávání výjimek existují tři syntaxe:

```
raise výjimka(argumenty)
raise výjimka(argumenty) from původní_výjimka
raise
```

Při použití první syntaxe musí být zadaná výjimka buď jednou z vestavěných výjimek, nebo naše vlastní výjimka odvozená od třídy `Exception`. Pokud předáme výjimce jako argument nějaký text, pak se tento text vypíše, je-li výjimka při svém zachycení vypsána. Druhá syntaxe je variací první. Výjimka se vyvolá jako zřetězená výjimka (viz Lekce 9), která obsahuje výjimku *`původní_výjimka`*, takže tato syntaxe se používá uvnitř sad `except`. Při použití třetí syntaxe, tedy pokud není výjimka uvedena, dojde k opětovnému vyvolání aktuálně aktivní výjimky. Pokud žádná výjimka aktivní není, vyvolá se výjimka `TypeError`.

Zřetězené výjimky ➤ 405

Vlastní výjimky

Vlastní výjimky jsou vývojáři definované datové typy (třídy). Tvorbě tříd se budeme věnovat v lekci 6, vytvoření jednoduchého typu vlastní výjimky je však velice jednoduché, a proto si příslušnou syntaxi ukážeme již nyní:

```
class názevVýjimky(bázováVýjimka): pass
```

Bázovou třídou by měla být třída `Exception` nebo třída, která je odvozena od třídy `Exception`.

Vlastní výjimky se dají použít třeba pro vyskakování z hluboce zanořených cyklů. Pokud máme například objekt typu `table` uchovávající záznamy (řádky), jež obsahují pole (sloupce), která mají více hodnot (prvků), pak bychom mohli požadovanou hodnotu vyhledat pomocí následujícího kódu:

```
found = False
for row, record in enumerate(table):
    for column, field in enumerate(record):
        for index, item in enumerate(field):
            if item == target:
                found = True
```

```
                break
        if found:
            break
    if found:
        break
if found:
    print("nalezeno: ({0}, {1}, {2})".format(row, column, index))
else:
    print("nenalezeno")
```

Výše uvedených 15 řádků kódu komplikuje skutečnost, že musíme vyskakovat z každého cyklu samostatně. Alternativní řešení spočívá v použití vlastní výjimky:

```
class FoundException(Exception): pass

try:
    for row, record in enumerate(table):
        for column, field in enumerate(record):
            for index, item in enumerate(field):
                if item == target:
                    raise FoundException()
except FoundException:
    print("nalezeno: ({0}, {1}, {2})".format(row, column, index))
else:
    print("nenalezeno")
```

Tím se kód zkrátil na 10 řádků, respektive 11, počítáme-li definici výjimky, a je navíc mnohem čitelnější. Je-li prvek nalezen, vyvoláme naši vlastní výjimku, načež se provede sada bloku `except` a blok `else` se přeskočí. A pokud prvek nalezen není, výjimka se nevyvolá, a proto se na konci provede sada `else`.

Podívejme se na další příklad, na němž si ukážeme různé způsoby zpracování výjimek. Všechny úryvky jsou součástí programu `checktags.py`, který čte všechny soubory HTML zadané na příkazovém řádku a provádí jisté jednoduché testy pro ověření, zda značky začínají „<“ a končí „>“ a zda jsou entity ve správném formátu. Program definuje čtyři vlastní výjimky:

```
class InvalidEntityError(Exception): pass
class InvalidNumericEntityError(InvalidEntityError): pass
class InvalidAlphaEntityError(InvalidEntityError): pass
class InvalidTagContentError(Exception): pass
```

Druhá a třetí výjimka je odvozena od první – důvod si vysvětlíme při probírání kódu, který tyto výjimky používá. Funkce `parse()`, která výjimky používá, má více než 70 řádků, a proto si ukážeme pouze ty části, které souvisejí se zpracováním výjimek.

```
fh = None
try:
    fh = open(filename, encoding="utf8")
```

```
    errors = False
    for lino, line in enumerate(fh, start=1):
        for column, c in enumerate(line, start=1):
            try:
```

Kód začíná docela běžným způsobem. Objekt souboru nastavíme na `None` a veškeré zpracování umístíme do bloku `try`. Program čte soubor řádek po řádku a každý řádek po jednotlivých znacích.

Všimněte si, že zde máme dva bloky `try`. Vnější blok `try` používáme pro zpracování výjimek souvisejících s objektem souboru a vnitřní pro zpracování výjimek vzniklých při analýze obsahu souboru.

```
...
                    elif state == PARSING_ENTITY:
                        if c == ";":
                            if entity.startswith("#"):
                                if frozenset(entity[1:]) - HEXDIGITS:
                                    raise InvalidNumericEntityError()
                            elif not entity.isalpha():
                                raise InvalidAlphaEntityError()
...
```

Funkce prochází nejrůznějšími stavy. Kupříkladu po přečtení ampersandu (`&`) vstoupí do stavu PARSING_ENTITY a uloží znaky mezi ampersandem (ale bez něj) a středníkem do řetězce `entity`.

Výše uvedená část kódu se postará o případ, kdy se při čtení entity objeví středník. Je-li entita číselná (začínající znaky „&#“, za nimiž následují šestnáctkové číslice, a končící „;“, například „&#20AC;“), převedeme její číselnou část na množinu, z níž odebereme všechny šestnáctkové číslice. Pokud něco zůstalo, pak je nejméně jeden znak neplatný, a proto vyvolání vlastní výjimku. Je-li entita alfabetická (začínající znakem „&“, za nímž následují písmena, a končící „;“, například „©“) a některý z jejích znaků není alfabetický, pak vyvoláme vlastní výjimku.

Typ `set` ➤ 123

```
...
                except (InvalidEntityError,
                        InvalidTagContentError) as err:
                    if isinstance(err, InvalidNumericEntityError):
                        error = "neplatná číselná entita"
                    elif isinstance(err, InvalidAlphaEntityError):
                        error = "neplatná alfabetická entita"
                    elif isinstance(err, InvalidTagContentError):
                        error = "neplatná značka"
                    print("CHYBA {0} v {1} na řádku {2} ve sloupci {3}"
                          .format(error, filename, lino, column))
                    if skip_on_first_error:
                        raise
...
```

Dojde-li k vyvolání výjimky při analýze, zachytíme ji v tomto bloku `except`. Pomocí bázové třídy `InvalidEntityError` dokážeme zachytit obě výjimky `InvalidNumericEntityError`

isinstance() ➤ 238

a `InvalidAlphaEntityError`. Poté použijeme funkci `isinstance()` pro zjištění, o který typ výjimky se jedná, a pro nastavení příslušné chybové zprávy. Vestavěná funkce `isinstance()` vrací hodnotu `True`, je-li její první argument stejného typu jako typ (nebo kterýkoli z jeho bázových typů) zadaný jako druhý argument.

Pro každou z našich tří výjimek představujících chybu při analýze bychom mohli použít samostatný blok `except`, avšak díky jejich sloučení nemusíme opakovat poslední čtyři řádky (od volání funkce `print()` po příkaz `raise`).

Program lze používat ve dvou režimech. Má-li proměnná `skip_on_first_error` hodnotu `False`, pak program pokračuje kontrolou souboru i po výskytu chyby při jeho analýze. To může u každého souboru vést k výpisu více chybových zpráv. Má-li proměnná `skip_on_first_error` hodnotu `True`, pak se po výskytu chyby a vypsání (jediné) chybové zprávy zavolá příkaz `raise`, který opětovně vyvolá výjimku vzniklou při analýze, kterou pak zachytí vnější blok `try` (určený pro jednotlivé soubory).

```
...
        elif state == PARSING_ENTITY:
            raise EOFError("chybí ';' na konci " + filename)
...
```

U konce analýzy souboru potřebujeme ověřit, zda se nenacházíme uprostřed nějaké entity. Je-li tomu tak, vyvoláme výjimku `EOFError`, což je vestavěná výjimka signalizující konec souboru, které však předáme vlastní chybový text. Stejně jednoduše bychom mohli vyvolat i naši vlastní výjimku.

```
...
    except (InvalidEntityError, InvalidTagContentError):
        pass # již ošetřeno
    except EOFError as err:
        print("CHYBA neočekávaný konec souboru (EOF):", err)
    except EnvironmentError as err:
        print(err)
    finally:
        if fh is not None:
            fh.close()
```

Pro vnější blok `try` jsme použili samostatné bloky `except`, poněvadž námi požadované chování v jednotlivých blocích se značně liší. Máme-li výjimku při analýze, pak víme, že se chybová zpráva již vypsala. Naším úkolem je tedy jen přerušit čtení souboru a přesunout se na další soubor, takže v obsluze výjimky nemusíme dělat vůbec nic. Výjimka `EOFError` může být způsobena skutečně předčasným koncem souboru nebo může jít o námi vyvolanou výjimku. V každém případě vypíšeme chybovou zprávu a text výjimky. Pokud dojde k výjimce `EnvironmentError` (tj. k výjimce `IOError` nebo `OSError`), jednoduše vypíšeme její chybovou zprávu. A nakonec bez ohledu na to, co se stalo, uzavřeme soubor, pokud byl dříve otevřený.

Vlastní funkce

Funkce jsou prostředky, díky kterým můžeme funkčnost zabalit a parametrizovat. V jazyku Python lze vytvářet čtyři druhy funkcí: globální funkce, lokální funkce, lambda funkce a metody.

Každá funkce, kterou jsme dosud vytvořili, byla *globální* funkcí. Globální objekty (včetně funkcí) jsou přístupné z každé části kódu ve stejném modulu (tj. ve stejném souboru `.py`), v němž je daný objekt vytvořen. Ke globálním objektům můžeme přistupovat i z jiných modulů, jak uvidíme v následující lekci.

Lokální funkce (označované též jako vnořené funkce) jsou funkce, které jsou definovány uvnitř jiných funkcí. Tyto funkce jsou viditelné pouze ve funkci, v níž jsou definované. Tento druh funkcí je zvláště užitečný při vytváření malých pomocných funkcí, které nejsou jinde zapotřebí. Poprvé si je ukážeme v lekci 7.

Lambda funkce jsou výrazy, a proto je můžeme vytvářet v okamžiku jejich použití. Ve srovnání s běžnými funkcemi jsou ale značně omezené.

Metody jsou funkce, které jsou spojené s určitým datovým typem a které lze použít pouze ve spojení s tímto datovým typem. Seznámíme se s nimi v lekci 6 při výkladu objektově orientovaného programování.

Jazyk Python nabízí řadu vestavěných funkcí, přičemž stovky (nebo tisíce, počítáme-li všechny metody) dalších přidává standardní knihovna a knihovny třetích stran, a proto funkce, kterou potřebujeme, je již často napsána. Z tohoto důvodu je vždy vhodné nahlédnout do webové dokumentace Pythonu a zjistit, jaké funkce jsou k dispozici. Nalistujte panel „Webová dokumentace".

Webová dokumentace

Ačkoliv tato kniha nabízí důkladný přehled jazyka Python 3, vestavěných funkcí a většiny běžně používaných modulů ze standardní knihovny, webová dokumentace Pythonu poskytuje značné množství referenční dokumentace týkající se jazyka i rozsáhlé standardní knihovny Pythonu. Dokumentace je k dispozici na adrese *docs.python.org* a také v samotném Pythonu.

V systému Windows je dokumentace dodávána ve formátu nápovědy systému Windows. Pro spuštění prohlížeče nápovědy stačí zvolit položku **Start → Všechny programy → Python 3.x → Python Manuals**. Tento nástroj nabízí rejstřík a hledání, což orientaci v dokumentaci značně usnadňuje. Uživatelé Unixu mají dokumentaci ve formátu HTML. Kromě hypertextových odkazů mají k dispozici také nejrůznější rozcestníkové stránky. Dále je na levé straně každé stránky velice užitečná funkce Quick Search (rychlé hledání).

Nejčastěji používaným webovým dokumentem určeným pro nové uživatele je Library Reference (příručka knihovny) a pro zkušené uživatele Global Module Index (seznam globálních modulů). Oba dokumenty obsahují seznamy odkazů na stránky pokrývající celou standardní knihovnu Pythonu. Dokument Library Reference navíc obsahuje odkazy na stránky pokrývající veškeré vestavěné prvky jazyka Python.

Do dokumentace určitě nahlédněte, zvláště do dokumentu Library Reference nebo Global Module Index, abyste viděli, co vše standardní knihovna Pythonu nabízí, a přes nejrůznější

odkazy si vyhledejte téma, které vás zajímá. Získáte tak základní ponětí o tom, co vše je k dispozici, a budete vědět, kde hledat dokumentaci, která vás zajímá. (Stručné shrnutí standardní knihovny Pythonu najdete v lekci 5.)

Nápověda je k dispozici také v samotném interpretu. Pokud zavoláte vestavěnou funkci `help()` bez argumentů, vstoupíte do interaktivního systému nápovědy. Požadované informace získáte tak, že budete postupovat podle uvedených pokynů. Pro návrat zpět do interpretu stačí napsat „q" nebo „quit". Pokud chcete získat nápovědu k určitému modulu nebo datovému typu, můžete zavolat funkci `help()` s argumentem ve formě modulu nebo datového typu. Například `help(str)` vrátí informace o datovém typu `str` včetně všech jeho metod, `help(dict.update)` vrátí informace o metodě `update()` datového typu `dict` a `help(os)` vrátí informace o modulu os (za předpokladu, že byl importován).

Jakmile se s Pythonem blíže seznámíte, bude vám často stačit prosté připomenutí, jaké atributy (tedy např. jaké metody) daný datový typ poskytuje. Tato informace je dostupná prostřednictvím funkce `dir()`. Například `dir(str)` vypíše všechny metody datového typu `str` a `dir(os)` vypíše všechny konstanty a funkce modulu os (opět za předpokladu, že byl importován).

Obecná syntaxe pro vytvoření (globální či lokální) funkce vypadá následovně:

```
def názevFunkce(parametry):
    sada
```

Parametry jsou volitelné, a pokud je jich dva a více, zapisují se jako posloupnost čárkou oddělených identifikátorů nebo jako posloupnost dvojic *identifikátor=hodnota*. Zde je kupříkladu funkce, která vypočítá plochu trojúhelníka pomocí Heronova vzorce:

```
def heron(a, b, c):
    s = (a + b + c) / 2
    return math.sqrt(s * (s - a) * (s - b) * (s - c))
```

Uvnitř funkce je každý z parametrů `a`, `b` a `c` inicializován odpovídající hodnotou, která byla funkci předána jako argument. Při zavolání funkce musíme zadat všechny argumenty, tedy například `heron(3, 4, 5)`. Pokud zadáme méně nebo více argumentů, vyvolá se výjimka `TypeError`. Při tomto způsobu volání funkce říkáme, že se používají *poziční argumenty*, protože každý předaný argument je použit jako hodnota parametru na odpovídající pozici. V tomto případě je tedy při zavolání funkce `a` nastaveno na hodnotu `3`, `b` na hodnotu `4` a `c` na `5`.

Každá funkce v jazyku Python vrací nějakou hodnotu, ačkoliv je naprosto přijatelné (a běžné) tuto návratovou hodnotu ignorovat. Návratovou hodnotou je buď jediná hodnota, nebo n-tice hodnot, přičemž vrácenými hodnotami mohou být také kolekce, takže na to, co lze vracet, prakticky neexistuje žádné omezení. Funkci můžeme opustit na kterémkoliv místě pomocí příkazu `return`. Pokud použijeme příkaz `return` bez argumentů nebo pokud neuvedeme žádný příkaz `return`, funkce vrátí hodnotu `None`. (V lekci 6 se budeme věnovat příkazu `yield`, který lze v určitém druhu funkcí použít místo příkazu `return`.)

Některé funkce mají parametry, pro které může existovat smysluplná výchozí hodnota. Zde je kupříkladu funkce, která počítá písmena v řetězci, přičemž jako výchozí se použijí písmena znakové sady ASCII:

```
def letter_count(text, letters=string.ascii_letters):
    letters = frozenset(letters)
    count = 0
    for char in text:
        if char in letters:
            count += 1
    return count
```

Pro parametr `letters` jsme stanovili výchozí hodnotu pomocí syntaxe *`parametr=výchozí_hodnota`*. Díky tomu můžeme zavolat funkci `letter_count()` s jediným argumentem, například `letter_count("Magda a Lenka")`. V tomto případě bude uvnitř funkce parametr `letters` obsahovat řetězec, který byl zadán jako výchozí hodnota. Tu však můžeme snadno změnit, například pomocí dalšího pozičního argumentu `letter_count("Magda a Lenka", "aeiouAEIOU")` nebo pomocí klíčovaného argumentu (viz následující text) `letter_count("Magda a Lenka", letters="aeiouAEIOU")`.

Syntaxe parametrů neumožňuje uvést za parametry s výchozí hodnotou parametry, které výchozí hodnotu nemají, takže `def bad(a, b=1, c):` nebude fungovat. Na druhou stranu nejsme nuceni předávat argumenty v pořadí, ve kterém jsou uvedeny ve funkci. Místo toho můžeme použít *klíčované argumenty* a předávat každý argument ve tvaru *`název=hodnota`*.

Zde je krátká funkce, která vrací zadaný řetězec nebo jeho zkrácenou verzi se zadaným indikátorem, je-li delší než stanovená délka:

```
def shorten(text, length=25, indicator="..."):
    if len(text) > length:
        text = text[:length - len(indicator)] + indicator
    return text
```

Zde je několik příkladů volání této funkce:

```
shorten("Dvě slůvka")                          # returns: 'Dvě slůvka'
shorten(length=7, text="Dvě slůvka")           # returns: 'Dvě ...'
shorten("Dvě slůvka", indicator="&", length=7) # returns: 'Dvě sl&'
shorten("Dvě slůvka", 7, "&")                  # returns: 'Dvě sl&'
```

Parametry `length` a `indicator` mají výchozí hodnoty, a proto lze některý z nich nebo oba zcela vynechat, přičemž se použije výchozí hodnota. K tomuto dojde při prvním volání. Ve druhém volání používáme pro oba uvedené parametry klíčované argumenty, takže je můžeme uvést v libovolném pořadí. Ve třetím volání mícháme poziční a klíčované argumenty. První argument jsme použili jako poziční (poziční argumenty musejí být vždy před klíčovanými) a poté jsme použili dva klíčované argumenty. Čtvrté volání využívá pouze poziční argumenty.

Panel „Čtení a zápis textových souborů" ➤ 131

Rozdíl mezi povinným a volitelným parametrem spočívá v tom, že parametr s výchozí hodnotou je volitelný (protože Python může použít výchozí hodnotu) a parametr bez výchozí hodnoty je povinný (protože Python neumí hádat). Opatrné používání výchozích hodnot může výsledný kód zjednodušit a volání značně zpřehlednit. Vzpomeňte si, že vestavěná funkce open() má jeden povinný argument (název souboru) a šest volitelných argumentů. Při použití směsice pozičních a klíčovaných argumentů jsme schopni zadat pouze ty argumenty, které nás skutečně zajímají, zatímco ostatní vynecháme. Díky tomu pak můžeme například místo `open(filename, "r", None, "utf8", None, None, True)` napsat `open(filename, encoding="utf8")`. Další výhodou klíčovaných argumentů je to, že činí volání funkcí mnohem čitelnější, zvláště pak v případě argumentů s logickou hodnotou.

> **Upozornění:** Výchozí hodnoty se při svém zadání vytvářejí v okamžiku provedení příkazu `def` (tj. při vytvoření funkce), a *ne* při volání funkce. V případě neměnitelných argumentů, jako jsou čísla nebo řetězce, v tom není žádný rozdíl, ale u měnitelných argumentů zde číhá zákeřná past.

```
def append_if_even(x, lst=[]):     # ŠPATNĚ!
    if x % 2 == 0:
        lst.append(x)
    return lst
```

Při vytvoření této funkce je parametr `lst` nastaven tak, aby ukazoval na nový seznam. A při každém zavolání této funkce bez druhého parametru bude výchozí seznam stejný jako seznam, který byl vytvořen v době vytvoření samotné funkce, a proto se žádný další nový seznam nevytvoří. To však obvykle není chování, které bychom chtěli. Většinou očekáváme, že se při každém volání této funkce bez druhého parametru vytvoří nový prázdný seznam. Zde je nová verze stejné funkce, tentokráte však s použitím správného postupu pro výchozí měnitelné argumenty:

```
def append_if_even(x, lst=None):
    if lst is None:
        lst = []
    if x % 2 == 0:
        lst.append(x)
    return lst
```

Zde se vytváří nový seznam pokaždé, když se tato funkce zavolá bez druhého argumentu. A pokud je tento argument zadán, tak se použije úplně stejně jako v předchozí verzi funkce. Tento postup, kdy je výchozí hodnotou `None` a vytváří se vždy nový objekt, by se měl používat u slovníků, seznamů, množin a u kteréhokoli jiného měnitelného datového typu, který chceme použít jako výchozí argument. Zde je malinko kratší verze naší funkce, která se chová naprosto stejně:

```
def append_if_even(x, lst=None):
    lst = [] if lst is None else lst
    if x % 2 == 0:
        lst.append(x)
    return lst
```

Pomocí podmíněného výrazu můžeme ušetřit řádek kódu u každého parametru, který má měnitelný výchozí argument.

Jména a dokumentační řetězce

Použití kvalitních jmen pro funkci a její parametry výrazným způsobem zpřehledňuje její účel a použití pro ostatní programátory (a také pro nás samotné po určité době od vytvoření funkce). Zde je několik doporučovaných pravidel, která je dobré vzít v potaz.

- Používejte nějaké schéma pro pojmenovávání a používejte jej konzistentně. V této knize používáme pro konstanty *`VELKÁ_PÍSMENA`*, *`TitleCase`* pro třídy (včetně výjimek), *`camelCase`* pro funkce a metody rozhraní GUI (Graphical User Interface neboli grafické uživatelské rozhraní – viz Lekce 15) a *`malá_písmena`* pro vše ostatní.
- U žádného jména nepoužívejte zkratky, nejsou-li standardizovány a běžně používány.
- Jména proměnných a parametrů používejte přiměřeným způsobem: `x` je výborný název pro souřadnici x a `i` je dobré jako počítadlo cyklu, ale jméno by obecně mělo být dostatečně dlouhé, a tudíž popisné. Jméno by nemělo popisovat typ dat, ale jejich význam (např. `amount_due` místo `money`), není-li ovšem použití pro určitý typ generické – viz například parametr text v příkladu s funkcí `shorten()` níže.
- Funkce a metody by měly mít jména, která říkají, co *dělají* nebo co *vracejí* (v závislosti na jejich významu), ale nikdy jak to dělají, protože to se může změnit.

Zde je několik příkladů pojmenování:

```
def find(l, s, i=0):                                    # ŠPATNĚ
def linear_search(l, s, i=0):                           # ŠPATNĚ
def first_index_of(sorted_name_list, name, start=0):    # SPRÁVNĚ
```

Všechny tři funkce vracejí indexovou pozici prvního výskytu jména v seznamu jmen počínaje zadaným počátečním indexem a s použitím algoritmu, který předpokládá, že seznam je již seřazen.

První pojmenování je špatné, protože neříká nic o tom, co se bude hledat, a jeho parametry (patrně) signalizují požadovaný typ (seznam, řetězec, celé číslo), aniž by signalizovaly, co vlastně znamenají. Druhé je také špatné, protože popisuje použitý algoritmus, který se ale může změnit. To sice pro uživatele nehraje žádnou roli, ale může to zmást údržbáře. Třetí pojmenování je dobré, protože jméno funkce říká, co funkce vrátí, a názvy parametrů jasně signalizují, co se očekává.

Žádná z těchto funkcí nedává žádným způsobem najevo, co se stane, není-li jméno nalezeno. Vrátí se třeba hodnota -1 nebo snad dojde k vyvolání výjimky? Tuto informaci je třeba nějakým způsobem zdokumentovat pro uživatele této funkce.

Dokumentaci můžeme do libovolné funkce přidat pomocí *dokumentačního řetězce* (docstring), což je obyčejný řetězec, který je umístěn bezprostředně za řádkem `def` a před začátkem kódu funkce. Zde je kupříkladu funkce `shorten()`, kterou jsme viděli již dříve, ovšem tentokrát úplně celá:

```
def shorten(text, length=25, indicator="..."):
    """Vrátí text nebo ořezanou kopii s připojeným indikátorem
```

```
    text je libovolný řetězec; length je maximální délka vráceného
    řetězce (včetně případného indikátoru); indikátor je řetězec přidaný
    na konec, který signalizuje, že text byl zkrácen

    >>> shorten("Druhá varieta")
    'Druhá varieta'
    >>> shorten("Hlasy z druhé strany ulice", 17)
    'Hlasy z druhé ...'
    >>> shorten("Rádio Malý Škuner", 10, "*")
    'Rádio Mal*'
    """
    if len(text) > length:
        text = text[:length - len(indicator)] + indicator
    return text
```

Není nijak neobvyklé, že je dokumentace funkce či metody delší než samotná funkce. První řádek dokumentačního řetězce se standardně používá pro stručný, jednořádkový popis, pak se uvádí prázdný řádek, za nímž následuje úplný popis, a poté několik příkladů napsaných tak, jak se objeví při interaktivním zadávání. V lekci 5 a 9 si ukážeme, jak použít příklady v dokumentaci funkce k vytvoření testů jednotek.

Rozbalení argumentů a parametrů

Rozbalení posloupnosti ➤ 117

V předchozí lekci jsme viděli, že pro zadání pozičních argumentů můžeme použít operátor rozbalení posloupnosti (`*`). Pokud například chceme vypočítat plochu trojúhelníka a délky jeho stran máme v nějakém seznamu, pak můžeme použít volání `heron(sides[0], sides[1], sides[2])` nebo seznam jednoduše rozbalit a provést mnohem jednodušší volání `heron(*sides)`. A pokud má seznam (nebo jiná posloupnost) více prvků, než kolik má funkce parametrů, můžeme pro extrakci správného počtu argumentů použít řezání.

Operátor rozbalení posloupnosti můžeme použít také v seznamu parametrů funkce. To je užitečné v situaci, kdy potřebujeme vytvořit funkci, která umí přijímat proměnný počet pozičních argumentů. Zde je funkce `product()`, která spočítá součin zadaných argumentů:

```
def product(*args):
    result = 1
    for arg in args:
        result *= arg
    return result
```

Tato funkce má jeden parametr s názvem `args`. Hvězdička před tímto parametrem znamená, že uvnitř funkce bude mít podobu n-tice obsahující libovolné množství zadaných pozičních argumentů. Zde je několik příkladů volání této funkce:

```
product(1, 2, 3, 4)    # args == (1, 2, 3, 4); vrátí: 24
product(5, 3, 8)       # args == (5, 3, 8); vrátí: 120
product(11)            # args == (11,); vrátí: 11
```

Za pozičními argumenty můžeme mít klíčované argumenty jako v níže uvedené funkci, která vypočítá součet zadaných argumentů umocněných na zadanou mocninu:

```
def sum_of_powers(*args, power=1):
    result = 0
    for arg in args:
        result += arg ** power
    return result
```

Tuto funkci můžeme zavolat buď jen s pozičními argumenty, například `sum_of_powers(1, 3, 5)`, nebo s pozičními i klíčovanými argumenty, například `sum_of_powers(1, 3, 5, power=2)`.

Hvězdičku je dále možné použít také jako „parametr" sám o sobě. V této formě se používá pro signalizaci, že za hvězdičkou již nemohou být další poziční argumenty, ale jen klíčované. Zde je upravená verze funkce `heron()`, která nyní přijímá přesně tři poziční argumenty a jeden volitelný klíčovaný argument.

```
def heron2(a, b, c, *, units="m2"):
    s = (a + b + c) / 2
    area = math.sqrt(s * (s - a) * (s - b) * (s - c))
    return "{0} {1}".format(area, units)
```

Zde je několik příkladů jejího volání:

```
heron2(25, 24, 7)                       # vrátí: '84.0 m2'
heron2(41, 9, 40, units="cm2")          # vrátí: '180.0 cm2'
heron2(25, 24, 7, "sq. inches")         # ŠPATNĚ! vyvolá výjimku TypeError
```

Ve třetím volání jsme se pokusili předat čtvrtý poziční argument, což nám ale * neumožnila a způsobila vyvolání výjimky `TypeError`.

Pokud použijeme hvězdičku jako první parametr, pak zcela zabráníme zadávání pozičních argumentů a přinutíme volající použít klíčované argumenty. Zde je takováto (smyšlená) signatura funkce:

```
def print_setup(*, paper="Letter", copies=1, color=False):
```

Funkci `print_setup()` můžeme zavolat bez parametrů, přičemž se použijí výchozí hodnoty, nebo můžeme změnit některé nebo všechny výchozí hodnoty, například `print_setup(paper="A4", color=True)`. Pokud se ale pokusíme použít poziční argumenty, například `print_setup("A4")`, vyvolá se výjimka `TypeError`.

Poziční argumenty funkce můžeme naplnit nejen rozbalením posloupnosti, ale též rozbalením mapování, k čemuž se používá operátor rozbalení mapování (**).* Pomocí operátoru ** můžeme funkci `print_setup()` předat slovník:

```
options = dict(paper="A4", color=True)
print_setup(**options)
```

* Jak jsme viděli v lekci 2, operátor ** funguje při použití jako binární operátor stejně jako funkce `pow()`.

Dvojice klíč-hodnota slovníku `options` se zde rozbalí tak, že se hodnota každého klíče přiřadí parametru, jehož název je stejný jako klíč. Pokud slovník obsahuje klíč, pro který neexistuje odpovídající parametr, vyvolá se výjimka `TypeError`. Jakýkoliv argument, pro který ve slovníku neexistuje odpovídající prvek, se nastaví na svoji výchozí hodnotu. Pokud ale žádnou výchozí hodnotu nemá, vyvolá se výjimka `TypeError`.

Operátor rozbalení mapování můžeme použít také mezi parametry. Díky tomu můžeme vytvářet funkce, které přijímají libovolný počet klíčovaných argumentů. Zde je funkce `add_person_details()`, která přijímá poziční argumenty představující identifikační číslo a příjmení a libovolný počet klíčovaných argumentů:

```
def add_person_details(id, surname, **kwargs):
    print("ID =", id)
    print("    příjmení =", surname)
    for key in sorted(kwargs):
        print("    {0} = {1}".format(key, kwargs[key]))
```

Tuto funkci bychom mohli zavolat pouze se dvěma pozičními argumenty nebo s dodatečnými informacemi, například `add_person_details(83272171, "Dočkal", forename="Martin", age=18)`. Tato možnost nám dává úžasnou flexibilitu. Můžeme tak samozřejmě přijímat proměnný počet pozičních i klíčovaných argumentů současně:

```
def print_args(*args, **kwargs):
    for i, arg in enumerate(args):
        print("poziční argument {0} = {1}".format(i, arg))
    for key in kwargs:
        print("klíčovaný argument {0} = {1}".format(key, kwargs[key]))
```

Tato funkce jen vypíše zadané argumenty. Lze ji zavolat bez argumentů nebo s libovolným počtem pozičních a klíčovaných argumentů.

Přístup k proměnným v globálním oboru platnosti

Někdy je vhodné mít několik globálních proměnných, k nimž přistupuje více funkcí v programu. To je obvykle dobrý postup pro „konstanty“, ne však pro proměnné, i když v případě krátkého jednoúčelového programu to může mít své opodstatnění.

Program `digit_names.py` přijímá z příkazového řádku volitelný jazyk („en“ nebo „cs“) a číslo a vypíše názvy každé ze zadaných cifer. Pokud tedy programu zadáme na příkazovém řádku „123“, vypíše „jedna dvě tři“. Tento program definuje tři globální proměnné:

```
Language = "cs"

ENGLISH = {0: "zero", 1: "one", 2: "two", 3: "three", 4: "four",
           5: "five", 6: "six", 7: "seven", 8: "eight", 9: "nine"}

CZECH = {0: "nula", 1: "jedna", 2: "dvě", 3: "tři", 4: "čtyři",
         5: "pět", 6: "šest", 7: "sedm", 8: "osm", 9: "devět"}
```

Dodržujeme zde konvenci, že název proměnné obsahující jen velká písmena označuje konstantu, a jako výchozí jazyk jsme zvolili češtinu. (Python neumožňuje vytvářet konstanty přímo, ale spoléhá se na to, že programátoři budou respektovat tuto konvenci.) V jiné části programu přistupujeme k proměnné `Language` a používáme ji pro zvolení příslušného slovníku:

```
def print_digits(digits):
    dictionary = ENGLISH if Language == "en" else CZECH
    for digit in digits:
        print(dictionary[int(digit)], end=" ")
    print()
```

Když Python narazí v této funkci na proměnnou `Language`, podívá se do lokálního (funkčního) oboru platnosti, kde ji nenalezne. Pak se tedy podívá do globálního oboru platnosti (souboru `.py`), kde ji konečně nalezne. Klíčovaný argument `end` použitý v prvním volání funkce `print()` je vysvětlen v panelu „Funkce print()".

Funkce print()

Funkce `print()` přijímá libovolný počet pozičních argumentů a tři klíčované argumenty `sep`, `end` a `file`. Všechny klíčované argumenty mají výchozí hodnoty. Výchozí hodnotou parametru `sep` je mezera. Při zadání dvou či více pozičních argumentů se mezi každým z nich vypíše obsah parametru `sep`. Je-li ovšem zadán pouze jeden poziční argument, nemá tento klíčovaný argument žádný význam. Výchozí hodnota parametru `end` je `\n`, což je důvod, proč se na konci každého volání funkce `print()` vypíše nový řádek. Výchozí hodnota parametru `file` je `sys.stdout`, což je standardní výstupní proud, který obvykle představuje konzoliu.

Kterýmkoli klíčovaným argumentům lze místo hodnot výchozích přiřadit hodnoty, které potřebujeme. Například parametr `file` můžeme nastavit na objekt souboru, který je otevřen pro zápis či připojování a oba parametry `sep` a `end` nastavit na jiné řetězce, včetně řetězce prázdného.

Pokud potřebujeme vypsat několik prvků na jednom řádku, pak stačí pro každý prvek zavolat funkci `print()`, nastavit její parametr `end` na vhodný oddělovač a nakonec zavolat funkci `print()` bez argumentů, aby se vypsal nový řádek. Podívejte se třeba na funkci `print_digits()` níže.

Zde je kód z funkce `main()` našeho programu, který dle potřeby mění hodnotu proměnné `Language` a volá funkci `print_digits()`, která vypíše výsledný výstup.

```
def main():
    if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
        print("použití: {0} [en|cs] číslo".format(sys.argv[0]))
        sys.exit()

    args = sys.argv[1:]
    if args[0] in {"en", "cs"}:
        global Language
```

```
        Language = args.pop(0)
    print_digits(args.pop(0))
```

Všimněte si použití příkazu `global`. Tento příkaz říká Pythonu, že proměnná existuje v globálním (souborovém) oboru platnosti a že by přiřazení do této proměnné nemělo způsobit vytvoření lokální proměnné téhož jména, ale mělo by se aplikovat na globální proměnnou,

> **Upozornění:** Pokud bychom nepoužili příkaz `global`, program by sice běžel, ale jakmile by Python v příkazu `if` narazil na proměnnou `Language`, podíval by se do lokálního (funkčního) oboru platnosti, kde by ji nenašel, a proto by vytvořil novou lokální proměnnou s názvem `Language`, takže by globální proměnná `Language` zůstala nedotčená. Tato zákeřná chyba by se projevila pouze v situaci, kdy bychom program spustili s argumentem „en“, protože pak by se vytvořila lokální proměnná `Language` a nastavila by se na hodnotu „en“, zatímco globální proměnná `Language` používaná ve funkci `print_digits()` by zůstala na své původní hodnotě „cs“.

U netriviálního programu je nejlepší globální proměnné vůbec nepoužívat, a pokud ano, tak jedině jako konstanty – zde není zapotřebí používat příkaz `global`.

Lambda funkce

Lambda funkce jsou funkce vytvářené pomocí následující syntaxe:

```
lambda parametry: výraz
```

Generátorové funkce ➤ 273

Parametry jsou volitelné, a jsou-li uvedeny, pak jde o prosté názvy proměnných oddělené čárkou neboli o poziční argumenty, ačkoliv lze i zde použít kompletní syntaxi pro argumenty podporovanou příkazem `def`. Výraz nesmí obsahovat větvení ani cykly (podmíněné výrazy jsou povoleny) a nesmí mít příkaz `return` (nebo `yield`). Výsledek lambda výrazu je anonymní funkce. Při zavolání lambda funkce obdržíme výsledek výpočtu daného výrazu. Je-li výrazem n-tice, pak bychom ji měli uzavřít do závorek.

Zde je jednoduchá lambda funkce pro přidání (či nepřidání) „s“ podle toho, je-li její argument `1`:

```
s = lambda x: "" if x == 1 else "s"
```

Lambda výraz vrátí anonymní funkci, kterou přiřadíme do proměnné `s`. Libovolnou (volatelnou) proměnnou lze zavolat pomocí závorek. Máme-li tedy počet souborů zpracovaných nějakou operací, můžeme vypsat zprávu pomocí funkce `s()` takto: `print("{0} file{1} processed".format(count, s(count)))`.

Lambda funkce se často používají jako klíčové funkce pro vestavěnou funkci `sorted()` a pro metodu `list.sort()`. Představte si, že máme seznam prvků ve formě n-tic s třemi prvky (skupina, číslo, jméno), který chceme seřadit nejrůznějšími způsoby. Zde je ukázka takového seznamu:

```
elements = [(2, 12, "Mg"), (1, 11, "Na"), (1, 3, "Li"), (2, 4, "Be")]
```

Pokud seznam seřadíme, obdržíme tento výsledek:

```
[(1, 3, 'Li'), (1, 11, 'Na'), (2, 4, 'Be'), (2, 12, 'Mg')]
```

Při výkladu funkce `sorted()` jsme viděli, že můžeme zadat klíčovou funkci, která upraví způsob řazení. Pokud například nechceme seznam seřadit přirozeným způsobem podle skupiny, čísla a jména, ale podle čísla a jména, napíšeme malinkou funkci `def ignore0(e): return e[1], e[2]`, kterou pak zadáme jako klíčovou funkci. Tvorba spousty malinkých funkcí, jako je tato, může být nepohodlná, a proto se v těchto případech častěji používá lambda funkce:

`sorted()` ➤ 138, 144

```
elements.sort(key=lambda e: (e[1], e[2]))
```

Zde je klíčovou funkcí `lambda e: (e[1], e[2])`, přičemž `e` je každý prvek (n-tice) v seznamu. Závorky kolem lambda výrazu jsou nezbytné, je-li výraz n-tice a lambda funkci vytváříme jako argument funkce. Stejného výsledku bychom dosáhli pomocí řezání:

```
elements.sort(key=lambda e: e[1:3])
```

Trošku propracovanější verze provádí řazení podle jmen (bez ohledu na velikost písmen) a čísel:

```
elements.sort(key=lambda e: (e[2].lower(), e[1]))
```

Zde máme dva ekvivalentní způsoby tvorby funkce, která vypočítá plochu trojúhelníka pomocí tradičního vzorce $\frac{1}{2} \times$ *základna* $\times$ *výška*:

```
area = lambda b, h: 0.5 * b * h
```

```
def area(b, h):
    return 0.5 * b * h
```

Nyní můžeme zavolat `area(6, 5)` bez ohledu na to, zda jsme funkci vytvořili pomocí lambda výrazu nebo pomocí příkazu `def`, a výsledek bude stále stejný.

Dalším pěkným použití lambda funkcí je vytvoření výchozího slovníku. Vzpomeňte si (viz předchozí Lekce), že pokud přistupujeme k výchozímu slovníku pomocí neexistujícího klíče, vytvoří se vhodný prvek se zadaným klíčem a s výchozí hodnotou. Zde je několik příkladů:

Výchozí slovníky ➤ 135

```
minus_one_dict = collections.defaultdict(lambda: -1)
point_zero_dict = collections.defaultdict(lambda: (0, 0))
message_dict = collections.defaultdict(lambda: "Zpráva není k dispozici")
```

Pokud přistoupíme ke slovníku `minus_one_dict` s neexistujícím klíčem, vytvoří se nový prvek se zadaným klíčem a hodnotou -1. Podobně dopadneme také v případě slovníku `point_zero_dict`, kde je výchozí hodnotou n-tice `(0, 0)`, a u slovníku `message_dict`, kde je výchozí hodnotou řetězec `"Zpráva není k dispozici"`.

Tvrzení

Co se stane, když funkce obdrží argumenty s neplatnými daty? Co se stane, když uděláme chybu v implementaci nějakého algoritmu a provedeme nesprávný výpočet? Nejhorší by bylo, pokud by provádění programu pokračovalo dál bez jakéhokoliv (zjevného) problému. Jednou z možností, jak se takovýmto záludným problémům vyhnout, je psát testy. Na toto téma se stručně podíváme v lekci 5. Další možnost spočívá v ustavení předběžných a následných podmínek, které by v případě nesplnění nahlásily nějakou chybu. V ideálním případě bychom měli použít testování a také ustavit předběžné a následné podmínky.

Předběžné a následné podmínky lze stanovit pomocí příkazů `assert`, které mají následující syntaxe:

```
assert logický_výraz, volitelný_výraz
```

Pokud se *`logický_výraz`* vyhodnotí na hodnotu `False`, vyvolá se výjimka `AssertionError`. Je-li zadán *`volitený_výraz`*, použije se jako argument výjimky `AssertionError`, což je užitečné pro poskytnutí chybové zprávy. Je třeba si ale uvědomit, že tvrzení jsou navržena pro vývojáře, a ne pro koncové uživatele. Problémy, k nimž dochází při běžném používání programu, jako je chybějící soubor nebo neplatný argument na příkazovém řádku, by se měly ošetřit jinými prostředky, jako je vypsání nebo zaprotokolování chybové zprávy.

Zde jsou dvě verze funkce `product()`. Obě jsou stejné v tom, že vyžadují, aby všechny zadané argumenty byly nenulové, přičemž volání s nulovým argumentem považují za chybu programování.

```
def product(*args): # pesimistická
    assert all(args), "Nulový arg."
    result = 1
    for arg in args:
        result *= arg
    return result
```

```
def product(*args): # optimistická
    result = 1
    for arg in args:
        result *= arg
    assert result, "Nulový arg."
    return result
```

„Pesimistická“ verze na levé straně kontroluje všechny argumenty (nebo argumenty až po první nulový argument) při každém volání. „Optimistická“ verze na pravé straně kontroluje pouze výsledek. Koneckonců je-li kterýkoliv argument nulový, pak bude i výsledek nulový.

Pokud některou z těchto funkcí `product()` zavoláme s nulovým argumentem, vyvolá se výjimka `AssertionError` a do chybového proudu (`sys.stderr`, což bývá obvykle konzolu) se vypíše výstup podobný následujícímu:

```
Traceback (most recent call last):
  File "program.py", line 456, in <module>
    x = product(1, 2, 0, 4, 8)
  File "program.py", line 452, in product
    assert result, "Nulový arg."
AssertionError: Nulový arg.
```

Python společně s námi zadanou chybovou zprávou automaticky vypíše informace ze zásobníku volání, které obsahují název souboru, funkci a číslo řádku.

Co uděláme s příkazy `assert` v okamžiku, kdy je program připraven pro veřejné vydání (a projde všemi svými testy a neporuší žádná tvrzení)? Můžeme říci Pythonu, aby příkazy `assert` neprováděl a v podstatě je tak za programu běhu zahazoval. Toho lze docílit spuštěním programu na příkazovém řádku s volbou `-O`, například `python -O program.py`. Další možností je nastavit proměnnou prostředí `PYTHONOPTIMIZE` na hodnotu `0`. Pokud pro naše uživatele nemají dokumentační řetězce žádný význam (což většinou nemají), můžeme použít volbu `-OO`, která prakticky vyjme příkazy `assert` a dokumentační řetězce. Pro nastavení této volby však neexistuje žádná proměnná prostředí. Někteří vývojáři se vydávají jednodušší cestou: vytvoří kopii svého programu se všemi příkazy `assert` deaktivovanými pomocí komentářů a za předpokladu, že projde testem, vydají tuto verzi bez příkazů `assert`.

Příklad: make_html_skeleton.py

V této části spojíme některé z technik probíraných v této lekci a ukážeme si je v kontextu uceleného programu. Velmi malé weby se často vytvářejí a udržují ručně. Jednou z možností, jak si tuto činnost zpříjemnit, je napsat program, jenž dokáže generovat rámcové soubory HTML, které lze později naplnit obsahem. Program make_html_skeleton.py je interaktivní program, který vyzve uživatele k zadání nejrůznějších detailů a poté vytvoří rámcový soubor HTML. Funkce `main()` tohoto programu obsahuje cyklus, takže uživatelé mohou vytvářet jeden skeleton za druhým se společnými daty (např. informacemi o copyrightu), které tak nemusejí zadávat stále znovu. Zde je přepis typické interakce:

```
make_html_skeleton.py

Vytvoření rámce souboru HTML

Zadejte své jméno (pro copyright): Jakub Slonek
Zadejte rok pro copyright [2009]: 2009
Zadejte název souboru: career-synopsis
Zadejte titulek: Přehled praxe
Zadejte popis (volitelné): přehled praxe Jakuba Slonka
Zadejte klíčové slovo (volitelné): dramaturg
Zadejte klíčové slovo (volitelné): herec
Zadejte klíčové slovo (volitelné): aktivista
Zadejte klíčové slovo (volitelné):
Zadejte název souboru s šablonou stylů (optional): style
Uložen rámec career-synopsis.html

Vytvořit další (a/n)? [a]:

Vytvoření rámce souboru HTML

Zadejte své jméno (pro copyright) [Jakub Slonek]:
Zadejte rok pro copyright [2009]:
Zadejte název souboru:
Zrušeno

Vytvořit další (a/n)? [a]: n
```

Všimněte si, že při vytváření druhého rámce jsou dříve zadané hodnoty k dispozici jako výchozí hodnoty, takže je již není nutné psát znovu. Pro název souboru žádná výchozí hodnota není, a proto je po jeho nezadání nově vytvářený rámec zrušen.

Viděli jsme, jak se program používá, a jsme tedy připraveni prostudovat jeho kód. Program začíná dvěma příkazy import:

```
import datetime
import xml.sax.saxutils
```

Modul `datetime` nabízí několik jednoduchých funkcí pro vytváření objektů typu `datetime.date` a `datetime.time`. Modul `xml.sax.saxutils` obsahuje užitečnou funkci `xml.sax.saxutils.escape()`, která přijímá řetězec a vrací jeho ekvivalent se speciálními znaky jazyka HTML („&“, „<“ a „>“) v zakódované podobě („&“, „<“ a „>“).

Dále definujeme tři globální řetězce, které používáme jako šablony.

```
COPYRIGHT_TEMPLATE = "Copyright (c) {0} {1}. All rights reserved."

STYLESHEET_TEMPLATE = ('<link rel="stylesheet" type="text/css" '
                       'media="all" href="{0}" />\n')

HTML_TEMPLATE = """<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" \
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="cs" xml:lang="cs">
<head>
<title>{title}</title>
<!-- {copyright} -->
<meta name="Description" content="{description}" />
<meta name="Keywords" content="{keywords}" />
<meta equiv="content-type" content="text/html; charset=utf-8" />
{stylesheet}\
</head>
<body>

</body>
</html>
"""
```

`str.format()` ➤ 83

Tyto řetězce použijeme jako šablony společně s metodou `str.format()`. V případě řetězce HTML_TEMPLATE jsme pro názvy polí použili místo indexových pozic jména (např. `{title}`). Za okamžik uvidíme, že jejich hodnoty musíme definovat pomocí klíčovaných argumentů.

```
class CancelledError(Exception): pass
```

Definujeme také jednu vlastní výjimku, kterou uvidíme v akci hned v několika funkcích tohoto programu.

Funkci `main()` používáme k inicializaci počátečních informací a k vytvoření cyklu. V každé iteraci má uživatel možnost zadat některé z údajů pro generovanou stránku HTML, přičemž po každém zadání může tento proces ukončit.

```
def main():
    information = dict(name=None, year=datetime.date.today().year,
                       filename=None, title=None, description=None,
                       keywords=None, stylesheet=None)
    while True:
```

```
        try:
            print("\nVytvoření rámce souboru HTML\n")
            populate_information(information)
            make_html_skeleton(**information)
        except CancelledError:
            print("Zrušeno")
        if (get_string("\nVytvořit další (a/n)?", default="a").lower()
            not in {"a", "ano"}):
            break
```

Funkce `datetime.date.today()` vrací objekt typu `datetime.date`, který uchovává dnešní datum. Nám stačí pouze atribut `year`. Všechny ostatní prvky s informacemi nastavujeme na hodnotu `None`, poněvadž pro ně neexistují žádné smysluplné výchozí hodnoty.

Uvnitř cyklu `while` program vypíše nadpis a poté zavolá `populate_information()` se slovníkem `information`, který se uvnitř této funkce aktualizuje. Dále se zavolá funkce `make_html_skeleton()`, která přijímá několik argumentů, jejichž hodnoty nepředáváme explicitně, ale prostřednictvím rozbalení slovníku `information`.

Pokud uživatel zruší zadávání informací například tím, že nezadá povinný údaj, program vypíše „Zrušeno“. Na konci každé iterace (ať už zrušené či nezrušené) je uživatel dotázán, zda si přeje vytvořit další rámec. Pokud odpoví záporně, přerušíme cyklus a program skončí.

```
def populate_information(information):
    name = get_string("Zadejte své jméno (pro copyright)", "name",
                      information["name"])
    if not name:
        raise CancelledError()
    year = get_integer("Zadejte rok pro copyright", "year",
                       information["year"], 2000,
                       datetime.date.today().year + 1, True)
    if year == 0:
        raise CancelledError()
    filename = get_string("Zadejte název souboru", "filename")
    if not filename:
        raise CancelledError()
    if not filename.endswith((".htm", ".html")):
        filename += ".html"
    ...
    information.update(name=name, year=year, filename=filename,
                       title=title, description=description,
                       keywords=keywords, stylesheet=stylesheet)
```

Vynechali jsme kód pro získání titulku, popisu, klíčových slov dokumentu HTML a souboru s šablonou stylů. Ve všech těchto případech používáme funkci `get_string()`, na kterou se podíváme za okamžik. Nyní stačí jen poznamenat, že tato funkce přijímá zprávu s výzvou, „název“ příslušné proměnné (použije se v chybové zprávě) a volitelnou výchozí hodnotu. Podobně funkce `get_integer()`

přijímá zprávu s výzvou, název proměnné, výchozí hodnotu, minimální a maximální hodnoty a informaci, zda je povolena 0.

Rozbalení mapování ➤ 177

Na konci aktualizujeme slovník `information` novými hodnotami pomocí klíčovaných argumentů. Pro každou dvojici *`klíč=hodnota`* je klíč název klíče ve slovníku, jehož hodnota bude nahrazena zadanou hodnotou, přičemž v tomto případě je každá hodnota proměnná s týmž názvem jako odpovídající klíč ve slovníku.

> **Upozornění:** Z teoretického hlediska to může vypadat, že bychom aktualizaci mohli provést také voláním `information.update(locals())`, protože všechny proměnné, které chceme aktualizovat, se nacházejí v lokálním oboru platnosti. Koneckonců často používáme rozbalování mapování s funkcí `locals()` pro předání argumentů metodě `str.format()`. Ve skutečnosti je předávání argumentů metodě `str.format()` pomocí funkce `locals()` obecně bezpečné, protože se použijí pouze ty názvy klíčů, které se používají ve formátovacím řetězci, přičemž ostatní jsou neškodně ignorovány. To ale není stejné jako aktualizace slovníku. Pokud k aktualizaci slovníku použijeme funkci `locals()`, aktualizuje slovník *vším*, co se nachází v lokálním oboru platnosti (tedy včetně samotného slovníku), a ne jen proměnnými, které nás zajímají. Proto není dobré používat pro naplnění či aktualizaci slovníku funkci `locals()`.

Tato funkce nemá žádnou explicitní návratovou hodnotu (takže vrací hodnotu `None`). Může být též ukončena vyvoláním výjimky `CancelledError`, přičemž se výjimka předá až do funkce `main()`, kde se zpracuje.

Nyní se podíváme na funkci `make_html_skeleton()`, kterou si rozdělíme na dvě části.

```
def make_html_skeleton(year, name, title, description, keywords,
                       stylesheet, filename):
    copyright = COPYRIGHT_TEMPLATE.format(year,
                                          xml.sax.saxutils.escape(name))
    title = xml.sax.saxutils.escape(title)
    description = xml.sax.saxutils.escape(description)
    keywords = ",".join([xml.sax.saxutils.escape(k)
                         for k in keywords]) if keywords else ""
    stylesheet = (STYLESHEET_TEMPLATE.format(stylesheet)
                  if stylesheet else "")
    html = HTML_TEMPLATE.format(**locals())
```

Pro získání textu s copyrightem voláme na řetězci `COPYRIGHT_TEMPLATE` metodu `str.format()`, které předáváme rok a jméno (vhodně zakódované pro dokument HTML) jako poziční argumenty nahrazující `{0}` a `{1}`. Poté vytváříme kopie textů titulku a popisu zakódované pro dokument HTML.

str.format() ➤ 83

U klíčových slov dokumentu HTML se musíme vypořádat se dvěma případy, které rozlišíme pomocí podmíněného výrazu. Pokud nejsou zadána žádná klíčová slova, nastavíme řetězec `keywords` na prázdný řetězec. V opačném případě použijeme seznamovou komprehenzi pro průchod přes všechna klíčová slova, čímž vytvoříme nový seznam řetězců, z nichž je každý zakódován pro dokument HTML. Tento seznam pak metodou `str.join()` spojíme do jediného řetězce s tím, že jednotlivé prvky oddělíme čárkou.

Text šablony stylů vytváříme podobným způsobem jako text copyrightu, ovšem v kontextu podmíněného výrazu, který zajistí, že text bude prázdný řetězec, pokud nebyla zadána žádná šablona stylů.

Text v proměnné `html` vytváříme z řetězce HTML_TEMPLATE s tím, že data pro nahrazovací pole nebereme z pozičních argumentů jako u ostatních šablonových řetězců, ale získáváme je z klíčovaných argumentů. Místo explicitního předávání každého argumentu pomocí syntaxe *`klíč=hodnota`* používáme rozbalení mapování na mapování, které obdržíme z funkce `locals()`. (Další možností je použít volání `format()` ve tvaru `.format(title=title, copyright=copyright` atd.)

Použití metody `str.format()` s rozbalením mapování ➢ 86

```
    fh = None
    try:
        fh = open(filename, "w", encoding="utf8")
        fh.write(html)
    except EnvironmentError as err:
        print("ERROR", err)
    else:
        print("Uložen rámec", filename)
    finally:
        if fh is not None:
            fh.close()
```

Jakmile je kód pro dokument HTML připraven, zapíšeme jej do souboru se zadaným názvem. Uživatele informujeme, že rámec byl uložen nebo že došlo k nějaké chybě. Jako obvykle používáme klauzuli `finally` pro zajištění, aby se soubor vždy uzavřel, pokud byl otevřen.

```
def get_string(message, name="string", default=None,
               minimum_length=0, maximum_length=80):
    message += ": " if default is None else " [{0}]: ".format(default)
    while True:
        try:
            line = input(message)
            if not line:
                if default is not None:
                    return default
                if minimum_length == 0:
                    return ""
                else:
                    raise ValueError("{0} may not be empty".format(
                                     name))
            if not (minimum_length <= len(line) <= maximum_length):
                raise ValueError("{name} musí mít nejméně "
                        "{minimum_length} a nejvíce "
                        "{maximum_length} znaků".format(
                        **locals()))
            return line
        except ValueError as err:
            print("CHYBA", err)
```

Použití metody `str.format()` s rozbalením mapování ➢ 86

Tato funkce má jeden povinný argument `message` a čtyři volitelné argumenty. Je-li zadána výchozí hodnota, uvedeme ji v řetězci se zprávou, aby uživatel viděl hodnotu, kterou obdrží, pokud bez zadání textu stiskne klávesu Enter. Zbývající část funkce je uzavřena do nekonečného cyklu. Z něj lze vyskočit tak, že uživatel zadá platný řetězec nebo stiskem klávesy Enter přijme výchozí hodnotu (je-li zadána). Pokud uživatel udělá nějakou chybu, vypíšeme chybovou zprávu a cyklus pokračuje. Jako obvykle pro předání lokálních proměnných metodě `str.format()` s formátovacím řetězcem obsahujícím pojmenovaná pole nepoužíváme explicitní syntaxi *`klíč=hodnota`*, ale prosté rozbalení mapování na mapování vrácené funkcí `locals()`.

Uživatel může cyklus (a celý program) ukončit také stiskem kláves Ctrl+C, což způsobí vyvolání výjimky `KeyboardInterrupt`. Tu ale žádná z obsluh výjimek v našem programu nezachytí, a proto program skončí a vypíše zásobník volání. Měli bychom uživateli tuto možnost ponechat? Pokud bychom mu ji vzali a v našem programu by byla chyba, zůstal by zaseknutý v nekonečném cyklu s jedinou možností na opuštění programu spočívající v zabití příslušného procesu. Pokud nemáme opravdu pádný důvod zabránit kombinaci kláves Ctrl+C v ukončení programu, pak bychom tuto výjimku neměli zachytávat v žádné obsluze výjimek.

Všimněte si, že tato funkce se netýká pouze programu `make_html_skeleton.py`. Snadno ji můžeme použít i v jiných interaktivních programech tohoto typu. Toto opětovné použití můžeme realizovat zkopírováním a vložením, což by ale vedlo k naprosto nezvladatelné údržbě. V následující lekci si ukážeme, jak vytvářet vlastní moduly s funkčností, kterou je možné sdílet mezi libovolným počtem programů.

```
def get_integer(message, name="integer", default=None, minimum=0,
                maximum=100, allow_zero=True):
...
```

Tato funkce je svou strukturou tak podobná funkci `get_string()`, že by bylo zbytečné ji zde uvádět. (Najdete ji samozřejmě ve zdrojovém kódu, který doprovází tuto knihu.) Parametr `allow_zero` lze použít pro situace, kdy to, že 0 není platná hodnota, chceme zvlášť zdůraznit.

Posledním příkazem v programu je obyčejné volání funkce `main()`. Celý program má něco málo přes 150 řádků a demonstruje několik prvků jazyka Python, s nimiž jsme se seznámili v této a předchozích lekcích.

Shrnutí

V této lekci jsme probírali kompletní syntaxi jazyka Python pro řídicí struktury. Kromě toho jsme si ukázali, jak vyvolávat a zachytávat výjimky a jak vytvářet vlastní typy výjimek.

Větší část lekce byla věnována vlastním funkcím. Ukázali jsme si, jak vytvářet funkce, a uvedli jsme několik doporučovaných pravidel pro pojmenování funkcí a jejich parametrů. Viděli jsme také, jak k funkcím přidat dokumentaci. Podrobně jsme prostudovali všestrannou syntaxi jazyka Python pro parametry a předávání argumentů, a to včetně fixního a proměnného počtu pozičních a klíčovaných argumentů a výchozích hodnot pro argumenty neměnitelných a měnitelných datových typů. Stručně jsme si shrnuli rozbalování posloupností operátorem `*` a ukázali jsme si, jak provést rozbalení mapování operátorem `**`. Rozbalení mapování je užitečné zvláště při aplikaci na slovník (nebo jiné

mapování) nebo na mapování vrácené funkcí `locals()` pro předání argumentů klíč-hodnota formátovacímu řetězci metody `str.format()`, který používá pojmenovaná pole.

Pokud potřebujeme přiřadit novou hodnotu globální proměnné uvnitř funkce, musíme ji deklarovat pomocí příkazu `global`, čímž zabráníme Pythonu, aby vytvořil lokální proměnnou a přiřadil novou hodnotu do ní. I když obecně je nejvhodnější používat globální proměnné pouze pro konstanty.

Lambda funkce se často používají jako klíčové funkce nebo v kontextech, kde je nutné předávat funkce jako argumenty. V této lekci jsme si ukázali, jak vytvářet lambda funkce buď jako anonymní funkce, nebo přiřazením do proměnné jako prostředky pro vytvoření malých pojmenovaných jednořádkových funkcí.

V této lekci jsme se též věnovali použití příkazu `assert`. Tento příkaz je velice užitečný pro specifikaci předběžných a následných podmínek, které mají platit při každém použití dané funkce a které mohou být skutečnou pomocí při vytváření robustního kódu a při hledání chyb.

V této lekci jsme prostudovali kompletní základy pro vytváření funkcí, k dispozici však máme ještě několik dalších technik. Patří mezi ně vytváření dynamických funkcí (vytváření funkcí za běhu programu, třeba s implementacemi, které se liší v závislosti na aktuálních okolnostech), na které se podíváme v lekci 5, lokální (vnořené) funkce, kterým se budeme věnovat v lekci 7, a rekurzivní funkce, generátorové funkce a další, které budeme probírat v lekci 8.

Přestože Python nabízí značné množství vestavěných funkčních prvků a velmi rozsáhlou standardní knihovnu, je pravděpodobné, že budeme psát určité funkce, které budou užitečné v mnoha námi vytvářených program. Kopírování a vkládání takovýchto funkcí by vedlo k těžko udržovatelnému kódu. Python naštěstí poskytuje čisté a snadno použitelné řešení: vlastní moduly. V následující lekci se naučíme, jak vytvářet naše vlastní moduly obsahující naše vlastní funkce. Kromě toho si ukážeme, jak importovat funkční prvky ze standardní knihovny i z našich modulů, a stručně se podíváme, co standardní knihovna nabízí, abychom nevymýšleli to, co již máme k dispozici.

Cvičení

Napište interaktivní program, který udržuje seznam řetězců v souborech.

Po spuštění by měl program vytvořit seznam všech souborů v aktuálním adresáři, které mají příponu `.lst`. Použijte `os.listdir(".")` pro získání všech souborů a odfiltrujte ty, které nemají příponu `.lst`. Pokud žádné takové soubory neexistují, program by měl vyzvat uživatele k zadání názvu souboru a připojit příponu `.lst`, pokud ji uživatel nezadá. Pokud existuje jeden nebo více souborů `.lst`, měly by se vypsat jako číslovaný seznam začínající od 1. Uživatel by měl být vyzván, aby zadal číslo souboru, který chce načíst, nebo 0, pokud chce zadat název pro nový soubor.

Pokud byl zvolen existující soubor, měly by se načíst jeho prvky. Pokud je prázdný nebo pokud byl zadán nový soubor, program by měl vypsat zprávu „v seznamu nejsou žádné prvky“.

Pokud nejsou k dispozici žádné prvky, program by měl nabídnout dvě možnosti: „Přidat“ a „Konec“. Jakmile má seznam jeden či více prvků, měly by se jeho prvky vypsat s číslováním začínajícím od 1 a uživatel by měl mít k dispozici možnosti „Přidat“, „Vymazat“, „Uložit“ (jen pokud dosud nebyl uložen) a „Konec“. Pokud uživatel zvolí „Konec“ a v seznamu jsou dosud neuložené změny, pak by

měl dostat možnost seznam uložit. Zde je přepis interakce s programem (byla odstraněna většina prázdných řádků a také nadpis „Strážce seznamu“, uváděný nad každým seznamem):

```
Zadejte název souboru: movies

-- v seznamu nejsou žádné prvky --
[P]řidat  [K]onec [p]: p
Přidat prvek: Láska nebeská

1: Láska nebeská
[P]řidat  [V]ymazat  [U]ložit  [K]onec [p]: p
Přidat prvek: Jak na věc

1: Jak na věc
2: Láska nebeská
[P]řidat  [V]ymazat  [U]ložit  [K]onec [p]:
Přidat prvek: Vetřelec

1: Jak na věc
2: Láska nebeská
3: Vetřelec
[P]řidat  [V]ymazat  [U]ložit  [K]onec [p]: s
CHYBA: neplatná volba--zadejte některou z těchto možností: 'PpVvUuKk'
Pro pokračování stiskněte klávesu Enter...
[P]řidat  [V]ymazat  [U]ložit  [K]onec [p]: v
Vymazat prvek (nebo 0 pro zrušení mazání): 3

1: Jak na věc
2: Láska nebeská
[P]řidat  [V]ymazat  [U]ložit  [K]onec [p]: u
Seznam byl uložen do movies.lst, celkem uloženo prvků: 2
Pro pokračování stiskněte klávesu Enter...

1: Jak na věc
2: Láska nebeská
[P]řidat  [V]ymazat  [K]onec [p]:
Přidat prvek: Čtyři svatby a jeden pohřeb

1: Jak na věc
2: Láska nebeská
3: Čtyři svatby a jeden pohřeb
[P]řidat  [V]ymazat  [U]ložit  [K]onec [p]: k
Uložit provedené změny (a/n) [a]:
Seznam byl uložen do movies.lst, celkem uloženo prvků: 3
```

Snažte se, aby byla funkce `main()` relativně malá (menší než 30 řádků), a umístěte do ní hlavní cyklus programu. Napište funkci pro získání nového nebo stávajícího názvu souboru (a ve druhém případě pro načtení jeho prvků) a funkci pro přeložení možností uživateli a pro získání jeho volby. Napište také funkce pro přidání prvku, vymazání prvku, vypsání seznamu (prvků nebo názvů souborů), načtení seznamu a uložení seznamu. Buď zkopírujte funkce `get_string()` a `get_integer()` z programu `make_html_skeleton.py`, nebo napište své vlastní verze.

Při vypisování seznamu nebo názvů souboru vypisujte čísla prvků s použitím šířky pole 1, pokud je k dispozici méně než 10 prvků, nebo 2, pokud je méně než 100 prvků, anebo 3 v ostatních případech.

Prvky udržujte v abecedním pořadí nezávislém na velikosti písmen a sledujte také, zda je seznam „špinavý" (tj. proběhly v něm nějaké změny, které dosud nebyly uloženy). Možnost uložení a výzvu k uložení při ukončování programu nabídněte uživateli pouze tehdy, když je seznam špinavý. Seznam se zašpiní přidáním nebo vymazáním prvku a opětovně očistí svým uložením.

Modelové řešení najdete v souboru `listkeeper.py`, který obsahuje méně než 200 řádků kódu.

LEKCE 5

Moduly

V této lekci:

- Moduly a balíčky
- Přehled standardní knihovny Pythonu

Zatímco pomocí funkcí můžeme rozdělit části kódu tak, aby je bylo možné opětovně použít v rámci daného programu, moduly poskytují prostředky pro shromažďování skupin funkcí (a jak uvidíme v následující lekci také vlastních datových typů) tak, aby je bylo možné použít v libovolném počtu programů. Python dále nabízí možnost vytvářet balíčky, což jsou skupiny modulů seskupené dohromady, obvykle z toho důvodu, že tyto moduly poskytují související funkční prvky, nebo proto, že na sobě vzájemně závisejí.

První část této lekce popisuje syntaxe pro importování funkčních prvků z modulů a balíčků, ať už se standardní knihovny nebo z našich vlastních modulů a balíčků. Ve druhé části se posuneme dále a podíváme se, jak vytvářet vlastní balíčky a moduly. Ukážeme si dva vlastní moduly, z nichž první bude seznamovací a druhý bude demonstrovat, jak vyřešit řadu praktických problémů, jako je nezávislost na platformě a testování.

Webová dokumentace ➤ 171

Druhá část nabízí stručný přehled standardní knihovny Pythonu. Je důležité mít na paměti, co tato knihovna obsahuje, protože používání předem definovaných funkčních prvků značně zrychluje programování. Mnohé moduly standardní knihovny je široce rozšířené, dobře otestované a robustní. Kromě přehledu si pomocí malých příkladů ukážeme několik běžných případů užití. U modulů probíraných v jiných lekcích budou navíc uvedeny křížové odkazy.

Moduly a balíčky

Modul jazyka Python je jednoduše řešeno soubor s příponou `.py`. Modul může obsahovat libovolný kód jazyka Python. Všechny programy, které jsme dosud napsali, byly obsažené v jediném souboru `.py`, a proto se jedná o moduly i programy. Podstatným rozdílem je to, že programy jsou navržené pro spouštění, kdežto moduly jsou navržené pro importování a použití v programech.

Ne všechny moduly mají přidružené soubory `.py`. Například modul `sys` je vestavěný do Pythonu a některé moduly jsou napsané v jiných jazycích (povětšinou v jazyku C). Nicméně větší část knihovny Pythonu je napsána v jazyku Python. Pokud tedy například napíšeme `import collections`, můžeme voláním `collections.namedtuple()` vytvářet pojmenované n-tice a funkčnost, k níž přistupujeme, se nachází v souboru modulu `collections.py`. Z hlediska našeho programu je nepodstatné, ve kterém jazyku je daný modul napsán, protože všechny moduly se importují a používají stejným způsobem.

Pro importování lze použít několik syntaxí:

```
import importovatelný_prvek
import importovatelný_prvek1, importovatelný_prvek2, ..., importovatelný_prvekN
import importovatelný_prvek as preferovaný_název
```

Balíčky ➤ 197

Zde je *importovatelný_prvek* obvykle modulem, jako je `collections`, může však jít také o balíček nebo modul v balíčku, přičemž se každá část odděluje tečkou (`.`), například `os.path`. První dvě syntaxe používáme v rámci této knihy. Jsou nejjednodušší a také nejbezpečnější, protože nehrozí možnost konfliktu názvů, což je dáno tím, že musíme vždy používat plně kvalifikované názvy.

Třetí syntaxe nám umožňuje dát importovanému balíčku nebo modulu název dle vlastní volby. To sice může teoreticky vést ke kolizi názvů, ovšem v praxi se tato syntaxe používá právě proto, aby k ní nedocházelo. Přejmenování je užitečné zejména tehdy, když experimentujeme s různými implementacemi nějakého modulu. Pokud máme například dva moduly `MyModuleA` a `MyModuleB`, které mají stejné rozhraní API (Application Programming Interface – aplikační programovací rozhraní), mohli bychom v programu napsat `import MyModuleA as MyModule` a později hladce přejít k druhému modulu příkazem `import MyModuleB as MyModule`.

Kde bychom měli příkazy `import` umisťovat? Běžně se všechny příkazy `import` umisťují na začátek souborů `.py`, za řádek shebang a před dokumentaci modulu. Jak jsme si již řekli v lekci 1, doporučujeme nejdříve importovat moduly standardní knihovny, poté moduly knihoven třetích stran a nakonec naše vlastní moduly.

Zde je několik dalších syntaxí příkazu `import`:

```
from importovatelný_prvek import objekt as preferovaný_název
from importovatelný_prvek import objekt1, objekt2, ..., objektN
from importovatelný_prvek import (objekt1, objekt2, objekt3, objekt4, objekt5,
                                  objekt6, ..., objektN)
from importovatelný_prvek import *
```

Tyto syntaxe mohou způsobovat konflikty názvů, poněvadž činí importované objekty (proměnné, funkce, datové typy nebo moduly) přímo přístupné. Pokud chceme použít syntaxi `from ... import` pro importování velkého množství objektů, můžeme použít více řádků, a to buď tak, že každý nový řádek kromě posledního potlačíme nebo názvy objektů uzavřeme do závorek, jak ukazuje třetí syntaxe.

Hvězdička (`*`) v poslední syntaxi znamená „importuj vše, co není soukromé“, což v praxi znamená buď to, že se importuje každý objekt v modulu kromě těch, jejichž název začíná podtržítkem, nebo že se importují všechny objekty, jejichž názvy jsou obsaženy v seznamu `__all__` definovaném jako globální proměnná daného modulu.

`__all__` ➤ 198

Zde je několik příkladů použití příkazu `import`:

```
import os
print(os.path.basename(filename)) # bezpečný, plně kvalifikovaný přístup

import os.path as path
print(path.basename(filename))    # riziko kolize názvů s path

from os import path
print(path.basename(filename))    # riziko kolize názvů s path

from os.path import basename
print(basename(filename))         # riziko kolize názvů s basename

from os.path import *
print(basename(filename))         # riziko kolize spousty názvů
```

Syntaxe `from importovatelný_prvek import *` importuje všechny objekty z daného modulu (nebo všechny moduly z daného balíčku), což může znamenat stovky názvů. V případě příkazu `from os.path import *` se importuje téměř 40 názvů, mezi něž patří i `dirname`, `exists` a `split`, což mohou být názvy, které bychom raději použili pro naše vlastní proměnné nebo funkce.

Pokud například napíšeme příkaz `from os.path import dirhame`, pak můžeme přímo zavolat `dirname()` bez kvalifikace. Pokud poté ale ve svém kódu napíšeme `dirname = "."`, bude nyní odkaz na objekt `dirname` místo funkce `dirname()` svázán s řetězcem ".", takže když se pokusíme zavolat `dirname()`, obdržíme výjimku `TypeError`, protože `dirname` nyní ukazuje na řetězec a řetězce nejsou volatelné.

Z hlediska případných kolizí názvů vytvářených syntaxí import * některé vývojářské týmy specifikují ve svých směrnicích, že se může používat pouze syntaxe `import importovatelný_prvek`. Nicméně určité rozsáhlé balíčky, zvláště pak knihovny GUI, se často importují tímto způsobem, protože obsahují obrovské množství funkcí a tříd (vlastních datových typů), jejichž ruční vypisování by bylo velice pracné.

Přirozeně zde vyvstává otázka, jak Python ví, kde importované moduly a balíčky hledat? Vestavěný modul `sys` obsahuje seznam `sys.path`, který uchovává seznam adresářů, které představují *cestu Pythonu* (Python path). První adresář je adresářem, který obsahuje samotný program, a to i tehdy, pokud byl program vyvolán z jiného adresáře. Je-li nastavena proměnná prostředí PYTHONPATH, jsou v ní specifikované další cesty v seznamu. Poslední cesty jsou ty, které jsou nezbytné pro přístup k standardní knihovně Pythonu – nastavují se při instalaci Pythonu.

Upozornění: Při prvním importu vestavěného modulu se Python po tomto modulu podívá postupně v každé cestě uvedené v seznamu `sys.path`. Jedním z důsledků tohoto postupu je to, že pokud vytvoříme modul nebo program se stejným názvem, jako má jeden z knihovních modulů Pythonu, najde se ten náš jako první, což nevyhnutelně způsoby problémy. Aby k tomuto nedošlo, nikdy nevytvářejte program či modul se stejný názvem, jako má některý z knihovních adresářů či modulů Pythonu nejvyšší úrovně. Výjimkou je samozřejmě situace, kdy vytváříte svou vlastní implementaci takovéhoto modulu a záměrně jej přepisujete. (Modul nejvyšší úrovně je modul, jehož soubor `.py` je umístěn v jednom z adresářů v cestě Pythonu, a ne v jednom z jejich podadresářů.) Například v systému Windows obsahuje cesta Pythonu obvykle adresář s názvem `C:\Python31\Lib`, takže na této platformě bychom neměli vytvářet modul s názvem `Lib.py` ani modul se stejným názvem, jako má kterýkoliv modul v adresáři `C:\Python31\Lib`.

Pro rychlý způsob kontroly, zda se nějaký název modulu již používá, stačí vyzkoušet příkaz `import` s tímto názvem. To lze provést v konzole zavoláním interpretu s volbou příkazového řádku `-c` („execute code" – proveď kód), za níž umístíme příslušný příkaz `import`. Pokud například chceme zjistit, zda existuje modul s názvem `Music.py` (nebo adresář nejvyšší úrovně v cestě Pythonu s názvem `Music`), pak můžeme do konzoly napsat následující příkaz:

```
python -c "import Music"
```

Pokud obdržíme výjimku `ImportError`, víme, že se žádný modul nebo adresář nejvyšší úrovně tohoto jména nepoužívá. Jakýkoliv jiný výstup (nebo žádný) znamená, že toto jméno je již zabráno. Tento postup ale naneštěstí nezaručuje, že daný název bude vždy bez problémů, poněvadž můžeme později nainstalovat balíček či modul Pythonu od třetí strany, který obsahuje konfliktní název, i když v praxi se jedná o velmi ojedinělý problém.

Pokud bychom například vytvořili soubor modulu s názvem `os.py`, došlo by ke konfliktu s knihovním modulem `os`. Pokud ale vytvoříme modul s názvem `path.py`, bude to v pořádku, protože bude importován jako modul `path`, kdežto knihovní modul se importuje jako `os.path`. V této knize je v názvech souborů našich vlastních modulů první písmeno vždy velké, čímž se vyhneme konfliktům názvů (alespoň na Unixu), protože názvy souborů modulů standardní knihovny obsahují jen malá písmena.

Program může importovat některé moduly, které následně importují své vlastní moduly, včetně některých, které již byly importovány. To však nezpůsobí žádné problémy. Kdykoliv je totiž nějaký modul importován, tak Python nejdříve zkontroluje, zda již nebyl importován dříve. Pokud nebyl, tak Python spustí zkompilovaný bajt-kód modulu, čímž vytvoří příslušné proměnné, funkce a další objekty daného modulu a interně si poznačí, že tento modul již byl importován. Při každém dalším importu tohoto modulu Python detekuje, že již byl importován, a neudělá nic.

Když Python potřebuje zkompilovaný bajt-kód modulu, automaticky jej vygeneruje. Tím se liší třeba od Javy, kde se kompilace do bajt-kódu musí provést explicitním způsobem. Python se nejdříve podívá po souboru se stejným názvem jako má soubor `.py` daného modulu, ale s příponou `.pyo`, což je optimalizovaná verze se zkompilovaným bajt-kódem modulu. Pokud žádný soubor `.pyo` neexistuje (nebo pokud je starší než soubor `.py`, což znamená, že není aktuální), Python se podívá po souboru s příponou `.pyc`, což je neoptimalizovaná verze se zkompilovaným bajt-kódem modulu. Pokud Python najde aktuální verzi se zkompilovaným bajt-kódem modulu, tak ji načte. V opačném případě načte soubor `.py` a zkompiluje verzi se zkompilovaným bajt-kódem. Tak jako tak bude mít nakonec Python modul v paměti ve formě zkompilovaného bajt-kódu.

Pokud by Python musel zkompilovat soubor `.py`, uloží jej do souboru `.pyc` (nebo `.pyo`, pokud je na příkazovém řádku uvedena volba `-O` nebo pokud je nastavena proměnná prostředí `PYTHONOPTIMIZE`), ovšem za předpokladu, že do příslušného adresáře lze zapisovat. Ukládání bajt-kódu lze zamezit volbou příkazového řádku `-B` nebo nastavením proměnné prostředí `PYTHONDONTWRITEBYTECODE`.

Používání souborů se zkompilovaným bajt-kódem vede k rychlejšímu spouštění, protože interpret nemusí kód načítat, kompilovat, ukládat (je-li to možné) a spouštět, ale stačí mu jen kód načíst a spustit. Je však třeba poznamenat, že na samotný běh kódu to žádný vliv nemá. Při instalaci Pythonu se moduly standardní knihovny obvykle zkompilují do bajt-kódu v rámci instalačního procesu.

Balíčky

Balíček je obyčejným adresářem, který obsahuje skupinu modulů a soubor s názvem `__init__.py`. Představte si, že máme smyšlenou skupinu souborů modulů pro čtení a zapisování nejrůznějších formátů grafických souborů, jako jsou například moduly `Bmp.py`, `Jpeg.py`, `Png.py`, `Tiff.py` a `Xpm.py`, z nichž každý nabízí funkce `load()`, `save()` a tak podobně.* Moduly bychom mohli nechat ve stejném adresáři, v němž je náš program, ale u větších programů, které používají množství vlastních modulů, by se grafické moduly úplně ztratily. Jejich umístěním do samostatného podadresáře (např. `Graphics`) je budeme mít pěkně pohromadě. A pokud k nim do adresáře `Graphics` přidáme prázdný soubor `__init__.py`, stane se z tohoto adresáře balíček:

* Rozsáhlou podporu pro práci s grafickými soubory poskytuje spousta modulů třetích stran, z nichž je nejpozoruhodnější knihovna Python Imaging Library (*www.pythonware.com/products/pil*).

```
Graphics/
    __init__.py
    Bmp.py
    Jpeg.py
    Png.py
    Tiff.py
    Xpm.py
```

Dokud bude adresář `Graphics` podadresářem uvnitř adresáře našeho programu nebo bude uveden v cestě Pythonu, můžeme libovolný z těchto modulů importovat a používat. Musíme ale myslet na to, aby název zastřešujícího modulu (`Graphics`) nebyl stejný jako kterýkoliv z názvů nejvyšší úrovně ve standardní knihovně, jinak by došlo ke konfliktu názvů. (Na systému Unix stačí, když budou mít názvy našich modulů první písmeno velké, neboť všechny moduly standardní knihovny mají názvy s malými písmeny.) Náš modul můžeme importovat a používat takto:

```
import Graphics.Bmp
image = Graphics.Bmp.load("bashful.bmp")
```

U krátkých programů můžeme používat kratší názvy, což lze v Pythonu provést dvěma malinko odlišnými způsoby.

```
import Graphics.Jpeg as Jpeg
image = Jpeg.load("doc.jpeg")
```

Zde jsme importovali modul `Jpeg` z balíčku `Graphics` a řekli jsme Pythonu, že se na něj chceme místo plně kvalifikovaného jména `Graphics.Jpeg` odkazovat jen jako na `Jpeg`.

```
from Graphics import Png
image = Png.load("dopey.png")
```

V tomto úryvku kódu importujeme modul `Png` přímo z balíčku `Graphics`. Při použití této syntaxe (`from ... import`) je modul `Png` přímo přístupný. Nikdo nás nenutí používat v našem kódu názvy používané v balíčku:

```
from Graphics import Tiff as picture
image = picture.load("grumpy.tiff")
```

Zde používáme modul `Tiff`, který jsme ale ve svém programu přejmenovali na modul `picture`.

V některých situacích je výhodné načíst všechny moduly balíčku pomocí jediného příkazu. K tomu je nutné upravit soubor `__init__.py` daného balíčku tak, aby obsahoval příkaz, který stanoví, které moduly se mají načíst. Tento příkaz musí přiřadit seznam názvů modulů do speciální proměnné `__all__`. Zde je například požadovaný řádek pro soubor `Graphics/__init__.py`:

```
__all__ = ["Bmp", "Jpeg", "Png", "Tiff", "Xpm"]
```

Nic víc není třeba, i když do souboru `__init__.py` samozřejmě můžeme umístit libovolný další kód. Nyní můžeme použít další typ příkazu `import`:

```
from Graphics import *
image = Xpm.load("sleepy.xpm")
```

Syntaxe `from balíček import *` přímo importuje všechny moduly uvedené v seznamu `__all__`. Po provedení tohoto příkazu je přímo přístupný nejen modul `Xpm`, ale také všechny ostatní moduly.

Jak jsme si řekli již dříve, tuto syntaxi lze aplikovat také na modul (tj. `from modul import *`) při tom se importují všechny funkce, proměnné a další objekty definované v zadaném modulu (kromě těch, jejichž název začíná podtržítkem). Pokud chceme mít kontrolu nad tím, co přesně se má při použití syntaxe `from modul import *` importovat, můžeme definovat seznam `__all__` i v samotném modulu. V takovém případě importuje příkaz `from modul import *` pouze ty objekty, jejichž názvy jsou uvedeny v seznamu `__all__`.

Dosud jsme si ukázali pouze jednu úroveň zanoření, Python nám však umožňuje vnořovat balíčky tak hluboko, jak potřebujeme. Můžeme tak mít v adresáři `Graphics` třeba podadresář `Vector` a v něm soubory modulů, například `Eps.py` a `Svg.py`:

```
Graphics/
    __init__.py
    Bmp.py
    Jpeg.py
    Png.py
    Tiff.py
    Vector/
        __init__.py
        Eps.py
        Svg.py
    Xpm.py
```

Adresář `Vector` se stane balíčkem v okamžiku, kdy do něj umístíme soubor `__init__.py`, a jak jsme si již řekli, tento soubor může být prázdný nebo může obsahovat seznam `__all__` pro potřeby programátorů, kteří chtějí provést import pomocí příkazu `from Graphics.Vector import *`.

Pro přístup k vnořenému balíčku vycházíme ze syntaxe, kterou jsme již používali:

```
import Graphics.Vector.Eps
image = Graphics.Vector.Eps.load("sneezy.eps")
```

Plně kvalifikovaný název je poněkud delší, čemuž se snaží někteří programátoři zamezit tím, že se snaží udržovat hierarchie svých modulů docela ploché.

```
import Graphics.Vector.Svg as Svg
image = Svg.load("snow.svg")
```

Pro modul můžeme vždy použít náš vlastní krátký název, jak jsme to provedli zde, i když se tím vystavujeme vyššímu riziku konfliktu názvů.

Všechny importy, které jsme dosud používali (a které budeme používat i ve zbývající části knihy), se označují jako *absolutní* importy, což znamená, že každý námi importovaný modul je v některém

z adresářů uvedených v proměnné `sys.path` (nebo podadresářů, pokud název v příkazu import obsahuje jednu či více teček, které slouží v podstatě jako oddělovače v cestě). Při vytváření rozsáhlých balíčků s více moduly a adresáři je často užitečné importovat další moduly, které jsou součástí stejného balíčku. Například v souboru `Eps.py` nebo `Svg.py` můžeme získat přístup k modulu `Png` pomocí konvenčního nebo *relativního* importu:

```
import Graphics.Png as Png
```

```
from ..Graphics import Png
```

Tyto dva úryvky kódu jsou ekvivalentní, protože oba přímo zpřístupňují modul `Png` uvnitř modulu, kde se používají. Všimněte si ale, že relativní importy, což jsou importy, které používají syntaxi `from` *`modul`* `import` s tečkami před názvem modulu (každá tečka představuje jeden krok směrem nahoru v hierarchii adresářů), můžeme použít pouze v modulech, které jsou uvnitř nějakého balíčku. Relativní importy usnadňují přejmenování zastřešujícího balíčku a uvnitř balíčků zamezují nechtěnému importování standardních modulů místo našich vlastních.

Vlastní moduly

Moduly jsou obyčejné soubory s příponou `.py`, a proto je lze vytvářet bez dalších formalit. V této části se podíváme na dva naše vlastní moduly. První modul `TextUtil` (v souboru `TextUtil.py`) obsahuje jen tři funkce: `is_balanced()`, která vrací hodnotu `True`, má-li zadaný řetězec vyvážené závorky nejrůznějších druhů, `shorten()` (viz dříve – strana 175) a `simplify()`, která dokáže z řetězce odříznout nežádoucí bílé místo a další znaky. Při probírání tohoto modulu se podíváme také na to, jak spouštět kód v dokumentačních řetězcích jako testy jednotek.

Druhý modul `CharGrid` (v souboru `CharGrid.py`) uchovává znakovou mřížku a umožňuje do této mřížky „kreslit“ čáry, obdélníky a text a celou mřížku vykreslit do konzoly. V tomto modulu si ukážeme několik technik, se kterými jsme se dosud nesetkali a které jsou typické pro rozsáhlejší a komplexnější moduly.

Modul TextUtil

Struktura tohoto modulu (a většiny dalších) se odlišuje od struktury programu. Prvním řádkem je řádek shebang a poté zde máme komentáře (obvykle copyright a informace o licenci). Dále je obvykle uváděn řetězec s trojitými uvozovkami, který poskytuje přehled obsahu modulu často doplněný ukázkovými příklady – jedná se o dokumentační řetězec modulu. Zde je začátek souboru `TextUtil.py` (ovšem bez řádků s informacemi o licenci):

```
#!/usr/bin/env python3
# Copyright (c) 2008-9 Qtrac Ltd. All rights reserved.
"""
Tento modul nabízí několik funkcí pro manipulaci s řetězci.

>>> is_balanced("(Python (není (jako (lisp))))")
True
>>> shorten("Velká křižovatka", 10)
'Velká k...'
```

```
>>> simplify(" nějaký    text    s  nadbytečnými  mezerami  ")
'nějaký text s nadbytečnými mezerami'
"""

import string
```

Dokumentační řetězec modulu je k dispozici programům (nebo dalším modulům), které tento modul importují, jako `TextUtil.__doc__`. Za dokumentačním řetězcem modulu následují importy a poté zbytek modulu.

Funkci `shorten()` jsme již viděli, a proto ji zde nebudeme opakovat. A protože se spíše než na funkce zaměřujeme na moduly, ukážeme si kromě funkce `simplify()`, kterou si ukážeme celou včetně dokumentačního řetězce, pouze kód funkce `is_balanced()`.

`shorten()` ➤ 175

Takto vypadá funkce `simplify()` rozdělená na dvě části:

```
def simplify(text, whitespace=string.whitespace, delete=""):
    r"""Vrátí text s vícenásobnými mezerami zredukovanými do jediné mezery

    Parametr whitespace je řetězec znaků, z nichž každý
    je považován za mezeru.
    Není-li parametr delete prázdný, měl by obsahovat řetězec, jehož
    znaky se vyhledají ve výsledném řetězci a odstraní.

    >>> simplify(" tohle    a\n také\t tamto")
    'tohle a také tamto'
    >>> simplify("  Vejce   a.s.\n")
    'Vejce a.s.'
    >>> simplify("  Vejce   a.s.\n", delete=",;:.")
    'Vejce as'
    >>> simplify(" nesamohláskový ", delete="aáeiouyý")
    'nsmhlskv'
    """
```

Po řádku s příkazem `def` následuje dokumentační řetězec funkce se známou strukturou: na jednom řádku popis, prázdný řádek, podrobnější popis a pak několik příkladů napsaných tak, jako by byly zadávány interaktivně. Řetězce v uvozovkách jsou v dokumentačním řetězci, a proto je nutné buď potlačit v nich uvedená zpětná lomítka, nebo dokumentační řetězec uzavřít do trojitých uvozovek, což jsme udělali my.

Holé řetězce ➤ 72

```
    result = []
    word = ""
    for char in text:
        if char in delete:
            continue
        elif char in whitespace:
            if word:
```

```
                result.append(word)
                word = ""
        else:
            word += char
    if word:
        result.append(word)
    return " ".join(result)
```

Seznam `result` používáme pro uchovávání „slov“, což jsou řetězce, jež nemají mezery nebo vyškrtnuté znaky (parametr `delete`). Zadaný text procházíme po jednotlivých znacích, přičemž přeskakujeme vyškrtnuté znaky. Narazíme-li na znak považovaný za bílé místo (parametr `whitespace`) a současně vytváříme slovo (proměnná `word`), přidáme toto slovo do seznamu `result` a nastavíme jej na prázdný řetězec. V opačném případě bílé místo přeskočíme. Jakýkoliv jiný znak přidáme do vytvářeného slova. Nakonec vrátíme jediný řetězec všech slov v seznamu `result` spojených jedinou mezerou.

Funkce `is_balanced()` je zapsána podobným způsobem: nejdříve je řádek s příkazem `def`, poté dokumentační řetězec s jednořádkovým popisem, prázdný řádek, podrobnější popis, několik příkladů a pak samotný kód. Zde je kód bez dokumentačního řetězce:

```
def is_balanced(text, brackets="()[]{}<>"):
    counts = {}
    left_for_right = {}
    for left, right in zip(brackets[::2], brackets[1::2]):
        assert left != right, "znaky závorek se musejí lišit"
        counts[left] = 0
        left_for_right[right] = left
    for c in text:
        if c in counts:
            counts[c] += 1
        elif c in left_for_right:
            left = left_for_right[c]
            if counts[left] == 0:
                return False
            counts[left] -= 1
    return not any(counts.values())
```

V této funkci sestavujeme dva slovníky. Klíči slovníku `counts` jsou otevírací znaky („(“, „[“, „{“ a „<“) a jeho hodnotami celá čísla. Klíči slovníku `left_for_right` jsou uzavírací znaky („)“, „]“, „}“ a „>“) a jeho hodnotami odpovídající otevírací znaky. Jakmile jsou slovníky připraveny, procházíme text znak po znaku. Jakmile narazíme na otevírací znak, zvýšíme odpovídající počítadlo (slovník `counts`). A podobně, jakmile narazíme na uzavírací znak, vyhledáme odpovídající uzavírací znak. Je-li počítadlo pro tento znak 0, znamená to, že jsme narazili na uzavírací znak bez odpovídajícího otevíracího znaku, a proto okamžitě vrátíme hodnotu `False`. V opačném případě snížíme příslušné počítadlo. Na konci by mělo mít každé počítadlo hodnotu 0, jsou-li všechny dvojice vyvážené, takže pokud je některé z nich nenulové, funkce vrátí hodnotu `False`. Jinak vrátí hodnotu `True`.

Až dosud bylo vše podobné jako v kterémkoliv jiném souboru `.py`. Pokud by byl soubor `TextUtil.py` programem, jistě by obsahoval několik dalších funkcí a na konci by bylo jediné volání některé z těchto funkcí, které by zahájilo zpracování. Avšak vzhledem k tomu, že se jedná o modul, který má být importován, jsou definované funkce dostačující. Nyní může libovolný program či modul importovat modul `TextUtil` a používat jej:

```
import TextUtil

text = " záhadný   hlavolam "
text = TextUtil.simplify(text)     # text == 'záhadný hlavolam'
```

Pokud chceme modul `TextUtil` zpřístupnit určitému programu, musíme soubor `TextUtil.py` umístit do stejného adresáře, ve kterém se nachází daný program. Pokud chceme modul `TextUtil` zpřístupnit všem svým programům, pak máme k dispozici několik možností. Jednou z nich je umístit modul do podadresáře s balíčky v rámci distribuce Pythonu. V systému Windows se obvykle jedná o adresář `C:\Python31\Lib\site-packages`, v případě systémů Mac OS X a Unix však může být jeho umístění různé. Tento adresář se nachází v cestě Pythonu, takže libovolný v něm umístěný modul bude vždy nalezen. Druhá možnost spočívá ve vytvoření adresáře speciálně pro náš vlastní modul, který chceme používat ve všech svých programech, a v nastavení proměnné prostředí `PYTHONPATH` na tento adresář. Třetí možností je umístit modul do *lokálního* podadresáře `site-packages`. Jedná se o adresář, který je umístěn v cestě Pythonu a který má v systému Windows podobu `%APPDATA%\Python\Python31\site-packages` a v systému Unix (včetně Mac OS X) podobu `~/.local/lib/python3.1/site-packages`. Druhá a třetí možnost má tu výhodu, že máme náš kód oddělený od oficiální instalace.

Mít modul `TextUtil` je sice skvělé, ale pokud jej bude používat spousta programů, pak si chceme být jisti, že funguje tak, jak je uvedeno. Jeden skutečně jednoduchý způsob spočívá v provedení příkladů v dokumentačním řetězci a v kontrole, zda obdržíme očekávané výsledky. To můžeme provést přidáním pouhých třech řádků na konec souboru `.py` tohoto modulu:

```
if __name__ == "__main__":
   import doctest
   doctest.testmod()
```

Kdykoliv je nějaký modul importován, Python pro něj vytvoří proměnnou s názvem `__name__` a uloží název modulu do této proměnné. Název modulu je prostý název souboru `.py` bez přípony. V tomto případě bude mít tedy proměnná `__name__` po importu modulu hodnotu `"TextUtil"`, takže podmínka `if` nebude splněna, a proto se poslední dva řádky neprovedou. To znamená, že tyto tři řádky nemají prakticky žádný význam, je-li modul importován.

Kdykoliv je nějaký soubor `.py` spuštěn, Python pro něj vytvoří proměnnou s názvem `__name__` a nastaví ji na řetězec `"__main__"`. Pokud tedy spustíme soubor `TextUtil.py` jako by se jednalo o program, Python nastaví proměnnou `__name__` na hodnotu `"__main__"` a podmínka `if` se vyhodnotí na `True`, takže se poslední dva řádky provedou.

Funkce `doctest.testmod()` použije introspektivní prvky Pythonu pro vyhledání všech funkcí v daném modulu a jejich dokumentačních řetězců a pokusí se provést všechny v nich umístěné úryvky kódu. Když modul takto spustíme, obdržíme výstup pouze v případě výskytu nějakých chyb. To

může být na první pohled znepokojující, protože to vypadá, že se nic nestalo. Pokud ale na příkazovém řádku předáme příznak `-v`, obdržíme výstup podobný následujícímu:

```
Trying:
    is_balanced("(Python (není (jako (lisp))))")
Expecting:
    True
ok
...
Trying:
    simplify(" nesamohláskový ", delete="aáeiouyý")
Expecting:
    'nsmhlskv'
ok
4 items passed all tests:
   3 tests in __main__
   5 tests in __main__.is_balanced
   3 tests in __main__.shorten
   4 tests in __main__.simplify
15 tests in 4 items.
15 passed and 0 failed.
Test passed.
```

Místo množství vynechaných řádků je uveden výpustek. Jsou-li v modulu funkce (nebo třídy či metody), které nemají testy, pak jsou při použití volby `-v` taktéž uvedeny. Všimněte si, že modul `doctest` nalezl testy v dokumentačním řetězci modulu i v dokumentačních řetězcích funkcí.

Příklady v dokumentačních řetězcích, které lze spouštět jako testy, se nazývají *dokumentační testy* (doctests). Je třeba poznamenat, že při psaní dokumentačních testů můžeme zavolat funkci `simplify()` a ostatní funkce bez nutnosti kvalifikace (k dokumentačním testům totiž dochází uvnitř samotného modulu). Vně tohoto modulu musíme za předpokladu, že jsme použili příkaz `import TextUtil`, používat kvalifikované názvy, například `TextUtil.is_balanced()`.

V následující podčásti si ukážeme, jak provádět více důkladných testů – konkrétně testovací případy, v nichž očekáváme selhání, například neplatná data způsobující výjimky. (Testování se budeme podrobně věnovat v lekci 9.) Kromě toho se podíváme na několik dalších problémů, k nimž dochází při vytváření modulu, jako je inicializace modulu, ohled na rozdíly mezi platformami a zajištění, aby se při použití syntaxe `from modul import *` do programu či modulu skutečně importovaly pouze ty objekty, které chceme zveřejnit.

Modul CharGrid

Modul `CharGrid` uchovává v paměti znakovou mřížku. Nabízí funkce pro „kreslení“ čar, obdélníků a textu do mřížky a pro vykreslení této mřížky do konzoly. Zde jsou dokumentační testy z dokumentačních řetězců:

```
>>> resize(14, 50)
>>> add_rectangle(0, 0, *get_size())
>>> add_vertical_line(5, 10, 13)
>>> add_vertical_line(2, 9, 12, "!")
>>> add_horizontal_line(3, 10, 20, "+")
>>> add_rectangle(0, 0, 5, 5, "%")
>>> add_rectangle(5, 7, 12, 40, "#", True)
>>> add_rectangle(7, 9, 10, 38, " ")
>>> add_text(8, 10, "Tohle je náš modul CharGrid")
>>> add_text(1, 32, "Pěkně odporný", "@")
>>> add_rectangle(6, 42, 11, 46, fill=True)
>>> render(False)
```

Funkce `CharGrid.add_rectangle()` přijímá nejméně čtyři argumenty: řádek a sloupec levého horního rohu a řádek a sloupec pravého spodního rohu. Znak použitý pro nakreslení obrysu lze zadat jako pátý argument a jako šestý argument logickou hodnotu signalizující, zda má být obdélník vyplněn (stejným znakem jako obrys). Při prvním zavolání předáme třetí a čtvrté argumenty rozbalením n-tice se dvěma prvky (šířka, výška) vrácené funkcí `CharGrid.get_size()`.

Funkce `CharGrid.render()` ve výchozím nastavení vymaže obrazovku před vypsáním mřížky, čemuž lze zamezit předáním hodnoty `False`, jak jsme zde učinili také my. Níže je uvedena mřížka, která je výsledkem výše uvedených dokumentačních testů:

```
%%%%*********************************************
%   %                          @@@@@@@@@@@@@@@  *
%   %                          @Pěkně odporný@  *
%   %     ++++++++++           @@@@@@@@@@@@@@@  *
%%%%%                                           *
*       ###################################     *
*       ###################################  ****   *
*       ##                                 ##  ****   *
*       ## Tohle je náš modul CharGrid ##  ****   *
* !     ##                                 ##  ****   *
* !   | ###################################  ****   *
* !   | ###################################     *
*     |                                         *
*********************************************
```

Modul začíná stejným způsobem jako modul `TextUtil`: řádkem shebang, komentářem s copyrightem a informacemi o licenci a dokumentačním řetězcem, který popisuje modul a obsahuje výše uvedené dokumentační testy. Řádný kód pak začíná dvěma importy, z nichž prvním je modul `sys` a druhý modul `subprocess`. Modulu `subprocess` se budeme věnovat v lekci 10.

Modul se řídí dvěma zásadami pro ošetření chyb. Několik funkcí definuje parametr `char`, jehož aktuálním argumentem musí být vždy řetězec obsahující přesně jeden znak. Porušení tohoto požadavku je považováno za fatální chybu programování, takže ke kontrole délky používáme příkazy `assert`.

Na druhou stranu předání čísla řádku či sloupce mimo rozsah je považováno za chybové, ale normální, a proto dojde k vyvolání vlastní výjimky.

Nyní prostudujeme několik ilustrativních a klíčových částí kódu tohoto modulu, přičemž začneme vlastními výjimkami:

```
class RangeError(Exception): pass
class RowRangeError(RangeError): pass
class ColumnRangeError(RangeError): pass
```

Žádná z funkcí v modulu, která vyvolává výjimku, nikdy nevyvolá chybu `RangeError`, ale vždy vyvolá specifickou výjimku v závislosti na tom, zda hodnota zadaná mimo rozsah představuje řádek nebo sloupec. Avšak tím, že používáme hierarchii, dáváme uživatelům modulu možnost zachytit specifickou výjimku nebo zachytit kteroukoliv z nich zachycením bázové třídy `RangeError`. Všimněte si také, že uvnitř dokumentačních testů používáme názvy výjimek tak, jak jsou zde uvedeny, ale při importu modulu příkazem `import CharGrid` je samozřejmě nutné psát `CharGrid.RangeError`, `CharGrid.RowRangeError` a `CharGrid.ColumnRangeError`.

```
_CHAR_ASSERT_TEMPLATE = ("je nutné zadat jediný znak: '{0}' "
                         "je příliš dlouhý")
_max_rows = 25
_max_columns = 80
_grid = []
_background_char = " "
```

Zde definujeme několik soukromých dat pro interní použití v modulu. Identifikátory začínají podtržítkem, takže pokud se modul importuje příkazem `from CharGrid import *`, neimportuje se žádná z těchto proměnných. (Další možností by bylo zřízení seznamu `__all__`.) Řetězec `_CHAR_ASSERT_TEMPLATE` se používá v souvislosti s metodou `str.format()` pro vytvoření chybové zprávy v příkazech `assert`. K ostatním proměnným se vrátíme, jakmile se dostaneme k jejich použití v kódu.

```
if sys.platform.startswith("win"):
    def clear_screen():
        subprocess.call(["cmd.exe", "/C", "cls"])
else:
    def clear_screen():
        subprocess.call(["clear"])
clear_screen.__doc__ = """Vymaže obrazovku pomocí příkazu pro \
vymazání obrazovky aktuálně používaného systému"""
```

Prostředek k vymazání obrazovky je nezávislý na platformě. V systému Windows musíme spustit program `cmd.exe` s příslušnými argumenty a na většině unixových systémů spustíme program `clear`. Funkce `subprocess.call()` modulu `subprocess` umožňuje spuštění externího programu, takže ji můžeme použít k vymazání obrazovky způsobem vhodným pro používanou platformu. Řetězec `sys.platform` uchovává název operačního systému, na kterém daný program běží (např. „win32“ nebo „linux2“). Jeden ze způsobů pro překlenutí rozdílů mezi platformami tedy spočívá v definování jediné funkce:

```
def clear_screen():
    command = (["clear"] if not sys.platform.startswith("win") else
               ["cmd.exe", "/C", "cls"])
    subprocess.call(command)
```

Nevýhodou tohoto postupu je, že i když víme, že se platforma za běhu programu nezmění, provádíme kontrolu platformy při každém volání funkce.

Abychom zamezili kontrole, na které platformě program běží, při každém zavolání funkce `clear_screen()`, vytvořili jsme funkci `clear_screen()` s ohledem na používanou platformu jedenkrát při importu modulu a od tohoto okamžiku ji vždy používáme. To je možné díky tomu, že příkaz `def` je příkazem jazyka Python úplně stejně jako kterýkoliv jiný. Jakmile interpret dorazí k příkazu `if`, provede buď první, nebo druhý příkaz `def`, čímž se dynamicky vytvoří první nebo druhá funkce `clear_screen()`. Vzhledem k tomu, že tato funkce není definována uvnitř jiné funkce (nebo uvnitř třídy, jak uvidíme v následující lekci), jedná se o globální funkci, která je přístupná stejně jako kterákoli jiná funkce v modulu.

Po vytvoření funkce explicitně nastaví její dokumentační řetězec. Díky tomu nemusíme psát stejný dokumentační řetězec na dvou místech a také vidíme, že dokumentační řetězec je jen jedním z atributů funkce. Mezi další atributy patří modul funkce a její název.

```
def resize(max_rows, max_columns, char=None):
    """Změní velikost mřížky, přičemž zahodí obsah a
    změní pozadí, nemá-li znak představující pozadí hodnotu None
    """
    assert max_rows > 0 and max_columns > 0, "příliš malé"
    global _grid, _max_rows, _max_columns, _background_char
    if char is not None:
        assert len(char) == 1, _CHAR_ASSERT_TEMPLATE.format(char)
        _background_char = char
    _max_rows = max_rows
    _max_columns = max_columns
    _grid = [[_background_char for column in range(_max_columns)]
             for row in range(_max_rows)]
```

Tato funkce používá příkaz `assert` pro vynucení zásady, že pokus o změnu velikosti mřížky na velikost menší než 1 x 1 představuje chybu programování. Je-li zadán znak pro pozadí, použije se příkaz `assert` pro zajištění, že se jedná o řetězec s přesně jedním znakem. Není-li tomu tak, bude chybovou zprávou tvrzení text konstanty `_CHAR_ASSERT_TEMPLATE` se zástupným symbolem `{0}` nahrazeným zadaným řetězcem `char`.

Naneštěstí musíme použít příkaz `global`, protože potřebujeme aktualizovat několik globálních proměnných uvnitř této funkce. Jak uvidíme v lekci 6, v tomto ohledu nám výrazně pomůže objektově orientovaný přístup.

Proměnnou `_grid` vytváříme pomocí seznamové komprehenze uvnitř seznamové komprehenze. Použití replikace seznamu (`[[char] * columns] *`) by zde nefungovalo, protože vnitřní seznam bude sdílený (mělce zkopírovaný). Mohli bychom ale použít vnořené cykly `for ... in`:

Seznamové komprehenze ➤ 120

```
_grid = []
for row in range(_max_rows):
    _grid.append([])
    for column in range(_max_columns):
        _grid[-1].append(_background_char)
```

Tento kód je pravděpodobně méně srozumitelný než seznamová komprehenze a je také mnohem delší.

Nyní se podíváme jen na jednu z kreslicích funkcí, abychom získali ponětí o tom, jak se takové kreslení provádí, protože naším hlavním zájmem je implementace modulu. Zde je funkce `add_horizontal_line()` rozdělená na dvě části:

```
def add_horizontal_line(row, column0, column1, char="-"):
    """Přidá do mřížky vodorovnou čáru s použití zadaného znaku

    >>> add_horizontal_line(8, 20, 25, "=")
    >>> char_at(8, 20) == char_at(8, 24) == "="
    True
    >>> add_horizontal_line(31, 11, 12)
    Traceback (most recent call last):
    ...
    RowRangeError
    """
```

Dokumentační řetězec obsahuje dva testy, z nichž první by měl fungovat a druhý by měl vyvolat výjimku. Když v dokumentačních testech pracujeme s výjimkami, stačí uvést řádek „Traceback“, protože ten je vždy stejný a říká modulu `doctest`, že očekáváme výjimku, pak následuje výpustek znamenající další řádky (které se mění) a nakonec řádek s výjimkou, kterou čekáme, že obdržíme. Funkce `char_at()` patří mezi funkce tohoto modulu. Vrací znak na zadaném řádku a sloupci v mřížce.

```
    assert len(char) == 1, _CHAR_ASSERT_TEMPLATE.format(char)
    try:
        for column in range(column0, column1):
            _grid[row][column] = char
    except IndexError:
        if not 0 <= row <= _max_rows:
            raise RowRangeError()
        raise ColumnRangeError()
```

Kód začíná stejnou kontrolou délky argumentu `char` jako ve funkci `resize()`. Místo explicitní kontroly argumentů `row` a `column` funguje funkce na základě předpokladu, že argumenty jsou platné. Pokud dojde k výjimce `IndexError` kvůli přístupu k neexistujícímu řádku nebo sloupci, výjimku zachytíme a vyvoláme místo ní příslušnou výjimku specifickou pro tento modul. Tento způsob programování se neformálně označuje jako „je snazší žádat o prominutí než o dovolení“ a ve srovnání se způsobem „dívej se, než skočíš“, kde se kontroly provádějí předem, je pro jazyk Python obecně považován za vhodnější. Spoléhání na vyvolání výjimky spíše než na předem provedenou kontrolu

je efektivnější v případech, kdy je výskyt výjimek vzácný. (Tvrzení se do oblasti „dívej se, než skočíš“ nepočítají, protože v kódu nasazeném do ostrého provozu by k nim nikdy nemělo dojít – proto jsou také často deaktivovány komentářem.)

Téměř na konci modulu je za definicemi všech funkcí jediné volání funkce `resize()`:

```
resize(_max_rows, _max_columns)
```

Toto volání inicializuje mřížku na výchozí velikost (25 x 80) a zajistí, aby kód, který importuje modul, mohl tento modul bezpečně a okamžitě používat. Bez tohoto volání by importující program či modul musel při každém importu tohoto modulu zavolat funkci `resize()` pro inicializaci mřížky, což by si programátoři museli pamatovat a mohlo by to navíc vést k vícenásobným inicializacím.

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Poslední tři řádky modulu jsou standardní pro moduly, které používají modul `doctest` pro kontrolu svých dokumentačních testů. (Testování se budeme podrobně věnovat v lekci 9.)

Modul `CharGrid` má jeden podstatný nedostatek: podporuje pouze jediný mřížkový znak. To se dá vyřešit například uchováváním kolekce mřížek v modulu, což by ale znamenalo, že uživatelé modulu budou muset při volání každé funkce uvádět klíč nebo index označující, na kterou mřížku odkazují. V případech, kdy je vyžadováno více instancí nějakého objektu, je lepším řešením vytvořit modul, který definuje třídu (vlastní datový typ), protože můžeme vytvářet tolik instancí třídy (objektů daného datového typu), kolik jen chceme. Další výhoda vytvoření třídy spočívá v tom, že bychom nemuseli pro ukládání (statických) dat třídy používat příkaz `global`. Způsob vytváření tříd si ukážeme v další lekci.

Přehled standardní knihovny Pythonu

Standardní knihovna Pythonu je obecně popisována jako „včetně baterií“. K dispozici je skutečně pestrá paleta funkčních prvků rozprostřených do přibližně dvou stovek balíčků a modulů.

Ve skutečnosti bylo pro jazyk Python v průběhu let vyvinuto tolik vysoce kvalitních modulů, že pokud bychom je všechny začlenili do standardní knihovny, vzrostla by velikost distribučních balíčků Pythonu přinejmenším o jeden řád. Moduly umístěné do této knihovny jsou tedy spíše odrazem historického vývoje Pythonu a zájmů jeho hlavních vývojářů než jakkoli koordinovanou či systematickou snahou o vytvoření „vyvážené“ knihovny. Kromě toho se některé moduly ukázaly jako velmi obtížně udržovatelné v rámci knihovny – zvláště modul Berkeley DB – a proto byly z knihovny odebrány a nyní jsou udržovány nezávisle na ní. To znamená, že pro Python je k dispozici řada skvělých modulů třetích stran, které (bez ohledu na svoji kvalitu a užitečnost) nejsou součástí standardní knihovny. (Na dva takovéto moduly se později podíváme: moduly PyParsing a PLY použijeme v lekci 14 pro vytvoření analyzátorů.)

V této části se podíváme na obsáhlejší přehled toho, co vše máme dispozici. Výklad rozdělíme dle tematických okruhů a vynecháme ty balíčky a moduly, které jsou silně specializované, a také ty, které jsou určeny pro konkrétní platformu. V mnoha případech si ukážeme malý příklad, abychom si mohli

některé balíčky a moduly přímo „osahat“. U balíčků a modulů probíraných v jiných částech knihy jsou k dispozici křížové odkazy.

Práce s řetězci

Modul `string` nabízí několik užitečných konstant, mezi něž patří `string.ascii_letters` a `string.hexdigits`. Dále nabízí třídu `string.Formatter`, z níž můžeme odvodit třídu poskytující přizpůsobené formátování řetězců.* Modul `textwrap` lze použít k zalomení řádků textu na zadanou šířku a k minimalizaci odsazení.

typ bytes ➤ 286

Modul `struct` poskytuje funkce pro zabalení, resp. rozbalení, čísel, logických hodnot a řetězců do (resp. z) objektů typu `bytes` s použitím jejich binární reprezentace. To může být užitečné při práci s daty, která se mají odeslat nebo přijmout z nízkoúrovňových knihoven napsaných v jazyku C. Moduly `struct` a `textwrap` používá program `convert-incidents.py`, na který se podíváme v lekci 7.

modul struct ➤ 288

Modul `difflib` nabízí třídy a metody pro porovnávání posloupností, jako jsou kupříkladu řetězce. Tento modul je schopen vytvořit výstup ve standardních formátech „diff“ i v jazyku HTML.

Nejvýkonnějším modulem Pythonu pro práci s řetězci je modul `re` (regulární výrazy), který si podrobně prostudujeme v lekci 13.

Třída `io.StringIO` nabízí objekty podobné řetězcům, které se chovají jako textový soubor umístěný v paměti. To může být užitečné, chceme-li pro zápis do řetězce použít stejný kód, který zapisuje do souboru.

Příklad: Třída io.StringIO

Python nabízí dva odlišné způsoby zápisu textu do souborů. První spočívá v použití metody `write()` objektu souboru a druhý v použití funkce `print()` s klíčovým argumentem `file` nastaveným na objekt souboru, který je otevřený pro zápis:

```
print("Chybová zpráva", file=sys.stdout)
sys.stdout.write("Chybová zpráva\n")
```

Oba řádky textu se vypíšou do objektu `sys.stdout`, což je objekt souboru, který představuje „standardní výstupní proud“. Tento proud je obvykle směrován do konzoly a od objektu `sys.stderr` („chybový výstupní proud“) se liší pouze v tom, že je ukládán do mezipaměti. (Python automaticky vytváří a otevírá proudy `sys.stdin`, `sys.stdout` a `sys.stderr` při spuštění programu.) Funkce `print()` přidává standardně nový řádek, což můžeme potlačit zadáním klíčového argumentu `end`, který nastavíme na prázdný řetězec.

V některých situacích je užitečné mít možnost zachytit do řetězce výstup, který má jít do nějakého souboru. To lze realizovat pomocí třídy `io.StringIO`, jež poskytuje objekt, který je možné používat stejně jako objekt souboru, přičemž se všechna zapsaná data uchovají v řetězci. Objektu typu `io.StringIO` můžeme předat počáteční řetězec, takže z něj lze také číst jako ze souboru.

* Výraz *odvození třídy* (nebo též *specializace*) se používá tehdy, když vytváříme vlastní datový typ (tj. třídu) založený na jiné třídě. Tomuto tématu se budeme plně věnovat v lekci 6.

Ke třídě `io.StringIO` můžeme přistupovat po importu modulu `io` a můžeme ji použít k zachycení výstupu určeného pro objekt souboru, jako je například `sys.stdout`:

```
sys.stdout = io.StringIO()
```

Pokud tento řádek umístíme na začátek programu za příkazy import, ale před jakékoliv použití proudu `sys.stdout`, odešle se veškerý text posílaný do proudu `sys.stdout` do objektu `io.StringIO`, který jsme vytvořili na tomto řádku a který nahradil standardní objekt souboru `sys.stdout`. Když se nyní provedou výše uvedené řádky `print()` a `sys.stdout.write()`, nepůjde jejich výstup do konzoly, ale do objektu `io.StringIO`. (Původní proud `sys.stdout` můžeme kdykoliv obnovit příkazem `sys.stdout = sys.__stdout__`.)

Všechny řetězce, které byly zapsány do objektu `io.StringIO`, můžeme získat pomocí funkce `io.StringIO.getvalue()`, což v našem případě znamená voláním funkce `sys.stdout.getvalue()`. Její návratovou hodnotou je řetězec obsahující všechny řádky, které byly dosud zapsány. Tento řetězec můžeme vypsat nebo uložit do souboru protokolu nebo poslat přes síťové připojení jako jakýkoliv jiný řetězec. S dalším příkladem použití třídy `io.StringIO` se setkáme v pozdější části této lekce (strana 233).

Programování na příkazovém řádku

Pokud potřebujeme, aby náš program byl schopen zpracovat text, který mohl být přesměrován z konzoly nebo který mohl být v souborech uvedených na příkazovém řádku, pak můžeme použít funkci `fileinput.input()` modulu `fileinput`. Tato funkce prochází všechny řádky přesměrované z konzoly (jsou-li nějaké) a všechny řádky v souborech uvedených na příkazovém řádku jako spojitou posloupnost řádků. Modul umí prostřednictvím funkcí `fileinput.filename()` a `fileinput.lineno()` ohlásit aktuální název souboru a číslo řádku a dále dokáže pracovat s některými typy komprimovaných souborů.

Pro práci s volbami příkazového řádku jsou k dispozici dva samostatné moduly. Modul `getopt` je populární, protože jej lze jednoduše používat a je již delší dobu součástí knihovny. Modul `optparse` je novější a výkonnější.

Příklad: Modul optparse

příklad `csv2html.py` ➤ 100

V lekci 2 jsme si popsali program `csv2html.py`. Ve cvičeních druhé lekce jsme navrhli rozšíření tohoto programu spočívající v přijímání argumentů z příkazového řádku („maxwidth“ jako celé číslo a „format“ jako řetězec). Modelové řešení (`csv2html2_ans.py`) má pro zpracování argumentů funkci o velikosti 26 řádků. Zde je začátek funkce `main()` programu `csv2html2_opt.py`, což je další verze programu, která pro zpracování argumentů příkazového řádku nepoužívá vlastní funkci, ale modul `optparse`:

```
def main():
    parser = optparse.OptionParser()
    parser.add_option("-w", "--maxwidth", dest="maxwidth", type="int",
            help=("maximální počet znaků, které lze "
                  "vypsat do řetězcových polí [výchozí: %default]"))
    parser.add_option("-f", "--format", dest="format",
```

```
        help=("formát pro výpis čísel "
              "[výchozí: %default]"))
parser.set_defaults(maxwidth=100, format=".0f")
opts, args = parser.parse_args()
```

Stačí pouze devět řádků kódu plus příkaz import `optparse`. Kromě toho nepotřebujeme explicitně řešit volby `-h` a `--help`, o něž se postará modul `optparse`, který vypíše vhodnou zprávu o použití programu pomocí textů z klíčových argumentů help, v nichž nahradí text „%default" výchozí hodnotou příslušné volby.

Dále si všimněte, že volby nyní používají standardní unixový styl krátkých a dlouhých názvů začínajících pomlčkou. Krátké názvy jsou vhodné pro interaktivní použití v konzole, zatímco dlouhé názvy jsou srozumitelnější při použití ve skriptech shellu. Například pro nastavení maximální šířky na 80 můžeme použít `-w80`, `-w 80`, `--maxwidth=80` nebo `--maxwidth 80`. Po analýze příkazového řádku jsou volby k dispozici prostřednictvím atributů s příslušnými názvy, například `opts.maxwidth` a `opts.format`. Všechny argumenty, které nebyly zpracovány (obvykle názvy souborů), jsou v seznamu `args`.

Pokud během analýzy příkazového řádku dojde k nějaké chybě, zavolá modul `optparse` funkci `sys.exit(2)`. To vede k čistému ukončení programu, při němž operační systém obdrží jako návratovou hodnotu programu číslo 2. Návratová hodnota 2 standardně označuje chybu při použití, 1 označuje jakýkoliv jiný druh chyby a 0 znamená úspěch. Pokud funkci `sys.exit()` zavoláme bez argumentů, vrátí operačnímu systému hodnotu 0.

Matematika a čísla

Kromě vestavěných čísel typu `int`, `float` a `complex` nabízí knihovna čísla typu `decimal.Decimal` a `fractions.Fraction`. K dispozici máme tři numerické knihovny: `math` pro standardní matematické funkce, `cmath` pro matematické funkce pracující s komplexními čísly a `random` poskytující řadu funkcí pro generování náhodných čísel. Tyto moduly jsme si představili v lekci 2.

Numerické abstraktní bázové třídy jazyka Python (tj. třídy, od nichž lze odvozovat další třídy, ale které nelze použít přímo) se nacházejí v modulu `numbers`. Tyto třídy jsou užitečné pro kontrolu, zda daný objekt `x` představuje libovolný druh čísla (např. `isinstance(x, numbers.Number)`) nebo zda je specifickým druhem čísla (např. `isinstance(x, numbers.Rational)` nebo `isinstance(x, numbers.Integral)`).

V oblasti vědeckého a inženýrského programování přijde vhod balíček třetí strany s názvem `NumPy`. Tento modul nabízí vysoce efektivní n-dimenzionální pole, základní funkce z oblasti lineární algebry a Fourierových transformací a nástroje pro integraci kódu napsaného v jazyku C, C++ a Fortran. Balíček `SciPy` obsahuje modul `NumPy`, který rozšiřuje o moduly pro statistické výpočty, zpracování signálu a obrázků, genetické algoritmy a pro řadu dalších oblastí. Oba balíčky jsou k dispozici zdarma na adrese *www.scipy.org*.

Datum a čas

Moduly `calendar` a `datetime` nabízejí funkce a třídy pro práci s datem a časem. Jsou ovšem založeny na idealizovaném gregoriánském kalendáři, takže nejsou vhodné pro práci s předgregoriánskými kalendářními daty. Práce s datem a časem je velice složité téma. Používané kalendáře se v jednotli-

vých místech a časech liší, den netrvá přesně 24 hodin, rok nemá přesně 365 dnů, odlišný je též letní čas i časová pásma. Třída `datetime.datetime` (ne však třída `datetime.date`) si sice dokáže poradit s časovými pásmy, ale ne sama od sebe. Tento nedostatek napravují moduly třetích stran, například `dateutil` z *www.labix.org/python-dateutil* a `mxDateTime` z *www.egenix.com/products/python/mx-Base/mxDateTime*.

Modul `time` pracuje s časovými známkami. Jedná se o prostá čísla, která uchovávají počet vteřin od začátku jisté epochy (1. 1. 1970 00:00:00 na Unixu). Tento modul lze použít pro získání časové známky aktuálního času na daném stroji v UTC (Coordinated Universal Time – koordinovaný světový čas) nebo jako místní čas, který bere v potaz letní čas. Dále jej lze použít pro vytváření nejrůznějším způsobem naformátovaných řetězců s datem, časem a datem i časem. Kromě toho dokáže analyzovat řetězce obsahující kalendářní data a časy.

Příklad: Moduly calendar, datetime a time

Objekty typu `datetime.datetime` se obvykle vytvářejí programově, kdežto objekty, jež uchovávají data a časy v UTC, se obvykle získávají z externích zdrojů, jako jsou časové známky souborů. Zde je několik příkladů:

```
import calendar, datetime, time
moon_datetime_a = datetime.datetime(1969, 7, 20, 20, 17, 40)
moon_time = calendar.timegm(moon_datetime_a.utctimetuple())
moon_datetime_b = datetime.datetime.utcfromtimestamp(moon_time)
moon_datetime_a.isoformat()     # vrátí: '1969-07-20T20:17:40'
moon_datetime_b.isoformat()     # vrátí: '1969-07-20T20:17:40'
time.strftime("%Y-%m-%dT%H:%M:%S", time.gmtime(moon_time))
```

Proměnná `moon_datetime_a` je typu `datetime.datetime` a uchovává datum a čas přistání vesmírné lodi Apollo 11 na Měsíci. Proměnná `moon_time` je typu in a uchovává počet vteřin od začátku epochy po přistání na Měsíci – toto číslo poskytuje funkce `calendar.timegm()`, která přijímá objekt typu `time_struct` získaný od funkce `datetime.datetime.utctimetuple()` a vrací počet vteřin reprezentovaných objektem typu `time_struct`. (K přistání na Měsíci došlo před unixovou epochou, a proto je výsledné číslo záporné.) Proměnná `moon_datetime_b` je typu `datetime.datetime` a vytváříme ji z celého čísla `moon_time`, abychom si ukázali převod z počtu vteřin od začátku epochy na objekt typu `datetime.datetime`.* Poslední tři řádky vracejí identické řetězce s datem a časem ve formátu ISO 8601.

Aktuální datum a čas v UTC je k dispozici jako objekt typu `datetime.datetime` prostřednictvím funkce `datetime.datetime.utcnow()` a jako počet vteřin od počátku epochy prostřednictvím funkce `time.time()`. Pro místní datum a čas použijte `datetime.datetime.now()` nebo `time.mktime(time.localtime())`.

* Pro uživatele systému Windows je nutné podotknout, že funkce `datetime.datetime.utcfromtimestamp()` neumí zpracovat záporné časové známky, což znamená časové známky pro data před 1. lednem 1970.

Algoritmy a datové kolekce představující kolekce

Modul `bisect` nabízí funkce pro prohledávání seřazených posloupností, jako jsou seřazené seznamy, a pro vkládání prvků nenarušující jejich seřazené pořadí. Funkce tohoto modulu používají algoritmus binárního hledání, takže jsou velice rychlé. Modul `heapq` nabízí funkce pro převod posloupnosti (např. seznamu) na haldu, což je datový typ představující kolekci, v němž je první prvek (na indexové pozici 0) vždy nejmenším prvkem. Dále nabízí funkce pro vkládání a odstraňování prvků při zachování posloupnosti ve stavu haldy.

Výchozí slovníky ➤ 135

Balíček `collections` nabízí slovník `collections.defaultdict` a datový typ představující kolekci s názvem `collections.namedtuple`, který jsme probírali již dříve. Kromě toho nabízí typy `collections.UserList` a `collections.UserDict`, ačkoliv častěji se třídy odvozují od vestavěných typů `list` a `dict`. Dalším typem je `collections.deque`, který je podobný seznamu, avšak zatímco seznam je velmi rychlý při přidávání a odstraňování prvků na konci, typ `collections.deque` je velmi rychlý při přidávání a odstraňování prvků na začátku i na konci.

Pojmenovaná n-tice ➤ 113

Uspořádané slovníky ➤ 136

Python 3.1 zavádí třídy `collections.OrderedDict` a `collections.Counter`. Třída `OrderedDict` má stejné rozhraní API jako běžný typ `dict`, avšak při iteraci jsou prvky vždy vraceny ve vkládaném pořadí (tj. od prvního po poslední vložený prvek) a metoda `popitem()` vždy vrátí naposledy přidaný (tj. poslední) prvek. Třída `Counter` je podtřídou typu `dict` a poskytuje rychlý a snadný způsob udržování nejrůznějších počítadel. Při zadání iterovatelného prvku nebo mapování (např. slovníku) dokáže instance třídy `Counter` například vrátit seznam jedinečných prvků nebo seznam nejčastějších prvků jako dvojice (prvek, počet).

V balíčku `collections` se nacházejí také nenumerické abstraktní bázové třídy jazyka Python (tj. třídy, od nichž lze odvozovat další třídy, ale které nelze použít přímo). Budeme se jim věnovat v lekci 8.

Modul `array` nabízí typ představující posloupnost s názvem `array.array`, který dokáže uchovávat čísla nebo znaky velice efektivním způsobem s ohledem na potřebný prostor v paměti. Chová se podobně jako seznam, tedy až na to, že možný typ uchovávaných objektů je zafixován při jeho vytvoření, takže na rozdíl od seznamů nedokáže uchovávat objekty odlišných typů. Výše zmíněný balíček třetí strany `NumPy` nabízí také efektivní pole.

Modul `weakref` nabízí funkční prvky pro vytváření slabých odkazů. Ty se chovají stejně jako běžné odkazy na objekty, ovšem s tím, že pokud jediným odkazem na daný objekt je slabý odkaz, může být tento objekt i tak naplánován na úklid z paměti. Díky tomu není nutné udržovat v paměti objekty jen proto, že na ně máme nějaký odkaz. Přirozeně lze zkontrolovat, zda objekt, na který slabý odkaz ukazuje, stále existuje, a v případě že ano, tak k tomuto objektu přistupovat.

Příklad: Modul heapq

Modul `heapq` nabízí funkce pro převod seznamu na haldu a pro přidávání a odstraňování prvků z haldy při zachování *vlastnosti haldy*. Halda je binární strom, který respektuje vlastnost haldy, která spočívá v tom, že první prvek (na indexové pozici 0) je vždy nejmenší.* Každý podstrom haldy je též haldou, takže také respektuje vlastnost haldy. Zde je příklad vytvoření nové haldy:

* Přesněji řečeno, modul `heapq` nabízí *minimální haldu*. Haldy, v nichž je první prvek vždy největší, se označují jako *maximální haldy*.

```
import heapq
heap = []
heapq.heappush(heap, (5, "rest"))
heapq.heappush(heap, (2, "work"))
heapq.heappush(heap, (4, "study"))
```

Máme-li seznam, můžeme jej převést na haldu pomocí funkce `heapq.heapify(`*`seznam`*`)`, která provede veškerá nezbytná přeuspořádání přímo v zadaném seznamu. Nejmenší prvek lze poté odstranit z haldy pomocí funkce `heapq.heappop(`*`halda`*`)`.

```
for x in heapq.merge([1, 3, 5, 8], [2, 4, 7], [0, 1, 6, 8, 9]):
    print(x, end=" ")     # vypíše: 0 1 1 2 3 4 5 6 7 8 8 9
```

Funkce `heapq.merge()` přijímá jako argumenty libovolný počet seřazených iterovatelných objektů a vrací iterátor, který prochází všechny prvky ze všech iterovatelných objektů seřazené ve správném pořadí.

Souborové formáty, kódování a perzistence dat

Kódování znaků ➤ 95

Standardní knihovna obsahuje rozsáhlou podporu pro nejrůznější standardní souborové formáty a kódování. Modul base64 má funkce pro čtení a zapisování s použitím kódování Base16, Base32 a Base64 specifikovaných v dokumentu RFC 3548.* Modul `quopri` obsahuje funkce pro čtení a zapisování formátu označovaného jako „quoted-printable". Tento formát je definován v dokumentu RCF 1521 a používá se pro data spadající do rozšíření MIME (Multipurpose Internet Mail Extensions – víceúčelová rozšíření internetové pošty). Modul `uu` má funkce pro čtení a zapisování dat v kódování označovaném jako uuencoding („Unix-to-Unix encoding"). Dokument RFC 1832 definuje standard externí reprezentace dat (External Data Representation Standard) a modul `xdrlib` poskytuje funkce pro čtení a zapisování data v tomto formátu.

K dispozici jsou též moduly pro čtení a zapisování archivních souborů ve většině oblíbených formátů. Modul `bz2` umí pracovat se soubory `.bz2`, modul `gzip` se soubory `.gz`, modul `tarfile` se soubory `.tar`, `.targ.gz` (také `.tgz`) a `.tar.bz2` a modul `zipfile` si poradí se soubory `.zip`. V této podčásti si ukážeme příklad použití modulu `tarfile` a později (strana 223) se podíváme na malý příklad, který používá modul `gzip`. Tento modul uvidíme opět v akci v lekci 7.

K dispozici je též podpora pro práci s některými formáty audia. Modul `aifc` podporuje formát AIFF (Audio Interchange File Format – výměnný souborový formát zvuku) a modul `wave` umí pracovat s nekomprimovanými soubory `.wav`. S určitými druhy audiodat lze manipulovat pomocí modulu `audioop` a modul `sndhdr` nabízí několik funkcí pro zjištění, jaký druh zvukových dat je v daném souboru uložen a jaké jsou některé z jeho vlastností, jako je například vzorkovací frekvence.

Formát pro konfigurační soubory (podobný souborům `.ini` ze starých Windows) specifikuje dokument RFC 822, přičemž funkce pro čtení a zapisování těchto souborů nabízí modul `configparser`.

---

* Dokumenty RFC (Request for Comments – žádost o komentáře) se používají pro specifikaci nejrůznějších internetových technologií. Každý má jedinečné identifikační číslo a z mnoha z nich se staly oficiálně přijímané standardy.

Řada aplikací (např. Excel) dokáže číst a zapisovat data ve formátu CSV (Comma Separated Value – hodnoty oddělené čárkou) nebo jeho varianty, jako jsou kupříkladu data oddělená tabulátorem. Tyto formáty dokáže číst a zapisovat modul `csv`, který umí zohlednit idiosynkrazie, jež znemožňují jejich přímé zpracování.

Standardní knihovna obsahuje kromě podpory rozličných souborových formátů také balíčky a moduly, jež poskytují perzistenci dat. Modul `pickle` se používá pro, ukládání resp., načítání jakýchkoli objektů Pythonu (včetně celých kolekcí) do (resp. ze) souborů na disku. Tento modul budeme podrobně probírat v lekci 7. Knihovna dále podporuje nejrůznější typy souborů DBM. Jedná se o soubory podobné slovníkům, jejichž prvky nejsou uloženy v paměti, ale na disku a jejichž klíče a hodnoty musejí být objekty typu `bytes` nebo řetězce. Modul `shelve`, kterému se budeme věnovat v lekci 12, lze použít pro soubory DBM s řetězcovými klíči a libovolnými objekty Pythonu jako hodnotami. Tento modul v pozadí převádí objekty Pythonu na objekty typu `bytes` a zpět. Moduly pro soubory DBM, databázové rozhraní API Pythonu a použití vestavěné databáze SQLite budeme probírat v lekci 12.

Příklad: Modul base64

Modul base64 se nejčastěji používá pro práci s binárními daty vkládanými do e-mailů jako text v kódování ASCII. Můžeme jej použít také pro ukládání binárních dat do souborů `.py`. Prvním krokem je převod binárních dat do formátu Base64. V tomto příkladu předpokládáme, že modul `base64` již byl importován, a že cesta je společně s názvem souboru `.png` uložena v proměnné `left_align_png`:

```
binary = open(left_align_png, "rb").read()
ascii_text = ""
for i, c in enumerate(base64.b64encode(binary)):
    if i and i % 68 == 0:
        ascii_text += "\\\n"
    ascii_text += chr(c)
```

left_align.png

typ bytes ➤ 286

V tomto úryvku kódu čteme soubor v binárním režimu a převádíme jej do řetězce Base64 tvořeného znaky z kódování ASCII. Za každý šedesátý osmý znak přidáváme kombinaci zpětného lomítka a znaku nového řádku. Tím omezíme šířku řádků na 68 znaků ASCII. Při opětovném čtení dat se tyto nové řádky budou ignorovat (protože zpětné lomítko je potlačí). Takto získaný text ASCII můžeme uložit jako literál typu `bytes` do souboru `.py`:

```
LEFT_ALIGN_PNG = b"""\
iVBORw0KGgoAAAANSUhEUgAAACAAAAAgCAYAAABzenr0AAAABGdBTUEAALGPC/xhBQAA\
...
bmquu8PAmVT2+CwVV6rCyA9UfFMCkI+bN6p18tCWqcUzrDOwBh2zVCR+JZVeAAAAAElF\
TkSuQmCC"""
```

Většinu řádků jsme zde vynechali a nahradili výpustkem. Tato data lze převést zpět do původní binární podoby takto:

```
binary = base64.b64decode(LEFT_ALIGN_PNG)
```

Tato binární data bychom mohli zapsat do souboru pomocí příkazu `open(`*`název_souboru`*`, "wb").write(binary)`. Ukládání binárních dat do souborů `.py` není tak kompaktní jako v jejich původní

podobě, může to však být užitečné v situaci, kdy potřebujeme napsat program jako jediný soubor `.py` vyžadující určitá binární data.

Příklad: Modul tarfile

Většina verzí systému Windows se nedodává s podporou formátu `.tar`, který je na unixových systémech velmi rozšířen. Tento nedostatek lze snadno vyřešit pomocí modulu Pythonu s názvem `tarfile`, který dokáže vytvářet a rozbalovat archivy `.tar` a `.targ.gz` (označované jako *archivy tarball*) a s nainstalovanými správnými knihovnami i archivy `.tar.bz2`. Program `untar.py` umí rozbalovat archivy tarball pomocí modulu `tarfile`. Zde si ukážeme pouze některé jeho klíčové části. Začneme prvním příkazem `import`:

```
BZ2_AVAILABLE = True
try:
    import bz2
except ImportError:
    BZ2_AVAILABLE = False
```

Modul `bz2` se používá pro práci s kompresním formátem bzip2. Jeho importování selže, pokud byl Python sestaven bez přístupu ke knihovně bzip2. (Binární distribuce Pythonu pro Windows se vždy sestavuje s vestavěnou kompresí bzip2. Pouze některá unixová sestavení mohou tuto knihovnu postrádat.) Počítáme s možností, že tento modul nemusí být k dispozici, a proto používáme blok `try ... except` a logickou proměnnou, na niž se můžeme později odkázat (ačkoliv kód, v němž se na ni odkazujeme, si zde neukážeme).

```
UNTRUSTED_PREFIXES = tuple(["/", "\\"] +
        [c + ":" for c in string.ascii_letters])
```

Tento příkaz vytváří n-tici `('/', '\', 'A:', 'B:', ..., 'Z:', 'a:', 'b:', ..., 'z:')`. Jakýkoliv soubor rozbalovaný z archivu tarball, jehož název začíná jedním z těchto prefixů, je podezřelý. Archivy tarball by totiž neměly obsahovat absolutní cesty, protože tím bychom riskovali přepsání systémových souborů. Jako opatření proto nerozbalíme žádný z takto identifikovaných souborů.

```
def untar(archive):
    tar = None
    try:
        tar = tarfile.open(archive)
        for member in tar.getmembers():
            if member.name.startswith(UNTRUSTED_PREFIXES):
                print("nedůvěryhodný prefix, ignoruji", member.name)
            elif ".." in member.name:
                print("podezřelá cesta, ignoruji", member.name)
            else:
                tar.extract(member)
                print("unpacked", member.name)
    except (tarfile.TarError, EnvironmentError) as err:
        error(err)
```

```
    finally:
        if tar is not None:
            tar.close()
```

Každý soubor v archivu tarball se nazývá *člen*. Funkce `tar.getmembers()` vrací seznam objektů typu `tarfile.TarInfo`, jeden pro každý člen. Název souboru člena včetně jeho cesty je uložen v atributu `tarfile.TarInfo.name`. Pokud tento název začíná nedůvěryhodným prefixem nebo pokud ve své cestě obsahuje „..“, vypíšeme chybovou zprávu. V opačném případě zavoláme metodu `tar.extract()`, která uloží daný člen na disk. Modul `tarfile` má vlastní skupinu výjimek. My jsme však postupovali co nejjednodušeji, takže při výskytu jakékoli výjimky vypíšeme chybovou zprávu a ukončíme program.

```
def error(message, exit_status=1):
    print(message)
    sys.exit(exit_status)
```

Pro úplnost zde uvádíme také funkci `error()`. Funkce `main()` (kterou jsme si zde neukázali) vypíše při zadání volby `-h` nebo `--help` zprávu o použití programu. V opačném případě provede před zavoláním funkce `untar()` s názvem souboru archivu tarball několik základních kontrol.

Práce se soubory, adresáři a procesy

Modul `shutil` nabízí vysokoúrovňové funkce pro práci se soubory a adresáři, mezi něž patří funkce `shutil.copy()` a `shutil.copytree`() pro kopírování souborů a celých adresářových stromů, `shutil.move()` pro přesouvání adresářových stromů a `shutil.rmtree()` pro odstraňování celých (i neprázdných) adresářových stromů.

Dočasné soubory a adresáře by se měly vytvářet pomocí modulu `tempfile`, který poskytuje nezbytné funkce (např. `tempfile.mkstemp()`) a dočasné prostředky vytváří tím nejbezpečnějším možným způsobem.

Funkce `filecmp.cmp()` z modulu `filecmp` lze použít k porovnání souborů a funkci `filecmp.cmpfiles()` můžeme použít k porovnání celých adresářů.

Jednou z oblastí, kde jsou programy Pythonu skutečně silné a efektivní, je organizování běhu jiných programů. K tomuto účelu slouží modul `subprocess`, který umí spouštět jiné procesy, komunikovat s nimi pomocí kanálů (pipes) a získávat jejich výsledky. Tomuto modulu se budeme věnovat v lekci 10. Ještě výkonnější alternativa spočívá v použití modulu `multiprocessing`, který nabízí rozsáhlé prostředky pro rozložení práce na více procesů a pro shromáždění výsledků a který lze často použít jako alternativu k vícevláknovému zpracování.

Modul `os` nabízí na platformě nezávislý přístup k funkčním prvkům operačního systému. Proměnná `os.environ` uchovává mapovací objekt, jehož prvky tvoří názvy proměnných prostředí s jejich hodnotami. Pracovní adresář programu poskytuje funkce `os.getcwd()` a pro jeho změnu slouží funkce `os.chdir()`. Modul dále nabízí funkce pro nízkoúrovňovou práci s popisovači souborů. Funkci `os.access()` lze použít pro zjištění, zda daný soubor existuje nebo zda je možné z něj číst nebo do něj zapisovat, a funkce `os.listdir()` vrací seznam záznamů (např. souborů a adresářů, avšak bez

záznamů „.“ a „..“) v zadaném adresáři. Funkce `os.stat()` vrací nejrůznější údaje o zadaném souboru či adresáři, jako je jeho režim, čas přístupu a velikost.

Adresáře lze vytvářet pomocí funkce `os.mkdir()` nebo v případě tvorby přechodových adresářů také pomocí funkce `os.makedirs()`. Prázdné adresáře lze odstranit funkcí `os.rmdir()` a adresářový strom obsahující pouze prázdné adresáře odstraníme funkcí `os.removedirs()`. Soubory či adresáře můžeme odstranit funkcí `os.remove()` a přejmenovat funkcí `os.rename()`.

Funkce `os.walk()` prochází celý adresářový strom a poskytuje postupně názvy všech souborů a adresářů.

Modul os nabízí také řadu nízkoúrovňových funkcí specifických pro určitou platformu, například pro práci s popisovači a pro unixová systémová volání typu `fork`, `spawn` a `exec`.

Zatímco modul `os` nabízí funkce pro interakci s operačním systémem, zejména pak v kontextu souborového systému, modul `os.path` poskytuje směsici prvků pro manipulaci s řetězci (představující cesty) a několik funkcí pro pohodlnější práci se souborovým systémem. Funkce `os.path.abspath()` vrací absolutní cestu k zadanému argumentu s tím, že se odstraní redundantní oddělovače cest a prvky „..“. Funkce `os.path.split()` vrátí dvojici, v níž první prvek obsahuje cestu a druhý název souboru (který může být prázdný, je-li zadána cesta bez názvu souboru). Tyto dvě části jsou též přístupné přímo prostřednictvím funkcí `os.path.basename()` a `os.path.dirname()`. Název souboru lze také rozdělit na dvě části, název a příponu, pomocí funkce `os.path.splitext()`. Funkce `os.path.join()` přijímá libovolný počet řetězců představujících cestu a vrací jedinou cestu používající oddělovač cesty specifický pro aktuální platformu.

Pokud potřebujeme více údajů o souboru či adresáři, můžeme použít funkci `os.stat()`. Pokud nám ale stačí jen jediný údaj, můžeme sáhnout po příslušné funkci modulu `os.path`, například `os.path.exists()`, `os.path.getsize()`, `os.path.isfile()` nebo `os.path.isdir()`.

Modul `mimetypes` obsahuje funkci `mimetypes.guess_type()`, která se pokusí odhadnout typ MIME zadaného souboru.

Příklad: Moduly os a os.path

V tomto příkladu si ukážeme, jak pomocí modulů `os` a `os.path` vytvořit slovník se všemi soubory na zadané cestě, v němž je každý klíč názvem souboru (včetně cesty) a každá hodnota časovou známkou (vteřiny od začátku epochy) poslední modifikace příslušného souboru, a to pro:

```
date_from_name = {}
for name in os.listdir(path):
    fullname = os.path.join(path, name)
    if os.path.isfile(fullname):
        date_from_name[fullname] = os.path.getmtime(fullname)
```

Tento kód je docela jednoduchý, lze jej ale použít pouze pro soubory nebo jediný adresář. Pokud potřebujeme projít celý adresářový strom, můžeme sáhnout po funkci `os.walk()`.

Zde je úryvek kódu z programu `finddup.py`.* Vytváříme v něm slovník, v němž je každý klíč dvojicí (velikost souboru, název souboru) s tím, že název souboru neobsahuje cestu, a každá hodnota seznamem úplných názvů souborů, které se shodují s názvem souboru svého klíče a mají stejnou velikost:

```
data = collections.defaultdict(list)

for root, dirs, files in os.walk(path):
    for filename in files:
        fullname = os.path.join(root, filename)
        key = (os.path.getsize(fullname), filename)
        data[key].append(fullname)
```

V každém adresáři vrátí funkce `os.walk()` kořen a dva seznamy, jeden s podadresáři v zadaném adresáři a druhý se soubory v tomto adresáři. Pro získání úplné cesty pro daný název souboru stačí zkombinovat kořen a název souboru. Všimněte si, že pro zanoření do podadresářů nemusíme používat rekurzi – tu za nás obstará funkce `os.walk()`. Jakmile vygenerujeme data, můžeme je projít a vytvořit zprávu ohledně možných duplicitních souborů:

```
for size, filename in sorted(data):
    names = data[(size, filename)]
    if len(names) > 1:
        print("{filename} ({size} bajtů) může být duplicitní "
              "({0} souborů):".format(len(names), **locals()))
        for name in names:
            print("\t{0}".format(name))
```

Klíči slovníku jsou n-tice (velikost, název soubor), a proto pro získání seřazení dat podle velikosti nemusíme použít klíčovou funkci. Má-li kterákoli n-tice (velikost, název souboru) ve svém seznamu více než jeden název souboru, může jít o duplicity.

```
...
shell32.dll (8460288 bajtů) může být duplicitní (2 soubory):
\windows\system32\shell32.dll
\windows\system32\dllcache\shell32.dll
```

Jedná se o poslední prvek z 3282 řádků výstupu vytvořeného spuštěním programu `finddup.py \ windows` v systému Windows.

Sítě a Internet

Balíčky a moduly pro práci se sítí a Internetem jsou hlavní součástí standardní knihovny Pythonu. Na nejnižší úrovni je modul `socket`, který poskytuje nejzákladnější síťové funkční prvky s funkcemi pro vytváření soketů, vyhledávání v systémech DNS (Domain Name System – systém doménových jmen) a zpracování IP adres (Internet Protocol – internetový protokol). Šifrované a autentizované

* Mnohem sofistikovanější program pro hledání duplicit s názvem `findduplicates-t.py`, který používá více vláken a kontrolu MD5, si ukážeme v lekci 10.

sokety lze zřizovat pomocí modulu `ssl`. Modul `socketserver` poskytuje servery TCP (Transmission Control Protocol – protokol pro řízení přenosu) a UDP (User Datagram Protocol – uživatelský datagramový protokol). Tyto servery dokážou zpracovat požadavky přímo nebo mohou pro zpracování každého požadavku vytvořit samostatný proces (rozvětvením procesu) nebo samostatné vlákno. Asynchronní zpracování soketů na straně klienta i serveru lze realizovat pomocí modulu `asyncore` a také vysokoúrovňového modulu `asynchat`, který je postaven na modulu `asyncore`.

Python definuje rozhraní WSGI (Web Server Gateway Interface – rozhraní brány webového serveru) poskytující standardní rozhraní mezi webovými servery a webovými aplikacemi napsanými v Pythonu. K podpoře tohoto standardu nabízí balíček `wsgiref` referenční implementaci rozhraní WSGI, která obsahuje moduly pro poskytování serverů HTTP splňující standard WSGI a pro zpracování hlaviček odpovědi a skriptů CGI (Common Gateway Interface – obecné rozhraní brány). Kromě toho modul `http.server` nabízí server HTTP, kterému lze předat obsluhu požadavků (k dispozici je též jedna standardní) pro spouštění skriptů CGI. Moduly `http.cookies` a `http.cookiejar` poskytují funkce pro správu souborů cookie a moduly `cgi` a `cgitb` nabízejí podporu skriptů CGI.

Klientský přístup k požadavkům protokolu HTTP nabízí modul `http.client`, i když balíček na vyšší úrovni `urllib` obsahuje moduly `urllib.parse`, `urllib.request`, `urllib. response`, `urllib.error` a `urllib.robotparser`, které poskytují snadný a pohodlnější přístup k adresám URL. Stažení souboru z Internetu tedy může být takto jednoduché:

```
fh = urllib.request.urlopen("http://www.python.org/index.html")
html = fh.read().decode("utf8")
```

Funkce `urllib.request.urlopen()` vrací objekt, který se chová podobně jako objekt souboru otevřený v režimu binárního čtení. Zde získáváme z webu Pythonu soubor `index.html` (jako objekt typu `bytes`) a ukládáme jej jako řetězec do proměnné `html`. Pomocí funkce `urllib.request.urlretrieve()` lze stáhnout soubory a uložit je do místních souborů.

Dokumenty HTML a XHTML je možné analyzovat pomocí modulu `html.parser`, adresy URL analyzujeme a vytváříme pomocí modulu `urllib.parse` a soubory `robots.txt` lze analyzovat modulem `urllib.robotparser`. Data ve formátu JSON (JavaScript Object Notation – objektová notace JavaScriptu) můžeme číst a zapisovat prostřednictvím modulu `json`.

Knihovna nabízí kromě podpory serveru a klienta HTTP také podporu pro volání XML-RPC (Remote Procedure Call – vzdálené volání procedury), kterou zajišťují moduly `xmlrpc.client` a `xmlrpc.server`. Funkčnost na straně klienta pro protokol FTP (File Transfer Protocol – protokol pro přenos souborů) poskytuje modul `ftplib`, pro protokol NNTP (Network News Transfer Protocol – přenosový protokol pro síťové diskuzní skupiny) modul `nntplib` a pro protokol TELNET modul `telnetlib`.

Modul `smtp` nabízí server SMTP (Simple Mail Transfer Protocol – jednoduchý protokol pro přenos pošty) a pro e-mailové klienty slouží moduly `smtplib` pro protokol SMTP, `imaplib` pro protokol IMAP4 (Internet Message Access Protocol – protokol pro přístup k internetové poště) a `poplib` pro protokol POP3 (Post Office Protocol – protokol pro příjem pošty). K poštovním schránkám v nejrůznějších formátech lze přistupovat pomocí modulu `mailbox`. Jednotlivé zprávy (včetně několikadílných zpráv) lze vytvářet a upravovat prostřednictvím modulu `email`.

Jsou-li pro tuto oblast balíčky a moduly standardní knihovny nedostatečně, je zde ještě projekt Twisted (*www.twistedmatrix.com*), který nabízí ucelenou síťovou knihovnu třetí strany. K dispozici je také řada knihoven pro webové programování třetích stran, mezi něž patří například Django (*www.djangoproject.com*) a Turbogears (*www.turbogears.org*) pro tvorbu webových aplikací a dále knihovny Plone (*www.plone.org*) a Zope (*www.zope.org*), které poskytují kompletní webové rámce a systémy pro správu obsahu. Všechny tyto knihovny jsou napsány v Pythonu.

XML

Dokumenty XML se obvykle analyzují dvěma způsoby. Jeden z nich je založen na modelu DOM (Document Object Model – objektový model dokumentu) a druhý na rozhraní SAX (Simple API for XML – jednoduché rozhraní API pro XML). K dispozici jsou dva analyzátory modelu DOM, jeden v modulu `xml.dom` a druhý v modulu `xml.dom.minidom`. Analyzátor pro rozhraní SAX nabízí modul `xml.sax`. Modul `xml.sax.saxutils` jsme již použili kvůli funkci `xml.sax.saxutils.escape()` (pro zakódování znaků „&“, „<“ a „>“ pro dokument XML). K dispozici je též funkce `xml.sax.saxutils.quoteattr()`, která provádí to stejné, ale navíc zakóduje uvozovky (aby byl zadaný text vhodný pro atributy značek), a funkce `xml.sax.saxutils.unescape()`, která provádí obrácený převod (dekódování).

Dále jsou zde dva další analyzátory. Modul `xml.parsers.expat` lze použít pro analýzu dokumentů XML pomocí knihovny expat, ovšem za předpokladu, že je tato knihovna k dispozici, a prostřednictvím modulu `xml.etree.ElementTree` můžeme analyzovat dokumenty XML s použitím jistého druhu rozhraní ve stylu slovníku a seznamu. (Samotné analyzátory modelu DOM a elementových stromů standardně používají analyzátor `expat`.)

Ručnímu zapisování kódu XML a zapisování kódu XML pomocí modelu DOM a elementových stromů a analyzování pomocí analyzátorů pro model DOM, rozhraní SAX a elementové stromy se budeme věnovat v lekci 7.

K dispozici je také knihovna třetí stránky lxml (*www.codespeak.net/lxml*), která o sobě tvrdí, že je „knihovnou s nejbohatší výbavou a nejsnadnějším používáním určenou pro práci s XML a HTML v jazyku Python“. Tato knihovna nabízí rozhraní, které je v podstatě nadmnožinou toho, co poskytuje modul pro práci s elementovými stromy, společně s dalšími prvky, mezi něž patří podpora pro jazyk XPath, XSLT a řadu dalších technologií kolem jazyka XML.

Příklad: Modul xml.etree.ElementTree

Analyzátory pro model DOM a rozhraní SAX poskytují rozhraní API, na která jsou zkušení programátoři pracující s jazykem XML zvyklí. Modul `xml.etree.ElementTree` naproti tomu nabízí přístup k analýze a zápisu kódu jazyka XML, který více odpovídá povaze programování v jazyku Python. Modul pro práci s elementovými stromy je poměrně nedávným přírůstkem do standardní knihovny*, a proto může být některým čtenářům neznámý. Z tohoto důvodu si zde ukážeme velice krátký příklad, abychom měli ponětí, o co vlastně jde. Lekce 7 nabízí mnohem hutnější příklad a pro srovnání také kód používající model DOM a rozhraní SAX.

* Modul `xml.etree.ElementTree` se poprvé objevil v Pythonu 2.5.

Webová stránka americké vládní organizace NOAA (National Oceanic and Atmospheric Administration – Národní úřad pro výzkum oceánů a atmosféry) poskytuje pestrou škálu dat, včetně souboru XML, který uvádí meteorologické stanice ve Spojených státech. Tento soubor obsahuje více než 20 000 řádků a najdeme v něm podrobné informace o přibližně dvou tisícovkách stanic. Zde je typický záznam:

```
<station>
    <station_id>KBOS</station_id>
    <state>MA</state>
    <station_name>Boston, Logan International Airport</station_name>
    ...
    <xml_url>http://weather.gov/data/current_obs/KBOS.xml</xml_url>
</station>
```

Několik řádků jsme vynechali a zredukovali jsme také odsazení používané v tomto souboru. Celý soubor má přibližně 840 KB, takže jsme jej zkomprimovali pomocí programu **gzip** na přijatelnějších 72 KB. Jenže analyzátor elementových stromů vyžaduje buď název souboru, nebo objekt souboru, který má načíst. Komprimovaný soubor mu ale předat nemůžeme, protože ten by se mu jevil jen jako náhodná binární data. Tento problém můžeme vyřešit pomocí dvou počátečních kroků:

```
binary = gzip.open(filename).read()
fh = io.StringIO(binary.decode("utf8"))
```

Funkce `gzip.open()` modulu `gzip` je podobná vestavěné funkci `open()` s výjimkou toho, že čte soubory zkomprimované algoritmem gzip (tj. s příponou `.gz`) jako binární data. Data potřebujeme mít k dispozici ve formě souboru, s nímž dokáže pracovat analyzátor elementových stromů, a proto binární data převádíme pomocí metody `bytes.decode()` na řetězec s kódováním UTF-8 (které používají soubory XML), z něhož pak vytváříme objekt typu `io.StringIO` chovající se jako soubor.

`io.StringIO` ➤ 210

typ `bytes` ➤ 286

```
tree = xml.etree.ElementTree.ElementTree()
root = tree.parse(fh)
stations = []
for element in tree.getiterator("station_name"):
    stations.append(element.text)
```

Zde vytváříme nový objekt typu `xml.etree.ElementTree.ElementTree` a předáváme mu objekt souboru, z něhož má načíst data XML, která chceme analyzovat. Z pohledu analyzátoru elementových stromů se jedná o objekt souboru otevřený pro čtení, ačkoliv ve skutečnosti jde o řetězec uvnitř objektu typu `io.StringIO`. Potřebujeme extrahovat názvy všech meteorologických stanic, což provedeme snadno pomocí metody `xml.etree.ElementTree.ElementTree.getiterator()`, která vrací iterátor, který vrací všechny objekty typu `xml.etree.ElementTree.Element` se zadaným názvem značky. Pro získání textu pak už jen stačí použít atribut `text` daného elementu. Podobně jako při použití funkce `os.walk()` nemusíme ani zde provádět žádnou rekurzi, o kterou se za nás postará iterátor. Dokonce ani nemusíme uvádět značku – v takovém případě pak iterátor vrátí každý element v celém dokumentu XML.

Další moduly

Na probrání všech 200 balíčků a modulů dostupných ve standardní knihovně zde není místo. Nicméně tento všeobecný přehled by měl být dostačující, abyste získali jistou představu o tom, co vše knihovna nabízí a jaké jsou její klíčové balíčky v hlavních oblastech jejího nasazení. Několik dalších oblastí si ještě probereme v této poslední podčásti.

V předchozí části jsme si ukázali, jak snadné je vytvářet a spouštět testy v dokumentačních řetězcích pomocí modulu `doctest`. Knihovna kromě toho nabízí rámec pro testování jednotek poskytovaný modulem `unittest`. Jedná se o verzi testovacího rámce JUnit z Javy přenesenou do prostředí Pythonu. Modul `doctest` nabízí také jistou základní integraci s modulem `unittest`. (Testování se budeme detailně věnovat v lekci 9.) K dispozici je též několik testovacích rámců třetích stran, například py.test z *codespeak.net/py/dist/test/test.html* a nose z *code.google.com/p/python-nose*.

Neinteraktivní aplikace, jako jsou kupříkladu servery, používají pro hlášení problémů obvykle zápis do souborů protokolu. Modul `logging` nabízí jednotné rozhraní pro protokolování a kromě možnosti protokolovat do souborů dokáže zaznamenávat zprávy také pomocí požadavků HTTP GET či POST nebo pomocí e-mailu či soketů.

Knihovna nabízí spoustu modulů pro introspekci a manipulaci s kódem. Většina z nich jde sice nad rámec této knihy, je zde ale jeden, který stojí za zmínku. Modul `pprint` obsahuje funkce pro „hezké vypisování" objektů Pythonu včetně datových typů představujících kolekce, což se někdy hodí pro ladění. Jednoduché použití modulu `inspect`, který nahlíží do živých objektů, si ukážeme v lekci 8.

Modul `threading` poskytuje podporu pro vytváření vícevláknových aplikací a modul `queue` poskytuje tři odlišné druhy front bezpečných vzhledem k vícevláknovému zpracování. Práci s vlákny budeme probírat v lekci 10.

Python neobsahuje žádnou nativní podporu pro programování grafických uživatelských rozhraní (GUI), programy napsané v Pythonu však mohou využít několik knihoven GUI. Knihovna Tk je dostupná prostřednictvím modulu `tkinter` a obvykle se instaluje jako standard. S programováním grafických uživatelských rozhraní se seznámíme v lekci 15.

Modul `abc` (Abstract Base Class – abstraktní bázová třída) nabízí funkce nezbytné pro vytváření abstraktní bázových tříd. Tento modul si rozebereme v lekci 8.

Mělké a hloubkové kopírování
➢ 146

Modul `copy` poskytuje funkce `copy.copy()` a `copy.deepcopy()`, které jsme probírali v lekci 3.

Přístup k *cizím funkcím*, tedy k funkcím ve sdílených knihovnách (soubory `.dll` ve Windows, soubory `.dylib` v Mac OS X a soubory `.so` v Linuxu), lze realizovat prostřednictvím modulu `ctypes`. Python dále nabízí rozhraní API pro jazyk C, je tedy možné vytvořit vlastní datové typy a funkce v jazyku C a zpřístupnit je Pythonu. Modul `ctypes` a rozhraní API Pythonu pro jazyk C je nad rámec této knihy.

Pokud žádný z balíčků a modulů zmíněných v této části nenabízí funkčnost, kterou potřebujete, pak ještě předtím, než si začnete potřebné funkce vytvářet sami, nahlédněte do rozcestníku dokumentace k Pythonu (Global Module Index) a podívejte se, zda není k dispozici nějaký vhodný modul, protože zde nejsme schopni zmínit se o každém z nich. Pokud ani zde nenajdete, co hledáte, zkuste se podívat na rozcestník balíčků Pythonu (*pypi.python.org/pypi*), který obsahuje několik tisíc doplňků

Pythonu od malých modulů tvořených jedním souborem až po rozsáhlé knihovny a rámcové balíčky obsahující desítky až stovky modulů.

Shrnutí

Na začátku této lekce jsme se seznámili s několika syntaxemi pro importování balíčků, modulů a objektů uvnitř modulů. Řekli jsme si, že řada programátorů používá pro zabránění kolizím názvů pouze syntaxi `import importovatelný_prvek` a že musíme být opatrní, abychom nedali svému programu či modulu stejný název, jaký má zastřešující modul či adresář Pythonu.

Probírali jsme také balíčky Pythonu. Jedná se o obyčejné adresáře se souborem `__init__.py` a s jedním nebo více moduly `.py`. Soubor `__init__.py` může být prázdný, ale kvůli podpoře syntaxe `from importovatelný_prvek import *` může obsahovat speciální proměnnou `__all__` nastavenou na seznam názvů modulů. Do souboru `__init__.py` můžeme také umístit libovolný inicializační kód. Řekli jsme si, že balíčky lze vnořovat prostým vytvářením podadresářů, z nichž každý obsahuje svůj vlastní soubor `__init__.py`.

Popsali jsme si dva vlastní moduly. První poskytoval jen pár funkcí a měl velice jednoduché dokumentační testy. Druhý byl mnohem propracovanější a obsahoval jednu vlastní výjimku, pomocí dynamické tvorby funkcí vytvářel funkci s implementací specifickou pro používanou platformu, obsahoval soukromá globální data, volání inicializační funkce a více propracovanějších dokumentačních testů.

Přibližně polovina lekce byla věnována přehledu standardní knihovny Pythonu. Zmínili jsme se o několika modulech pro práci s řetězci a ukázali jsme si pár příkladů se třídou `io.StringIO`. V jednom příkladu jsme viděli, jak zapisovat text do souboru buď pomocí vestavěné funkce `print()`, nebo metody `write()` objektu souboru a jak místo skutečného souboru použít objekt typu `io.StringIO`. V předchozích lekcích jsme zpracovávali volby na příkazovém řádku tak, že jsme sami analyzovali data v seznamu `sys.argv`, avšak při probírání podpory knihovny pro programování na příkazovém řádku jsme se seznámili s modulem `optparse`, který výrazným způsobem zjednodušuje práci s argumenty příkazového řádku. Od této chvíle budeme tento modul využívat mnohem častěji.

Zmínili jsme se o skvělé podpoře Pythonu pro čísla a o číselných typech knihovny a jejich třech modulech s matematickými funkcemi a také o podpoře pro vědecké a inženýrské výpočty poskytované projektem SciPy. Stručně jsme si popsali třídy knihovny a třetích stran pro práci s datem a časem a ukázali jsme si, jak získat aktuální datum a čas a jak provádět převody mezi typem `datetime.datetime` a počtem vteřin od počátku epochy. Dále jsme probírali další datové typy představující kolekce a algoritmy pro práci s uspořádanými posloupnostmi nabízenými standardní knihovnou společně s několika příklady použití funkcí modulu `heapq`.

Probírali jsme moduly, které podporují nejrůznější kódování souborů (kromě kódování znaků), a také moduly pro zabalování a rozbalování nejznámějších archivačních formátů a moduly podporující zvuková data. Ukázali jsme si, jak pomocí kódování Base64 uložit binární data do souboru `.py` a také program pro rozbalování archivů tarball. K dispozici máme též značnou podporu pro práci s adresáři a soubory, přičemž vše je abstrahováno do funkcí nezávislých na používané platformě. Ukázali jsme si příklady pro vytváření slovníku s klíči tvořenými názvy souborů a hodnotami ve formě časové

známky poslední modifikace a pro provádění rekurzivního procházení adresáře za účelem nalezení případných duplicitních souborů na základě jejich názvu a velikosti.

Velká část knihovny je věnována síťovému a internetovému programování. Stručně jsme prozkoumali, co vše je k dispozici, od holých soketů (včetně šifrovaných soketů) přes servery TCP a UDP až po server HTTP a podporu pro rozhraní WSGI. Zmínili jsme se také o modulech pro práci se soubory cookie, skripty CGI a daty HTTP a pro analyzování HTML, XHTML a adres URL. Mezi další zmíněné moduly patří ty, které podporují vzdálená volání procedur XML-RPC, protokoly vyšší úrovně, jako jsou FTP a NNTP, e-mailové klienty a servery používající protokol SMTP a podporu protokolů IMAP4 a POP3 na straně klientů.

Zmínili jsme též komplexní podporu knihovny pro zapisování a analyzování kódu jazyka XML včetně analyzátorů modelu DOM, rozhraní SAX a elementových stromů a modulu `expat`. Ukázali jsme si také jeden příklad využívající modul pro analýzu elementových stromů. Dále jsme se zmínili o několika z mnoha dalších balíčků a modulů nabízených standardní knihovnou.

Standardní knihovna Pythonu představuje neobyčejně užitečný zdroj, který může ušetřit značné množství času a úsilí a v řadě případů nám díky nabízeným funkčním prvkům umožňuje psát mnohem menší programy. Kromě toho existují doslova tisíce balíčků třetích stran vyplňujících mezery, které bychom mohli ve standardní knihovně objevit. Díky této předdefinované funkčnosti se můžeme v mnohem větší míře zaměřit na to, co chceme, aby náš program prováděl, přičemž necháme na modulech knihovny, aby se postaraly o většinu detailů.

Tato Lekce nás přivedla na konec základů procedurálního programování. V pozdějších lekcích, zvláště v lekci 8, se podíváme na pokročilejší a specializovanější procedurální techniky a v následující lekci se seznámíme s objektově orientovaným programováním. Python sice můžeme používat jen jako čistě procedurální jazyk, což může být zvláště u malých prográmků praktické, ale pro středně velké až rozsáhlé programy, pro vlastní balíčky a moduly a pro dlouhodobou udržovatelnost je objektově orientovaný přístup daleko lepší. Naštěstí je vše, co jsme se dosud naučili, užitečné a relevantní také v oblasti objektově orientovaného programování, takže v následujících lekcích budeme pokračovat v budování našich znalostí a dovedností v oblasti jazyka Python na všech dosud položených základech.

Cvičení

Napište program vypisující obsah adresáře jako příkaz `dir` ve Windows nebo `ls` v Unixu. Výhoda vytvoření našeho vlastního programu spočívá v tom, že do něj můžeme zabudovat preferované výchozí chování a hodnoty a používat na všech platformách stejný program bez toho, abychom si museli pamatovat odlišnosti mezi příkazy `dir` a `ls`. Vytvořte program, který podporuje následující rozhraní:

```
Usage: ls.py [volby] [cesta1 [cesta2 [... cestaN]]]
Cesty jsou volitelné. Nejsou-li zadány, použije se aktuální adresář.
Options:
  -h, --help            show this help message and exit
  -H, --hidden          zobrazí skryté soubory [výchozí: off]
  -m, --modified        zobrazí datum a čas poslední modifikace [výchozí: off]
```

```
  -o ORDER, --order=ORDER
                        seřadí výstup podle ('name', 'n', 'modified', 'm',
                        'size', 's') [výchozí: name]
  -r, --recursive       sestupuje rekurzivně  do podadresářů [výchozí: off]
  -s, --sizes           zobrazí velikosti [výchozí: off]
 (Výstup programu byl pro účely knihy trošku upraven.)
Zde je ukázkový výstup na malém adresáři s použitím příkazového řádku ls.py -ms -os
Graphics/:
2009-11-20 10:04:52              49 Graphics/__init__.py
2009-11-20 10:04:52             351 Graphics/Bmp.py
2009-11-20 10:04:52             351 Graphics/Xpm.py
2009-11-20 10:04:52             357 Graphics/Jpeg.py
2009-11-20 10:04:52             357 Graphics/Tiff.py
2009-11-20 10:04:52             368 Graphics/Png.py
                                    Graphics/Vector/
6 souborů, 1 adresář
```

Na příkazovém řádku jsme použili seskupování voleb (automaticky se o to postará modul `optparse`), stejného výsledku bychom dosáhli také při použití samostatných voleb, například `ls.py -m -s -os Graphics/`, nebo dokonce s použitím ještě většího seskupení `ls.py -msos Graphics/` nebo s použitím dlouhých voleb `ls.py --modified --sizes --order=size Graphics/` nebo libovolnou kombinací těchto možností. Všimněte si, že „skryté" soubory či adresáře definujeme jako soubory či adresáře, jejichž název začíná tečkou (.).

Toto cvičení je docela náročné. Budete si muset pročíst dokumentaci k modulu `optparse`, z níž se dozvíte, jak definovat volby, které nastavují hodnotu `True`, a jak nabídnout fixní seznam možností. Pokud uživatel nastaví rekurzivní volbu, budete muset zpracovat soubory pomocí funkce `os.walk()`. V opačném případě musíte použít funkci `os.listdir()` a zpracovat soubory a adresáře ručně.

Jeden ze složitějších aspektů tkví v tom, jak zabránit, aby se při rekurzi použily skryté adresáře. Lze je odříznout od seznamu `dirs` funkce `os.walk()` (která je tudíž přeskočí) modifikací tohoto seznamu. Buďte však opatrní, abyste nepřiřazovali do samotné proměnné `dirs`, protože tím by se seznam, na který ukazuje, nijak nezměnil, ale jen by se (zbytečně) nahradil jiným. V modelovém řešení přiřazujeme do řezu celého seznamu, což znamená `dirs[:] = [dir for dir in dirs if not dir.startswith(".")]`.

Znaky pro seskupování čísel představujících velikosti souboru získáte nejlépe importem modulu `locale`, zavoláním funkce `locale.setlocale()` pro získání výchozího národního prostředí uživatele a použitím formátovacího znaku `n`. Výsledný program `ls.py` má přibližně 130 řádků rozdělených do čtyř funkcí.

`locale.setlocale()` ➤ 90

LEKCE 6

Objektově orientované programování

V této lekci:

- Objektově orientovaný přístup
- Vlastní třídy
- Vlastní třídy představující kolekce

Ve všech předchozích lekcích jsme v hojné míře používali objekty, avšak náš styl programování byl striktně procedurální. Python je jazyk podporující více paradigmat. Umožňuje nám programovat v procedurálním, objektově orientovaném a funkcionálním stylu nebo v libovolné směsici těchto stylů, protože nás nijak nenutí programovat jedním určitým způsobem.

Není vůbec žádný problém napsat libovolný program v procedurálním stylu, zvláště pak u malých prográmků (řekněme tak do 500 řádků kódu). Avšak u většiny programů, především u programů středních až rozsáhlých, nabízí objektově orientované programování řadu výhod.

V této lekci se budeme věnovat všem základním principům a technikám pro realizaci objektově orientovaného programování v jazyku Python. První část je určena především těm, kteří jsou méně zkušení, a také pro ty, kteří byli dosud zvyklí jen na procedurální programování (např. v jazyku C nebo Fortran). Nejdříve se podíváme na několik problémů, které mohou vyvstat u procedurálního programování a které dokáže objektově orientované programování vyřešit. Poté si stručně popíšeme přístup jazyka Python k objektově orientovanému programování a vysvětlíme si související terminologii. Pak už začínají dvě hlavní části této lekce.

Ve druhé části se budeme věnovat vytváření vlastních datových typů, které uchovávají jediné prvky (i když tyto prvky mohou samy mít mnoho atributů) a ve třetí části se podíváme na tvorbu vlastních datových typů představujících kolekce, které dokážou uchovávat libovolný počet objektů libovolného typu. Tyto části pokrývají většinu aspektů objektově orientovaného programování v jazyku Python, i když některá pokročilejší témata si necháme až na lekci 8.

Objektově orientovaný přístup

V této části se podíváme na některé problémy čistě procedurálního přístupu. Představte si situaci, kdy potřebujete reprezentovat kruhy – spoustu kruhů. Mezi minimální data nezbytná pro reprezentaci kruhu patří jeho pozice (x, y) a poloměr. Jednoduchý přístup spočívá v použití n-tice se třemi prvky pro každý kruh:

```
circle = (11, 60, 8)
```

Nevýhodou tohoto přístupu je to, že není jasné, co každý element n-tice představuje. Může to znamenat (x, y, poloměr) nebo také (poloměr, x, y). Další nevýhoda tkví v tom, že k elementům musíme přistupovat pouze přes indexové pozice. Máme-li dvě funkce `distance_from_origin(x, y)` a `edge_distance_from_origin(x, y, poloměr)`, pak musíme pro jejich zavolání s n-ticí kruhu použít rozbalení n-tice:

```
distance = distance_from_origin(*circle[:2])
distance = edge_distance_from_origin(*circle)
```

U obou těchto volání předpokládáme, že n-tice kruhu mají tvar (x, y, poloměr). Tento problém s pořadím prvků a použitím rozbalování můžeme vyřešit pomocí pojmenované n-tice:

```
import collections
Circle = collections.namedtuple("Circle", "x y radius")
circle = Circle(13, 84, 9)
distance = distance_from_origin(circle.x, circle.y)
```

Nyní můžeme vytvářet n-tice `Circle` se třemi prvky s pojmenovanými atributy, díky kterým je volání funkcí mnohem srozumitelnější, protože pro přístup k prvkům můžeme používat jejich jména. Problémy však stále zůstávají. Neexistuje například nic, co by nám zabránilo ve vytvoření neplatného kruhu:

```
circle = Circle(33, 56, -5)
```

Kruh se záporným poloměrem je nesmysl, zde se však přesto vytvoří pojmenovaná n-tice `circle`, aniž by došlo k vyvolání výjimky, tedy stejně, jako kdyby byl poloměr zadán jako proměnná obsahující záporné číslo. Tato chyba se projeví až při zavolání funkce `edge_distance_from_origin()`, ovšem jen tehdy, pokud tato funkce skutečně kontroluje, zda kruh neobsahuje záporný poloměr. Tato neschopnost validace při vytvoření objektu je pravděpodobně nejhorším aspektem při čistě procedurálním přístupu.

Pokud chceme, aby byly kruhy měnitelné, abychom je tak mohli změnou souřadnic přesouvat nebo u nich změnou poloměru měnit velikost, můžeme k tomuto účelu použít soukromou metodu `collections.namedtuple._replace()`:

```
circle = circle._replace(radius=12)
```

Stejně jako jsme vytvořili n-tici `Circle`, můžeme nastavit i neplatná data, aniž by nás cokoliv zastavilo (nebo varovalo).

Pokud by kruhy vyžadovaly množství změn, pak můžeme sáhnout po měnitelném datovém typu, jako je seznam:

```
circle = [36, 77, 8]
```

Zde ale nemáme žádnou ochranu před zadáním neplatných dat a pro přístup k prvkům podle jména nám nezbývá nic jiného než vytvořit nějaké konstanty, abychom pak mohli psát něco jako `circle[RADIUS] = 5`. Při používání seznamů ale vyvstávají další problémy. Můžeme třeba naprosto legálně zavolat metodu `circle.sort()`! Další možností je použít slovník, například `circle = dict(x=36, y=77, radius=8)`, opět ale nemáme žádnou možnost, jak zajistit zadání platného poloměru a jak zabránit volání nepatřičných metod.

Objektově orientované principy a terminologie

Potřebujeme nějaký způsob zabalení dat, která jsou nezbytná pro reprezentaci kruhu, a nějaký způsob pro omezení metod, které lze na tato data aplikovat, aby tak byly přístupné jen platné operace. Obou těchto věcí lze dosáhnout vytvořením vlastního datového typu `Circle`. Jak se tento datový typ vytváří, uvidíme v této části později, nejdříve si musíme udělat přípravu a vysvětlit si terminologii. Nebojte se, bude-li vám terminologie na první pohled nejasná. Vše se vyjasní, jakmile se dostaneme k příkladům.

Termíny *třída*, *typ* a *datový typ* používáme pro označení téhož pojmu. V jazyku Python můžeme vytvářet vlastní třídy, které jsou plně integrovány a které lze používat stejně jako vestavěné datové typy. Již jsme se setkali s celou řadou tříd, mezi něž patří například `dict`, `int` a `str`. Termín *objekt*, a případně také *instance*, používáme pro označení instance určité třídy. Například `5` je objekt typu `int` a `"obdélník"` je objekt typu `str`.

Většina tříd zapouzdřuje data a metody, které lze na tato data aplikovat. Například třída `str` uchovává jako svá data řetězec znaků ze znakové sady Unicode a podporuje metody, jako je `str.upper()`. Řada tříd podporuje také dodatečné možnosti. Můžeme tak například spojit dva řetězce (nebo libovolné dvě posloupnosti) pomocí operátoru + a zjistit délku posloupnosti pomocí vestavěné funkce `len()`. Tyto možnosti poskytují *speciální metody*, které jsou podobné normálním metodám, ale jejich jméno je předem definované a vždy začíná a končí dvěma podtržítky. Chceme-li například vytvořit třídu, která podporuje spojování pomocí operátoru + a funkci `len()`, pak v ní můžeme implementovat speciální metody `__add__()` a `__len__()`. A naopak bychom nikdy neměli definovat žádnou metodu, která začíná a končí dvěma podtržítky, pokud se nejedná o některou z předdefinovaných speciálních metod vhodných pro naši třídu. Díky tomu budeme mít jistotu, že nikdy nedojde ke konfliktům v pozdějších verzích Pythonu, i kdyby se v nich objevily nové předdefinované speciální metody.

Objekty mají obvykle atributy, přičemž metody jsou volatelné atributy a ostatní atributy jsou data. Například objekt typu `complex` má atributy `imag` a `real` a spoustu metod, včetně speciálních metod, jako je třeba `__add__()` a `__sub__()` (pro podporu binárních operátorů + a -), a běžných metod, jako je `conjugate()`. Datové atributy (často označované též jen jako „atributy“) se běžně implementují jako *proměnné instance*, což jsou proměnné, které existují samostatně v každém objektu daného typu. Ukážeme si příklady atributů a také způsobu, jak poskytovat datové atributy jako *vlastnosti*. Vlastnost je prvek dat objektu, který je přístupný jako proměnná instance, přičemž tento přístup je na pozadí zpracován určitou metodou. Jak sami uvidíme, použití vlastností usnadňuje validaci dat.

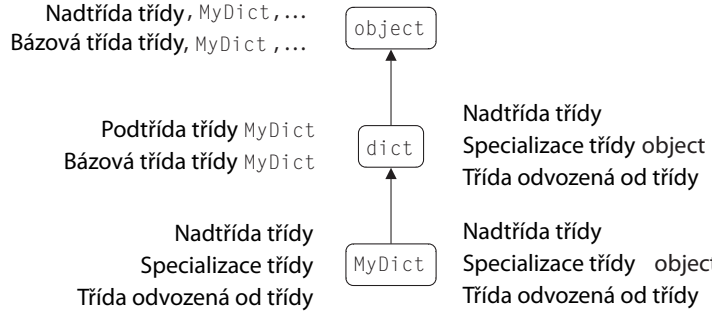
Uvnitř metod (což jsou funkce, jejichž prvním argumentem je instance, na které jsou volány) můžeme přistupovat k několika druhům proměnných. K proměnným instance daného objektu lze přistupovat kvalifikováním jejich jména se samotnou instancí. Uvnitř metod lze vytvářet lokální proměnné, k nimž přistupujeme bez kvalifikace. K proměnným třídy (někdy označovaným jako statické proměnné) přistupujeme kvalifikací jejich jména s názvem třídy a globální proměnné, což jsou proměnné modulu, jsou přístupné bez kvalifikace.

V literatuře o Pythonu se můžete setkat s koncepcí *oborů názvů* (namespace), což je mapování názvů na objekty. Těmito obory názvů jsou moduly. Například za příkazem import `math` můžeme přistupovat k objektům v modulu `math` jejich kvalifikováním s názvem jejich oboru názvů (např. `math.pi` a `math.sin()`). Podobně také třídy a objekty fungují jako obory názvů. Pokud například máme `z = complex(1, 2)`, pak obor názvů objektu `z` má dva atributy, k nimž můžeme přistupovat jako `z.real` a `z.imag`.

Jednou z výhod objektové orientace je to, že pokud máme třídu, můžeme ji *specializovat*. To znamená, že můžeme vytvořit novou třídu, která zdědí všechny atributy (data a metody) od původní třídy, abychom pak mohli metody přidat nebo nahradit nebo přidat nové proměnné instance. Můžeme *vytvořit podtřídu* (což je další označení pro specializaci) z libovolné třídy Pythonu, která může být vestavěná nebo součástí standardní knihovny nebo jednou z našich vlastních tříd.* Možnost vytvářet podtřídy je jednou z obrovských výhod nabízených objektově orientovaným programováním, protože tak můžeme jednoduše použít stávající třídu s vyzkoušenou a otestovanou funkčností jako základ pro novou třídu, která tu původní rozšiřuje přidáním nových datových atributů nebo nové funkčnosti velmi čistým a přímočarým způsobem. Kromě toho můžeme objekty naší nové třídy předávat funkcím a metodám, které byly napsány pro původní třídu a které budou i nadále fungovat správně.

* Z některých tříd knihovny implementovaných v jazyku C nelze vytvářet podtřídy. U takovýchto tříd je to uvedeno v jejich dokumentaci.

Termínem *bázová třída* (base class) označujeme třídu, která je zděděná. Bázová třída může být bezprostředním předkem nebo může být v hierarchii tříd na některém vyšším místě. Dalším termínem pro bázovou třídu je *nadtřída* (super class). Termín *podtřída* nebo *odvozená třída* budeme používat pro popsání třídy, která vznikla zděděním (tj. specializací) od jiné třídy. V Pythonu je každá vestavěná a knihovní třída a každá námi vytvořená třída přímo či nepřímo odvozená od nejzákladnější bázové třídy s názvem `object`. Obrázek 6.1 znázorňuje některé pojmy z terminologie dědičnosti.



Obrázek 6.1: Něco z terminologie objektově orientované dědičnosti

V podtřídě lze libovolnou metodu přepsat, což znamená reimplementovat (opětovně implementovat). To je stejné jako v jazyku Java (kromě metod Javy označených klíčovým slovem `final` jako „finální“).* Máme-li objekt typu `MyDict` (což je třída odvozená od typu `dict`) a zavoláme metodu, která je definovaná ve třídě `dict` i `MyDict`, zavolá Python správně verzi ve třídě `MyDict`. Tomuto principu se říká *dynamická vazba metod* (dynamic method binding) nebo též *polymorfizmus*. Pokud potřebujeme v reimplementované metodě zavolat verzi této metody v bázové třídě, můžeme tak učinit pomocí vestavěné funkce `super()`.

Python dále podporuje „kachní typování“ (duck typing), což lze stručně vysvětlit větou: „Jestliže to chodí jako kachna a kváká jako kachna, pak je to kachna“. Jinými slovy, pokud chceme zavolat určité metody na objektu, pak nezáleží na tom, jaké třídy je daný objekt, ale pouze na tom, zda má metody, které chceme zavolat. V předchozí lekci jsme viděli, že když potřebujeme objekt souboru, pak můžeme použít buď vestavěnou funkci `open()`, nebo vytvořit objekt typu `io.StringIO`, protože objekty typu `io.StringIO` mají stejné rozhraní API (Application Programming Interface – aplikační programovací rozhraní), což znamená stejné metody jako objekty souboru získané zavoláním funkce `open()` v textovém režimu.

Dědičnost se používá pro modelování vztahu „objekty třídy *jsou* objekty nadtřídy“ („is-a“), což znamená, že objekty třídy jsou v podstatě stejné jako objekty nadtřídy, ovšem s určitými změnami, jako jsou dodatečné atributy a metody. Další možností je použít *agregaci* (označovanou též jako *kompozice*). V takovém případě třída obsahuje jednu či více proměnných instance, které ukazují na objekty jiných tříd. Agregace se používá pro modelování vztahu „objekty třídy *mají* objekty jiné třídy“ („has-a“). V Pythonu používá každá třída dědičnost, protože všechny vlastní třídy mají jako svoji nejzákladnější bázovou třídu `object` a většina tříd používá také agregace, protože většina tříd má proměnné instance nejrůznějších typů.

* V terminologii jazyka C++ jsou všechny metody Pythonu virtuální.

Některé objektově orientované jazyky nabízejí dva principy, které jazyk Python nezná. Prvním je přetěžování, což znamená, že ve stejné třídě máme metody s týmž názvem, ale s odlišnými parametry. Ovšem díky všestranným možnostem práce s argumenty v jazyku Python nejde prakticky o žádné omezení. Druhým principem je řízení přístupu. Není zde žádný neprůstřelný mechanizmus pro zajištění soukromí dat. Pokud ale vytvoříme atributy (proměnné instance nebo metody), které začínají dvěma podtržítky, Python zabrání neúmyslnému přístupu, takže tyto atributy lze považovat za soukromé. (To se provádí pomocí techniky zvané komolení jmen – name mangling. Příklad si ukážeme v lekci 8.)

Názvy našich modulů mají první písmeno velké a stejně budeme pojmenovávat také své vlastní třídy. Můžeme definovat tolik tříd, kolik jen chceme, a to buď přímo v programu, nebo v modulech, přičemž názvy tříd nemusejí odpovídat názvům modulů a moduly mohou obsahovat libovolné množství definic tříd.

Viděli jsme, jak lze pomocí tříd vyřešit některé problémy, seznámili jsme se s nezbytnou terminologií a věnovali jsme se základní problematice, a proto se můžeme nyní pustit do tvorby několika svých vlastních tříd.

Vlastní třídy

V dřívějších lekcích jsme již vlastní třídy vytvářeli, jednalo se o naše vlastní výjimky. Zde jsou dvě nové syntaxe pro tvorbu vlastních tříd:

```
class názevTřídy:
    sada

class názevTřídy(bázové_třídy):
    sada
```

Podtřídy představující výjimky, které jsme vytvářeli, neobsahovaly žádné nové atributy (žádná data ani metody instance), a proto jsme použili sadu `pass` (tj. nic nového), což je jen jeden příkaz, který jsme tudíž umístili na stejný řádek jako příkaz `class`. Všimněte si, že stejně jako u příkazů `def` je i `class` příkaz, takže i třídy můžeme vytvářet dynamicky, pokud chceme. Metody třídy se vytvářejí pomocí příkazů `def` v sadě třídy. Instance třídy vytváříme zavoláním třídy s případnými argumenty. Například `x = complex(4, 8)` vytvoří komplexní číslo a nastaví odkaz na objekt `x` tak, aby na něj ukazoval.

Atributy a metody

Začněme velmi jednoduchou třídou `Point`, která uchovává souřadnice (x, y). Tato třída je umístěna v souboru `Shape.py` a její kompletní implementace (bez dokumentačních řetězců) vypadá takto:

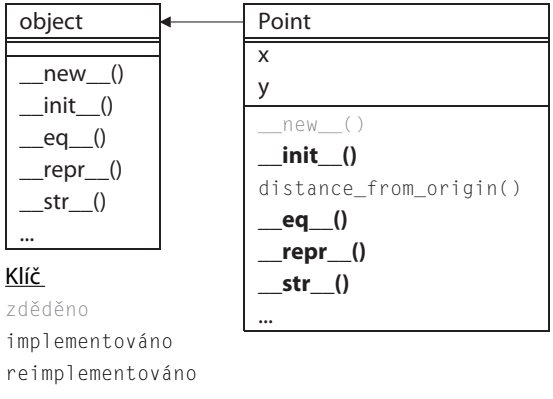
```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance_from_origin(self):
```

```
        return math.hypot(self.x, self.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

Neuvedli jsme žádné bázové třídy, a proto je třída `Point` přímou podtřídou třídy `object`. Mohli bychom tedy úplně stejně napsat `class Point(object)`. Dříve než se podíváme na jednotlivé metody, ukážeme si několik příkladů s praktickým použitím této třídy:

```
import Shape
a = Shape.Point()
repr(a)                         # vrátí: 'Point(0, 0)'
b = Shape.Point(3, 4)
str(b)                          # vrátí: '(3, 4)'
b.distance_from_origin()        # vrátí: 5.0
b.x = -19
str(b)                          # vrátí: '(-19, 4)'
a == b, a != b                  # vrátí: (False, True)
```

Třída `Point` má dva datové atributy `self.x` a `self.y` a pět metod (nepočítáme-li zděděné metody), z nichž čtyři jsou speciální. Všechny jsou uvedeny na obrázku 6.2. Jakmile importujeme modul `Shape`, můžeme třídu `Point` používat jako jakoukoli jinou třídu. K datovým atributům můžeme přistupovat přímo (např. `y = a.y`), přičemž třída se díky podpoře operátoru rovnosti (==) a možnosti vytvářet řetězce v reprezentační i řetězcové formě krásně integruje se všemi ostatními třídami Pythonu. Python je navíc dostatečně chytrý, takže umí na základě operátoru rovnosti odvodit operátor nerovnosti (!=). (Pokud chceme vyšší kontrolu, například v situacích, kdy tyto operátory nejsou vzájemným opakem, pak můžeme specifikovat oba operátory zvlášť.)



Obrázek 6.2: Hierarchie dědičnosti třídy Point

Python automaticky při volání metod doplňuje první argument, což je odkaz na objekt ukazující na samotný objekt (v jazycích C++ a Java je označován jako *this*). Tento parametr musí být v seznamu parametrů uveden a na základě nepsané dohody se označuje jako `self`. Všechny atributy objektu (atributy ve formě dat a metod) musejí být kvalifikovány objektem `self`. Oproti jiným jazykům je tak nutné psát o něco více kódu, výhodou je ale absolutní srozumitelnost. Při použití objektu `self` totiž vždy víme, že přistupujeme k atributu objektu.

K vytvoření objektu musíme provést dva kroky. Nejdříve je nutné vytvořit holý nebo též neinicializovaný objekt, který je pak nutné inicializovat a připravit na použití. Některé objektově orientované jazyky (např. C++ a Java) slučují tyto dva kroky do jednoho, ale Python je udržuje oddělené. Při vytváření objektu (např. `p = Shape.Point()`) se nejdříve zavolá speciální metoda `__new__()` pro vytvoření objektu a následně se zavolá speciální metoda `__init__()` pro jeho inicializaci.

Alternativní typ `FuzzyBool` ➤ 250

V praxi stačí u téměř všech tříd Pythonu, které vytvoříme, reimplementovat pouze metodu `__init__()`, protože metoda `object.__new__()` je skoro vždy dostatečná a volá se automaticky, pokud neposkytneme naši vlastní metodu `__new__()`. (V pozdější části této lekce si ukážeme vzácný příklad vyžadující reimplementaci metody `__new__()`.) To, že v podtřídě není nutné reimplementovat metody, je další výhodou objektově orientovaného programování. Je-li totiž metoda bázové třídy dostatečná, nemusíme ji ve své podtřídě reimplementovat. Tento princip funguje díky tomu, že když na nějakém objektu zavoláme metodu, která není ve třídě tohoto objektu implementována, tak Python automaticky prochází bázové třídy objektu, jejich bázové třídy a tak pořád dál, dokud volanou metodu nenajde. A pokud ji vůbec nenajde, vyvolá se výjimka `AttributeError`.

Pokud například provedeme příkaz `p = Shape.Point()`, začne Python hledat metodu `Point.__new__()`. Tuto metodu jsme neimplementovali, a proto ji Python hledá v bázových třídách třídy `Point`. V našem případě máme pouze jednu bázovou třídu `object`, která požadovanou metodu obsahuje, takže Python zavolá `object.__new__()` a vytvoří holý neinicializovaný objekt. Pak začne hledat metodu pro inicializaci s názvem `__init__()`, kterou jsme reimplementovali, a proto již nemusí dál hledat a zavolá `Point.__init__()`. Nakonec nastaví odkaz na objekt `p` tak, aby ukazoval na nově vytvořený a inicializovaný objekt typu `Point`.

Metody třídy `Point` jsou velmi krátké a nachází se několik stránek zpět, a proto si je při výkladu ukážeme znovu.

```
def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
```

V inicializační metodě vytváříme dvě proměnné instance `self.x` a `self.y` a přiřazujeme jim hodnoty parametrů `x` a `y`. Python tuto metodu nalezne při vytváření nového objektu typu `Point`, a proto se metoda `object.__init__()` nezavolá. To je dáno tím, že Python při nalezení požadované metody již dále nehledá.

Objektově orientovaní puristé mohou tuto metodu začínat voláním metody `__init__()` bázové třídy příkazem `super().__init__()`. Výsledkem tohoto volání funkce `super()` je volání metody `__init__()` bázové třídy. U tříd, které dědí přímo od třídy `object`, k tomu ale není žádný důvod. V této knize budeme volat metody bázových tříd pouze tehdy, bude-li to nezbytné – například při vytváření tříd, které jsou navržené pro vytváření podtříd, nebo při vytváření tříd, které nedědí přímo od třídy

`object`. Je to také do určité míry záležitostí stylu programování – je totiž naprosto odůvodnitelné volat na začátku vlastní metody `__init()__` vždy metodu `super().__init__()`.

```
def distance_from_origin(self):
    return math.hypot(self.x, self.y)
```

Jedná se o tradiční metodu provádějící výpočet na základě proměnných instance daného objektu. U metod je docela běžné, že jsou poměrně krátké a že mají jako argument pouze objekt, na kterém jsou volání, protože data, která metoda potřebuje, jsou často dostupná uvnitř objektu.

```
def __eq__(self, other):
    return self.x == other.x and self.y == other.y
```

Metody by neměly mít názvy, které začínají a končí dvěma podtržítky, pokud se nejedná o některou z předdefinovaných speciálních metod. Jak je patrné z tabulky 6.1, Python nabízí speciální metody pro všechny porovnávací operátory.

Tabulka 6.1: Speciální metody pro porovnávání

Speciální metoda	Použití	Popis
`__lt__(self, other)`	`x < y`	Vrací hodnotu `True`, je-li x menší než y.
`__le__(self, other)`	`x <= y`	Vrací hodnotu `True`, je-li x menší nebo stejné jako y.
`__eq__(self, other)`	*x == y*	Vrací hodnotu `True`, pokud se x rovná y.
`__ne__(self, other)`	`x != y`	Vrací hodnotu `True`, pokud se x nerovná y.
`__ge__(self, other)`	`x >= y`	Vrací hodnotu `True`, je-li x větší nebo stejné jako y.
`__gt__(self, other)`	`x > y`	Vrací hodnotu `True`, je-li x větší než y.

Všechny instance vlastních tříd podporují ve výchozím stavu operátor ==, který vrací hodnotu `False`, pokud neporovnáváme objekt sám se sebou. Toto chování můžeme přepsat reimplementací speciální metody `__eq__()`, jak jsme to udělali v tomto příkladu. Python pak automaticky doplní i metodu `__ne__()` (není rovno) pro operátor nerovnosti, pokud implementujeme metodu `__eq__()`, ale neimplementujeme metodu `__ne__()`.

Všechny instance vlastních tříd jsou ve výchozím stavu hashovatelné, takže na nich lze volat funkci `hash()` a je možné je použít jako klíče slovníku a ukládat do množin. Pokud ale reimplementujeme metodu `__eq__()`, přestanou být instance hashovatelné. Jak je možné tento problém vyřešit, si ukážeme při probírání třídy `FuzzyBool` v pozdější části.

Implementováním této speciální metody můžeme porovnávat objekty typu `Point`, pokud bychom se ale pokusili porovnat objekt typu `Point` s objektem jiného typu (např. typu `int`), obdrželi bychom výjimku `AttributeError` (protože objekty typu `int` nemají atribut `x`). Na druhou stranu můžeme porovnávat objekty typu `Point` s ostatními objekty, které shodou okolností mají atribut `x` (díky „kachnímu typování“ v jazyku Python), což ale může vést k překvapivým výsledkům.

FuzzyBool ➤ 249

Chceme-li zamezit nepatřičným porovnáváním, pak máme k dispozici několik možností. Jednou z nich je použití tvrzení, například `isinstance(other, Point)`. Další spočívá ve vyvolání výjimky

`TypeError` signalizující, že porovnání mezi danými dvěma typy není podporováno, například `if not isinstance(other, Point): raise TypeError()`. Třetí možností (která je navíc z hlediska Pythonu nejkorektnější) je provést následující: `if not isinstance(other, Point): return NotImplemented`. Je-li v tomto případě vrácena hodnota `NotImplemented`, Python se pokusí zavolat `other.__eq__(self)`, aby zjistil, zda typ `other` podporuje porovnávání s typem `Point`. Pokud žádná taková metoda neexistuje nebo pokud také vrátí hodnotu `NotImplemented`, tak to Python vzdá a vyvolá výjimku `TypeError`. (Je třeba poznamenat, že hodnotu `NotImplemented` mohou vracet pouze opětovné implementace speciálních metod pro porovnávání uvedených v tabulce 6.1.)

Vestavěná funkce `isinstance()` přijímá objekt a třídu (nebo n-tici tříd) a vrací hodnotu `True`, pokud je daný objekt instancí zadané třídy (nebo jedné ze tříd v zadané n-tici) nebo některé její bázové třídy (nebo některé bázové třídy ze tříd v zadané n-tici).

```
    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)
```

`str.format()` ➤ 83

Vestavěná funkce `repr()` zavolá pro zadaný objekt speciální metodu `__repr__()` a vrátí její výsledek. Vrácený řetězec může být dvojího druhu. Z prvního lze vyhodnocením pomocí vestavěné funkce `eval()` vytvořit objekt ekvivalentní tomu, na němž byla zavolána funkce `repr()`. Druhý se používá v případě, kdy nelze použít první druh řetězce. Příklad si ukážeme později. Zde je způsob, jak můžeme z objektu typu `Point` přejít na řetězec a zase zpět na objekt typu `Point`:

```
p = Shape.Point(3, 9)
repr(p)                                 # vrátí: 'Point(3, 9)'
q = eval(p.__module__ + "." + repr(p))
repr(q)                                 # vrátí: 'Point(3, 9)'
```

`import` ➤ 194

Použili jsme `import Shape`, a proto musíme při vyhodnocování funkcí `eval()` uvést název modulu. (To by nebylo nutné, pokud bychom provedli import odlišným způsobem, například `from Shape import Point`.) Python vybavuje každý objekt několika soukromými atributy, mezi něž patří také atribut `__module__`, což je řetězec uchovávající název modulu daného objektu, který je v tomto příkladu `"Shape"`.

Dynamické provádění kódu ➤ 334

Na konci tohoto úryvku máme dva objekty `p` a `q` typu `Point` se stejnými hodnotami atributů, takže se při porovnání rovnají. Funkce `eval()` vrátí výsledek provedení zadaného řetězce, který tudíž musí obsahovat platný příkaz jazyka Python.

```
    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

Vestavěná funkce `str()` funguje podobně jako funkce `repr()`, na zadaném objektu ale zavolá speciální metodu `__str__()`. Výsledkem by měl být řetězec srozumitelný pro člověka, u kterého se neočekává, že bude vhodný pro zadání funkce `eval()`. Navážeme-li na předchozí příklad, pak při volání `str(p)` (nebo `str(q)`) obdržíme řetězec `'(3, 9)'`.

Prošli jsme si jednoduchou třídu `Point` a podívali jsme se také na podrobnosti, o kterých je sice dobré vědět, ale které lze povětšinou nechat v pozadí. Třída Point uchovává souřadnice (x, y), což je základní složka, kterou potřebujeme pro reprezentaci kruhu, kterému jsme se věnovali na začátku

této lekce. V následujícím oddílu si ukážeme, jak vytvořit vlastní třídu `Circle`, kterou odvodíme od třídy `Point`, takže nebudeme muset duplikovat kód pro atributy `x` a `y` nebo pro metodu `distance_from_origin()`.

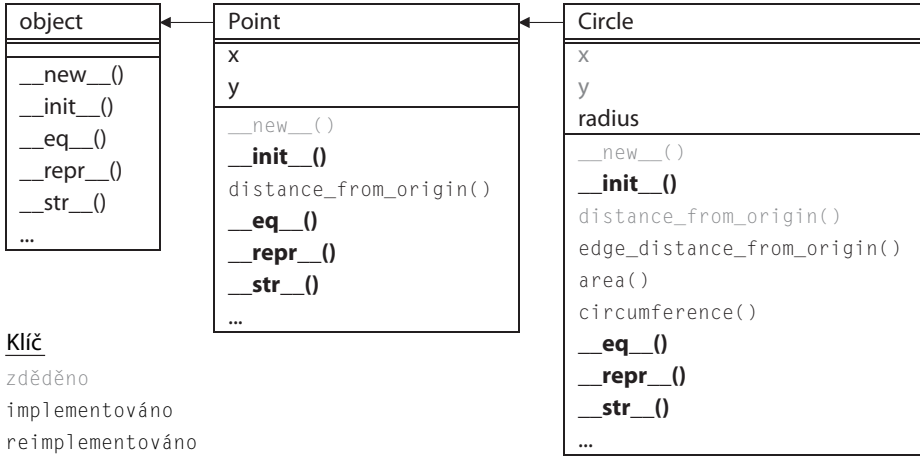
Dědičnost a polymorfismus

Třída `Circle` je pomocí dědičnosti postavena na třídě `Point`. K ní přidává jeden datový atribut (poloměr), tři nové metody a několik metod reimplementuje. Zde je kompletní definice třídy `Circle`:

```
class Circle(Point):

    def __init__(self, radius, x=0, y=0):
        super().__init__(x, y)
        self.radius = radius

    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin() - self.radius)

    def area(self):
        return math.pi * (self.radius ** 2)

    def circumference(self):
        return 2 * math.pi * self.radius

    def __eq__(self, other):
        return self.radius == other.radius and super().__eq__(other)

    def __repr__(self):
        return "Circle({0.radius!r}, {0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return repr(self)
```

Dědičnosti je dosaženo prostým uvedením třídy (nebo tříd), od kterých má být naše třída odvozena, na řádku s příkazem `class`.* Zde jsme dědili od třídy Point. Hierarchii dědičnosti pro třídu `Circle` zachycuje obrázek 6.3.

* Vícenásobná dědičnost, abstraktní bázové typy a další pokročilé objektově orientované technologie budeme probírat v lekci 8.



Obrázek 6.3: Hierarchie dědičnosti třídy Circle

Uvnitř metody `__init__()` voláme pomocí funkce `super()` metodu `__init__()` bázové třídy, čímž se vytvoří a inicializují atributy `self.x` a `self.y`. Uživatelé této třídy mohou zadat neplatný poloměr (např. -2), a proto si v následujícím oddílu ukážeme, jak tomuto problému předcházet vytvořením robustnějším atributů pomocí vlastností.

Metody `area()` a `circumference()` jsou jednoduché. Metoda `edge_distance_from_origin()` volá při svém výpočtu metodu `distance_from_origin()`. Třída `Circle` neposkytuje vlastní implementaci pro metodu `distance_from_origin()`, a proto se vyhledá a použije metoda definovaná ve třídě `Point`. Jiná situace nastává u reimplementované metody `__eq__()`. Tato metoda porovnává poloměr tohoto kruhu s poloměrem druhého kruhu, a jsou-li oba stejné, pak pomocí funkce `super()` explicitně zavolá metodu `__eq__()` bázové třídy. Pokud bychom nepoužili funkci `super()`, dostali bychom se do nekonečné rekurze, protože metoda `Circle.__eq__()` by stále volala sama sebe. Všimněte si také, že funkci `super()` nemusíme předávat argument `self`, který za nás automaticky předá Python.

Zde je několik příkladů:

```
p = Shape.Point(28, 45)
c = Shape.Circle(5, 28, 45)
p.distance_from_origin()     # vrátí: 53.0
c.distance_from_origin()     # vrátí: 53.0
```

Metodu `distance_from_origin()` voláme na objektu typu `Point` i `Circle`, poněvadž objekty typu `Circle` mohou zastupovat objekty typu `Point`.

Polymorfizmus znamená, že libovolný objekt zadané třídy lze použít tak, jako by šlo o objekt kterékoli z bázových tříd jeho třídy. Proto při vytváření podtřídy musíme implementovat pouze námi doplněné metody a reimplementovat pouze ty stávající metody, které chceme nahradit. A při reimplementování metod můžeme v případě potřeby použít její implementaci v bázové třídě, což provedeme použitím funkce `super()` uvnitř reimplementace.

V případě třídy `Circle` jsme implementovali dodatečné metody `area()` a `circumference()` a reimplementovali jsme metody, které jsme potřebovali změnit. Reimplementace metod `__repr__()` a `__str__()` je nezbytná, protože bez ní by se použily metody bázové třídy a vrácené řetězce by neobsahovaly údaje o objektu typu `Circle`, ale o objektu typu `Point`. Nezbytná je též reimplementace metod `__init__()` a `__eq__()`, protože musíme počítat se skutečností, že objekty typu `Circle` mají další atribut a v obou případech využíváme implementace metod v bázové třídě pro minimalizaci práce, kterou musíme provést.

Mělké a hloubkové kopírování ➤ 146

Třídy `Point` a `Circle` jsou z hlediska našich potřeb hotové. Mohli bychom doplnit další metody, například ostatní speciální metody pro porovnávání, které by nám umožnily objekty typu `Point` a `Cricle` seřazovat. Další věcí, která by mohla být potřebná a pro kterou nemáme žádnou metodu, je kopírování objektů typu `Point` či `Circle`. Většina tříd Pythonu neposkytuje metodu `copy()` (výjimkami jsou `dict.copy()` a `set.copy()`). Pokud bychom chtěli objekt typu `Point` nebo `Circle` zkopírovat, můžeme k tomu snadno použít modul `copy` a jeho funkci `copy.copy()`. (Pro objekty typu `Point` a `Circle` není nutné použít funkci `copy.deepcopy()`, protože obsahují pouze neměnitelné proměnné instance.)

Řízení přístupu k atributům pomocí vlastností

V předchozím oddílu obsahovala třída `Point` metodu `distance_from_origin()` a třída `Circle` metody `area()`, `circumference()` a `edge_distance_from_origin()`. Všechny tyto metody vracejí jedinou hodnotu typu `float`, takže z pohledu uživatele těchto tříd může klidně jít o atributy, které jsou samozřejmě určené pouze pro čtení. V souboru `ShapeAlt.py` je k dispozici alternativní implementace třídy `Point` a `Circle`, přičemž všechny zde uvedené metody jsou nabízeny jako vlastnosti. Díky tomu můžeme psát kód tímto způsobem:

```
circle = Shape.Circle(5, 28, 45)  # předpokládáme: import ShapeAlt as Shape
circle.radius                     # vrátí: 5
circle.edge_distance_from_origin  # vrátí: 48.0
Zde jsou implementace metod pro čtení vlastností area a edge_distance_from_origin
třídy ShapeAlt.Circle:
    @property
    def area(self):
        return math.pi * (self.radius ** 2)

    @property
    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin - self.radius)
```

Pokud definujeme pouze metody pro čtení vlastností, jak jsme to provedli zde, budou tyto vlastnosti pouze pro čtení. Kód pro vlastnost `area` je stejný jako pro dříve definovanou metodu `area()`. Kód vlastnosti `edge_distance_from_origin` se od předchozího malinko liší, protože nyní místo volání metody `distance_from_origin()` přistupuje k vlastnosti `distance_from_origin` z bázové třídy. Nejmarkantnější odlišností v obou definicích je *dekorátor* `property`. Dekorátor je funkce, která přijímá jako svůj argument funkci či metodu a vrací „dekorovanou" verzi, což je jistá verze této funkce či metody, která je nějakým způsobem pozměněná. Dekorátor se označuje symbolem zavináče (@) umístěným před jeho název. Prozatím považujte dekorátor za součást syntaxe (v lekci 8 si ukážeme, jak vytvářet vlastní dekorátory).

Dekorátorová funkce `property()` je vestavěná a přijímá až čtyři argumenty: funkci pro čtení vlastnosti, funkci pro zápis vlastnosti, funkci pro vymazání a dokumentační řetězec. Efekt použití dekorátoru `@property` je stejný jako zavolání funkce `property()` s jedním argumentem ve formě funkce pro čtení vlastnosti. Vlastnost `area` bychom tedy mohli vytvořit také takto:

```
def area(self):
    return math.pi * (self.radius ** 2)
area = property(area)
```

Tato syntaxe se ale používá jen zřídka, protože dekorátor nabízí kratší a čistší zápis.

V předchozím oddílu jsme si řekli, že v souvislosti s atributem `radius` třídy `Circle` se neprovádí žádná validace. Validaci můžeme poskytnout převedením atributu `radius` na vlastnost. Metodu `Circle.__init__()` nemusíme nijak měnit a veškerý kód přistupující k atributu `Circle.radius` bude i nadále fungovat. Jediné, co se změní, je to, že od nynějška bude atribut `radius` při každém nastavení validován.

Programátoři v jazyku Python běžně používají vlastnosti místo explicitních metod pro čtení a zápis vlastnosti (např. `getRadius()` a `setRadius()`), které se tak často používají v ostatních objektově orientovaných jazycích. To je dáno tím, že lze tak snadno změnit datový atribut na vlastnost, aniž by to nějak ovlivnilo použití dané třídy.

Pro převod atributu na čitelnou/zapisovatelnou vlastnost musíme vytvořit soukromý atribut, v němž budou uchována skutečná data, a doplnit metody pro čtení a zápis vlastnosti. Zde je celý kód pro metody pro čtení a zápis vlastnosti `radius` včetně dokumentačního řetězce:

```
    @property
    def radius(self):
        """Poloměr kruhu

        >>> circle = Circle(-2)
        Traceback (most recent call last):
        ...
        AssertionError: poloměř musí být nenulový a nezáporný
        >>> circle = Circle(4)
        >>> circle.radius = -1
        Traceback (most recent call last):
        ...
        AssertionError: poloměř musí být nenulový a nezáporný
        >>> circle.radius = 6
        """
        return self.__radius

    @radius.setter
    def radius(self, radius):
        assert radius > 0, "poloměř musí být nenulový a nezáporný"
        self.__radius = radius
```

Pro zajištění, aby byl poloměr nenulový a nezáporný, používáme příkaz `assert` a hodnotu poloměru ukládáme do soukromého atributu `self.__radius`. Všimněte si, že metody pro čtení a zápis (a také pro vymazání, pokud je třeba) mají stejný název. Jsou to právě dekorátory, které mezi nimi rozlišují a které je vhodným způsobem přejmenují, aby nedošlo ke konfliktu názvů.

Dekorátor pro nastavení vlastnosti může na první pohled vypadat zvláštně. Každá vytvořená vlastnost má atributy `getter`, `setter` a `deleter`, takže jakmile pomocí dekorátoru `@property` vytvoříme vlastnost `radius`, máme k dispozici atributy `radius.getter`, `radius.setter` a `radius.deleter`. Atribut `radius.getter` nastavuje na metodu pro čtení vlastnosti dekorátor `@property`. Ostatní dva atributy nastavuje Python tak, aby nedělaly nic (takže atribut nelze zapisovat ani vymazat), dokud je nepoužijeme jako dekorátory, přičemž se v podstatě nahradí metodou, kterou dekorují.

Inicializační metoda třídy `Circle` (`Circle.__init__()`) obsahuje příkaz `self.radius = radius`, který zavolá metodu pro nastavení vlastnosti `radius`, takže pokud je při vytváření objektu typu `Circle` zadán neplatný poloměr, vyvolá se výjimka `AssertionError`. A podobně se při pokusu o nastavení stávajícího poloměru objektu typu `Circle` na neplatnou hodnotu zavolá metoda pro nastavení vlastnosti, která vyvolá výjimku. Dokumentační řetězec obsahuje dokumentační testy k otestování, že se v těchto případech správně vyvolá výjimka. (Testování budeme podrobně probírat v lekci 9.)

Oba typy `Point` a `Circle` jsou vlastními datovými typy, které nabízejí dostatečnou funkčnost pro praktické požití. Většina datových typů, které budeme vytvářet, bude v tomto ohledu podobná, avšak někdy je nezbytné vytvořit vlastní datový typ, které je kompletní v každém směru. Příklady takového datového typu si ukážeme v následujícím oddílu.

Tvorba kompletních, plně integrovaných datových typů

Při vytváření kompletního datového typu máme k dispozici dvě možnosti. První možnost spočívá ve vytvoření datového typu úplně od začátku. I když bude takový datový typ odvozen od třídy `object` (jako ostatně všechny třídy jazyka Python), musíme poskytnout všechny datové atributy a metody požadované tímto datovým typem (kromě metody `__new__()`). Další možností je odvození ze stávajícího datového typu, který je podobný tomu, jejž chceme vytvořit. V tomto případě nám obvykle stačí reimplementovat metody, které se mají chovat odlišně, a „odimplementovat“ ty metody, které vůbec nechceme.

V následujícím textu budeme implementovat datový typ `FuzzyBool` úplně od začátku a pak budeme implementovat stejný datový typ s využitím dědičnosti, čímž snížíme množství práce, kterou je třeba vykonat. Vestavěný typ `bool` je dvouhodnotový (`True` a `False`), avšak v některých oblastech AI (Artificial Intelligence – umělá inteligence) se používají typy `fuzzy`, které mají hodnoty odpovídající „pravdě“ a „nepravdě“, a také hodnoty představující přechod mezi těmito krajními polohami. V naší implementaci budeme používat hodnoty s pohyblivou řádovou čárkou, kde `0.0` označuje hodnotu `False` a `1.0` označuje hodnotu `True`. V tomto systému znamená `0.5` padesátiprocentní pravdu a `0.25` dvacetipětiprocentní pravdu. Zde je několik příkladů použití (fungují stejně v kterékoli z implementací):

```
a = FuzzyBool.FuzzyBool(.875)
b = FuzzyBool.FuzzyBool(.25)
a >= b                              # vrátí: True
bool(a), bool(b)                    # vrátí: (True, False)
```

```
~a                                  # vrátí: FuzzyBool(0.125)
a & b                               # vrátí: FuzzyBool(0.25)
b |= FuzzyBool.FuzzyBool(.5)        # b je nyní: FuzzyBool(0.5)
"a={0:.1%} b={1:.0%}".format(a, b)  # vrátí: 'a=87.5% b=50%'
```

Chceme, aby typ `FuzzyBool` podporoval celou skupinu porovnávacích operátorů (`<`, `<=`, `==`, `!=`, `>=`, `>`) a tři základní logické operace: negaci (`~`), konjunkci (`&`) a disjunkci (`|`). Kromě logických operací chceme poskytnout další logické metody `conjunction()` a `disjunction()`, které přijímají libovolné množství objektů typu `FuzzyBool` a vracejí příslušný výsledek jako objekt typu `FuzzyBool`. A k tomu, aby byl náš typ kompletní, musíme implementovat převody na typy `bool`, `int`, `float` a `str` a poskytnout reprezentační formu vhodnou pro funkci `eval()`. Posledními požadavky je to, aby typ `FuzzyBool` podporoval specifikaci formátu pomocí metody `str.format()`, aby objekty typu `FuzzyBool` bylo možné použít jako klíče slovníku nebo jako členy množin a aby byly objekty typu `FuzzyBool` neměnitelné, ovšem s možností použít operátory rozšířeného přiřazení, které zajišťují komfortní použití.

Tabulka 6.1 (strana 237) obsahuje speciální metody pro porovnávání, tabulka 6.2 uvádí základní speciální metody a tabulka 6.3 (strana 245) nabízí číselné speciální metody, jež zahrnují bitové operátory (`~`, `&` a `|`), které budou objekty typu `FuzzyBool` používat pro své logické operace, a také aritmetické operátory, jako je `+` a `-`, které typ `FuzzyBool` nebude implementovat, protože pro něj nejsou vhodné.

Tabulka 6.2: Základní speciální metody

Speciální metoda	Použití	Popis
`__bool__(self)`	bool(x)	Je-li uvedena, pak vrací pravdivostní hodnotu pro objekt x (užitečné pro `if x: ...`).
`__format__(self, formát)`	„{0}“.format(x)	Poskytuje podporu metody `str.format()` pro vlastní třídy.
`__hash__(self)`	hash(x)	Je-li uvedena, pak lze objekt x použít jako klíč slovníku nebo uchovávat v množině.
`__init__(self, args)`	x = X(args)	Volá se při inicializaci objektu.
`__new__(cls, args)`	x = X(args)	Volá se při vytváření objektu.
`__repr__(self)`	repr(x)	Vrací řetězcovou reprezentaci objektu x, takže platí `eval(repr(x)) == x`.
`__repr__(self)`	ascii(x)	Vrací řetězcovou reprezentaci objektu x s použitím pouze znaků ze znakové sady ASCII.
`__str__(self)`	str(x)	Vrací řetězcovou reprezentaci objektu x srozumitelnou pro člověka.

Reimplementace metody `__new__()` ➤ 251
`ascii()` ➤ 73
`str.format()` ➤ 87

Speciální metoda __del__()

Speciální metoda `__del__(self)` se volá v okamžiku zničení objektu, tedy alespoň teoreticky. V praxi se totiž nemusí vůbec zavolat, a to ani při ukončení programu. Ba co víc, když napíšeme příkaz `del x`, tak se nestane nic jiného, než že se vymaže odkaz na objekt `x` a počet odkazů na objekty, které ukazují na objekt, na který ukazoval odkaz na objekt `x`, se o jedničku sníží. Až v okamžiku, kdy tento počet klesne na 0, se může zavolat metoda `__del__()`, avšak Python nijak nezaručuje, že se vůbec kdy zavolá. Z tohoto důvodu se metoda `__del__()` jen zřídka reimplementuje (v žádném z příkladů v této knize ji nereimplementujeme) a neměla by se používat k uvolnění prostředků, takže není vhodná pro uzavírání souborů nebo odpojování síťových či databázových spojení.

Python nabízí dva samostatné mechanizmy pro zajištění, že se prostředky náležitě uvolní. První spočívá v použití bloku `try ... finally`, jak jsme viděli již dříve a opět uvidíme v lekci 7, a druhý v použití kontextového objektu ve spojení s příkazem `with` – toto téma budeme probírat v lekci 8.

Tabulka 6.3: Číselné a bitové speciální metody

Speciální metoda	Použití	Speciální metoda	Použití
`__abs__(self)`	`abs(x)`	`__complex__(self)`	`complex(x)`
`__float__(self)`	`float(x)`	`__int__(self)`	`int(x)`
`__index__(self)`	`bin(x) oct(x) hex(x)`	`__round__(self, číslice)`	`round(x, číslice)`
`__pos__(self)`	`+x`	`__neg__(self)`	`-x`
`__add__(self, other)`	`x + y`	`__sub__(self, other)`	`x - y`
`__iadd__(self, other)`	`x += y`	`__isub__(self, other)`	`x -= y`
`__radd__(self, other)`	`y + x`	`__rsub__(self, other)`	`y - x`
`__mul__(self, other)`	`x * y`	`__mod__(self, other)`	`x % y`
`__imul__(self, other)`	`x *= y`	`__imod__(self, other)`	`x %= y`
`__rmul__(self, other)`	`y * x`	`__rmod__(self, other)`	`y % x`
`__floordiv__(self, other)`	`x // y`	`__truediv__(self, other)`	`x / y`
`__ifloordiv__(self, other)`	`x //= y`	`__itruediv__(self, other)`	`x /= y`
`__rfloordiv__(self, other)`	`y // x`	`__rtruediv__(self, other)`	`y / x`

Speciální metoda	Použití	Speciální metoda	Použití
`__divmod__(self, other)`	`divmod(x, y)`	`__rdivmod__(self, other)`	`divmod(y, x)`
`__pow__(self, other)`	`x ** y`	`__and__(self, other)`	`x & y`
`__ipow__(self, other)`	`x **= y`	`__iand__(self, other)`	`x &= y`
`__rpow__(self, other)`	`y ** x`	`__rand__(self, other)`	`y & x`
`__xor__(self, other)`	`x ^ y`	`__or__(self, other)`	`x | y`
`__ixor__(self, other)`	`x ^= y`	`__ior__(self, other)`	`x |= y`
`__rxor__(self, other)`	`y ^ x`	`__ror__(self, other)`	`y | x`
`__lshift__(self, other)`	`x << y`	`__rshift__(self, other)`	`x >> y`
`__ilshift__(self, other)`	`x <<= y`	`__irshift__(self, other)`	`x >>= y`
`__rlshift__(self, other)`	`y << x`	`__rrshift__(self, other)`	`y >> x`
		`__invert__(self)`	`~x`

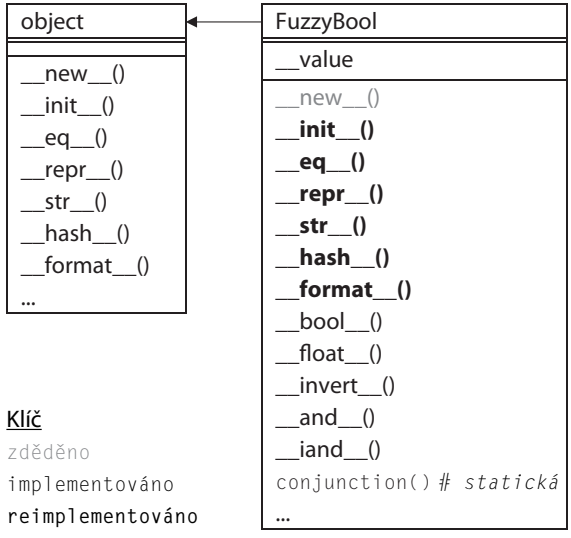
Tvorba datových typů úplně od začátku

Vytvoření typu `FuzzyBool` úplně od začátku znamená, že musíme poskytnout atribut pro uchování hodnoty objektu typu `FuzzyBool` a také všechny potřebné metody. Zde je řádek s příkazem `class` a inicializační metody ze souboru `FuzzyBool.py`:

```
class FuzzyBool:

    def __init__(self, value=0.0):
        self.__value = value if 0.0 <= value <= 1.0 else 0.0
```

Vlastnost `ShapeAlt.Circle.radius` ➤ 242

Atribut `value` je soukromý, protože chceme, aby se typ `FuzzyBool` choval jako neměnitelný typ, takže povolení přístupu k tomuto atributu by nebylo správné. Dále při zadání hodnoty mimo rozsah použijeme hodnotu `0.0` (nepravda). V dříve definované třídě `ShapeAlt.Circle` jsme použili striktnější zásadu, takže jsme při použití neplatného poloměru při tvorbě nového objektu typu `Circle` vyvolali výjimku. Strom dědičnosti pro třídu `FuzzyBool` je na obrázku 6.4.



Obrázek 6.4: Hierarchie dědičnosti třídy FuzzyBool

Nejjednodušším logickým operátorem je logická negace (`NOT`), pro niž jsme si vypůjčili operátor bitové inverze (~):

```
def __invert__(self):
    return FuzzyBool(1.0 - self.__value)
```

Speciální metoda `__and__()` poskytuje bitový operátor (`&`) pro logickou konjunkci (`AND`), přičemž místní verzi (`&=`) poskytuje metoda `__iand__()`:

```
def __and__(self, other):
    return FuzzyBool(min(self.__value, other.__value))

def __iand__(self, other):
    self.__value = min(self.__value, other.__value)
    return self
```

Bitový operátor `AND` vrací nový objekt typu `FuzzyBool` založený na aktuálním a druhém objektu typu `FuzzyBool`, kdežto rozšířené (místní) přiřazení aktualizuje soukromý atribut `value`. Toto není přesně vzato neměnitelné chování, odpovídá ale chování některých neměnitelných typů Pythonu, jako je třeba typ `int`, kde například použití operátoru += vypadá, jako by se levý operand změnil, ale ve skutečnosti je opětovně svázán tak, aby ukazoval na nový objekt typu `int` uchovávající výsledek součtu. V tomto případě není opětovné svazování nutné, protože ve skutečnosti měníme samotný objekt typu `FuzzyBool`. A dále kvůli podpoře řetězení operací vracíme objekt argument `self`.

Mohli bychom implementovat také metodu `__rand__()`. Tato metoda se zavolá, když jsou argumenty `self` a `other` odlišného typu a metoda `__and__()` není pro tuto dvojici typů implementována. Pro třídu `FuzzyBool` to ale není potřeba. Většina speciálních metod pro binární operátory má obě verze „i“ (in-place – místní) a „r“ (reflektuj, což znamená, prohoď operandy).

Implementaci metody `__or__()`, která poskytuje bitový operátor |, a metody `__ior__()`, která poskytuje místní operátor |=, jsme si neukázali, protože obě metody jsou stejné jako ekvivalentní metody `AND`, tedy až na to, že místo minimální z hodnot argumentů `self` a `other` bereme hodnotu maximální.

```
def __repr__(self):
    return ("{0}({1})".format(self.__class__.__name__,
                              self.__value))
```

Vytvořili jsme reprezentační formu vhodnou pro funkci `eval()`. Máme-li například `f = FuzzyBool.FuzzyBool(.75)`, pak volání funkce `repr(f)` vytvoří řetězec `'FuzzyBool(0.75)'`.

Všechny objekty mají speciální atributy dodávané automaticky Pythonem. Jeden z nich se nazývá `__class__` a ukazuje na objekt třídy. Všechny třídy mají soukromý atribut `__name__`, který je opět poskytován zcela automaticky. Pomocí těchto atributů jsme se dostali k názvu třídy, který jsme použili pro reprezentační formu. To znamená, že pokud by byla třída `FuzzyBool` odvozena pouze kvůli přidání nových metod, fungovala by zděděná metoda `__repr__()` správně, aniž by bylo nutné ji reimplementovat, protože vybere název třídy naší podtřídy.

```
    def __str__(self):
        return str(self.__value)
```

Pro řetězcovou formu jsme vrátili jen hodnotu s pohyblivou řádovou čárkou naformátovanou jako řetězec. Použití funkce `super()` pro zamezení nekonečné rekurze zde není nutné, protože funkci `str()` voláme na atributu `self.__value`, ne na samotné instanci.

```
    def __bool__(self):
        return self.__value > 0.5

    def __int__(self):
        return round(self.__value)

    def __float__(self):
        return self.__value
```

Speciální metoda `__bool__()` převede instanci na logickou hodnotu, takže musí vždy vracet buď `True`, nebo `False`. Speciální metoda `__int__()` poskytuje převod na celé číslo. Použili jsme vestavěnou funkci `round()`, protože funkce `int()` zadané číslo jen ořeže (takže by vrátila hodnotu `0` pro jakoukoli hodnotu typu `FuzzyBool` kromě hodnoty `1.0`). Převod na číslo s pohyblivou řádovou čárkou je jednoduchý, protože naše hodnota je již číslem s pohyblivou řádovou čárkou.

```
    def __lt__(self, other):
        return self.__value < other.__value

    def __eq__(self, other):
        return self.__value == other.__value
```

Pro poskytnutí kompletní skupiny porovnávacích operátorů (<, <=, ==, !=, >=, >) je nezbytné implementovat alespoň tři z nich (<, <= a ==), poněvadž Python dokáže odvodit > z <, != z == a >= z <=. Zde jsme si ukázali pouze dvě reprezentační metody, protože všechny jsou velice podobné.*

Kompletní porovnávání ➤ 366

```
def __hash__(self):
    return hash(id(self))
```

Instance vlastních tříd podporují ve výchozím stavu operátor == (který vrací vždy hodnotu `False`) a jsou hashovatelné (takže mohou být klíči slovníků a lze je přidávat do množin). Pokud ale reimplementujeme speciální metodu `__eq__()` pro náležité testování rovnosti, přestanou být instance hashovatelné. To lze napravit poskytnutím speciální metody `__hash__()`, jak jsme to provedli zde.

Python nabízí hashovací funkce pro řetězce, čísla, zmrazené množiny a další třídy. Zde jsme jen použili vestavěnou funkci `hash()` (která dokáže pracovat na libovolném typu, který má speciální metodu `__hash__()`) a předali jsme jí jedinečné ID objektu, z něhož má hash vypočítat. (Soukromý atribut `self.__value` použít nemůžeme, protože ten se může v důsledku rozšířeného přiřazení změnit, kdežto hashová hodnota objektu se nesmí změnit nikdy.)

Vestavěná funkce `id()` vrací jedinečné číslo pro objekt zadaný jako argument. Toto číslo je obvykle adresou objektu v paměti, my ale nemůžeme předpokládat nic jiného, než že žádné dva objekty nemají totéž ID. Operátor `is` používá v pozadí funkci `id()` pro zjištění, zda dva odkazy na objekty ukazují na tentýž objekt.

```
def __format__(self, format_spec):
    return format(self.__value, format_spec)
```

Vestavěná funkce `format()` je skutečně nezbytná jen v definicích třídy. Přijímá jeden objekt a volitelnou specifikaci formátu a vrací řetězec s vhodně naformátovaným objektem.

Když ve formátovacím řetězci použijeme nějaký objekt, zavolá se jeho metoda `__format__()` s tímto objektem a specifikací formátu jako argumenty. Jak jsme viděli výše, tato metoda vrátí vhodně naformátovanou instanci.

Příklady použití typu `FuzzyBool` ➤ 243

Všechny vestavěné třídy mají vhodné metody `__format__()`. Zde využíváme metodu `float.__format__()`, které předáváme hodnotu s pohyblivou řádovou čárkou a zadaný formátovací řetězec. Naprosto stejného výsledku bychom dosáhli také tímto způsobem:

```
def __format__(self, format_spec):
    return self.__value.__format__(format_spec)
```

Použití funkce `format()` vyžaduje méně psaní a je čitelnější. Nic nás ale nenutí používat funkci `format()`, takže můžeme vyvinout vlastní jazyk pro specifikaci formátu a interpretovat jej uvnitř metody `__format__()`. Podstatné je, abychom vždy vrátili nějaký řetězec.

```
@staticmethod
def conjunction(*fuzzies):
    return FuzzyBool(min([float(x) for x in fuzzies]))
```

* Ve skutečnosti jsme implementovali pouze výše uvedené metody `__lt__()` a `__eq__()`, protože ostatní porovnávací metody byly vygenerovány automaticky. Jak k tomu dochází, si ukážeme v lekci 8.

Vestavěná funkce `staticmethod()` je navržena tak, aby se používala jako dekorátor, který jsme využili při definici výše uvedené metody. Statické metody jsou obyčejné metody, které nepřijímají Pythonem speciálně předávaný první argument `self`.

Operátor `&` lze řetězit, takže máme-li objekty typu `FuzzyBool` s názvy `f`, `g` a `h`, pak můžeme získat jejich konjunkci zapsáním `f & g & h`. To funguje skvěle, je-li těchto objektů malý počet, pokud jich ale máme desítky nebo více, začne být tento způsob poněkud neefektivní, protože každý operátor `&` představují volání jisté funkce. Díky výše definované metodě můžeme docílit stejného výsledku pomocí jediného volání `FuzzyBool.FuzzyBool.conjunction(f, g, h)`. Tento zápis můžeme ještě zkrátit pomocí instance třídy `FuzzyBool`, statické metody však nedostávají argument `self`. Pokud tedy takovou metodu zavoláme a chceme zpracovat i onu instanci, pak ji musíme znovu předat jako argument, například `f.conjunction(f, g, h)`.

Odpovídající metodu `disjunction()` jsme si neukázali, protože se liší pouze názvem a tím, že místo funkce `min()` používá funkci `max()`.

Někteří programátoři Pythonu považují statické metody za něco, co do jazyka Python nepatří, a používají je pouze při převodu kódu z jiného jazyka (např. z C++ nebo Javy) nebo pokud mají metodu, která nepoužívá parametr `self`. V Pythonu je lepší místo statické metody vytvořit funkci modulu, jak uvidíme v následujícím textu, nebo metodu třídy, jak uvidíme v poslední části.

Podobně vede vytvoření proměnné uvnitř definice třídy, ale mimo jakoukoli metodu k vytvoření statické proměnné (proměnné třídy). Pro konstanty je obvykle vhodnější použít soukromé globální proměnné modulu, avšak proměnné třídy mohou být užitečné pro sdílení dat mezi všemi instancemi dané třídy.

Nyní jsme dokončili implementaci třídy `FuzzyBool`, kterou jsme vytvořili úplně od začátku. Museli jsme reimplementovat 15 metod (17, pokud bychom provedli minimum pro všechny čtyři porovnávací operátory) a implementovali jsme dvě statické metody. V následujícím textu si ukážeme alternativní implementaci založenou tentokrát na odvození od typu `float`. Obsahuje reimplementaci pouze osmi metod a implementaci dvou funkcí modulu – a „odimplementování" 32 metod.

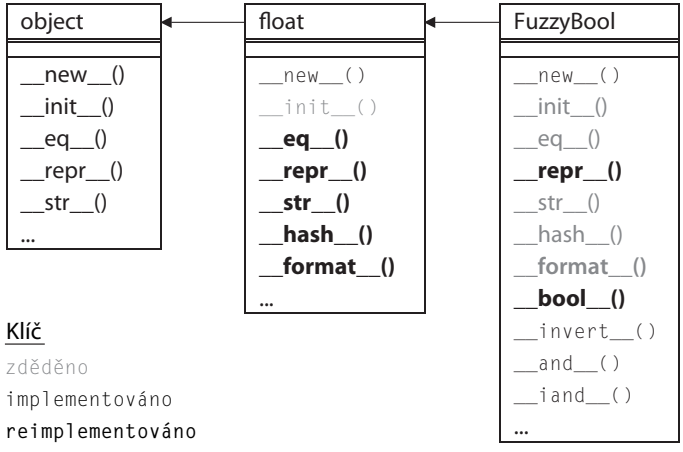
Ve většině objektově orientovaných jazyků se dědičnost používá k tvorbě nových tříd, které mají všechny metody a atributy tříd, od kterých dědí, a k tomu další metody a atributy, které má mít nová třída. To vše jazyk Python plně podporuje, díky čemuž můžeme přidat nové metody nebo reimplementovat zděděné metody k úpravě jejich chování. Ovšem kromě toho nám Python umožňuje efektivně metody odimplementovat, což znamená, že se nová třída bude chovat tak, jako by některé ze zděděných metod vůbec neměla. Tento postup se možná nebude zamlouvat objektově orientovaným puristům, protože porušuje polymorfizmus, avšak přinejmenším v Pythonu může být čas od času užitečný.

Tvorba datových typů z jiných datových typů

Nyní se podíváme na implementaci třídy `FuzzyBool` umístěnou v souboru `FuzzyBoolAlt.py`. Okamžitě viditelným rozdílem oproti předchozí verzi je to, že původně statické metody `conjunction()` a `disjunction()` zde mají podobu funkcí modulu. Například:

```
def conjunction(*fuzzies):
    return FuzzyBool(min(fuzzies))
```

Kód pro tuto funkci je mnohem jednodušší než předtím, protože objekty typu `FuzzyBoolAlt.FuzzyBool` jsou odvozeny od třídy `float`, a proto je lze použít přímo místo objektů typu `float` bez nutnosti jakéhokoliv převodu. (Strom dědičnost je k dispozici na obrázku 6.5.) Přístup k funkci je také čistší než dříve. Místo nutnosti zadávat modul a třídu (nebo použít instanci) stačí po provedení příkazu `import FuzzyBoolAlt` jen napsat `FuzzyBoolAlt.conjunction()`.



Obrázek 6.5: Hierarchie dědičnost alternativní třídy FuzzyBool

Zde je řádek s definicí třídy `FuzzyBool` a její metoda `__new__()`:

```
class FuzzyBool(float):

    def __new__(cls, value=0.0):
        return super().__new__(cls, value if 0.0 <= value <= 1.0 else 0.0)
```

Při vytváření nové třídy je tato obvykle měnitelná a pro vytvoření holého neinicializovaného objektu se opírá o metodu `object.__new__()`. Ovšem v případě neměnitelných tříd potřebujeme provést vytvoření a inicializaci v jediném kroku, protože jakmile je neměnitelný objekt vytvořen, nelze jej už nijak změnit.

Metoda `__new__()` se volá před vytvoření každého objektu (tvorba objektu je úkolem právě metody `__new__()`), a proto nepřijímá objekt `self`, který dosud neexistuje. Ve skutečnosti se totiž jedná o *metodu třídy*. Metody třídy se podobají běžným metodám, od nichž se liší tím, že se nevolají na instanci, ale na objektu, přičemž Python jim dodává jako první argument třídu, na které se volají. Jméno proměnné `cls` pro třídu je jen otázkou konvence, tedy stejně jako je `self` tradičním názvem pro samotný objekt.

Když tedy napíšeme `f = FuzzyBool(0.7)`, tak Python v pozadí zavolá `FuzzyBool.__new__(FuzzyBool, 0.7)` pro vytvoření nového objektu, řekněme *`fuzzy`*, poté zavolá *`fuzzy`*`.__init__()` k provedení jakékoli další inicializace a nakonec vrátí odkaz na objekt ukazující na objekt *`fuzzy`* (proměnná `f` je nastavena na tento odkaz na objekt). Většina práce metody `__new__()` je předána implementaci v bázové třídě (`object.__new__()`). Nám už jen stačí zajistit, aby byla zadaná hodnota v platném rozsahu.

Metody třídy se ustanovují pomocí vestavěné funkce `classmethod()` použití jako dekorátor. Standardně ale není nutné před metodou `__new__()` uvádět dekorátor `@classmethod`, protože Python již

ví, že tato metoda je vždy metodou třídy. Tento dekorátor musíme použít pouze tehdy, když chceme vytvářet jiné metody třídy, jak uvidíme v poslední části této lekce.

Metodu třídy jsme již viděli, a proto si můžeme vyjasnit různé druhy metod podporované jazykem Python. Metodám třídy přidává jejich první argument Python a jedná se o třídu dané metody. U běžných metod přidává první argument Python a jedná se instanci, na níž byla daná metoda zavolána. Statickým metodám se žádný první argument nepřidává. A všechny druhy metod pak dostávají libovolné argumenty, které jim předáme (jako jejich druhé a následné argumenty v případě metod tříd a běžných metod a jako jejich první a následné argumenty v případě statických metod).

```
def __invert__(self):
    return FuzzyBool(1.0 - float(self))
```

Tato metoda se používá na podporu bitového operátoru `NOT` (~), tedy stejně jako v předchozí verzi třídy. Všimněte si, že místo přístupu k soukromému atributu, který uchovává hodnotu objektu typu `FuzzyBool`, používáme přímo argument `self`. To můžeme provést díky odvození od typu `float`, což znamená, že objekt typu `FuzzylBool` můžeme použít všude tam, kde je očekáván objekt typu `float` – samozřejmě jen za předpokladu, že se nepoužije žádná z „odimplementovaných" metod typu `FuzzyBool`.

```
def __and__(self, other):
    return FuzzyBool(min(self, other))

def __iand__(self, other):
    return FuzzyBool(min(self, other))
```

Logika těchto metod je stejná jako u předchozí verze třídy (i když kód je malinko odlišný) a stejně jako v případě metody `__invert__()` můžeme použít argumenty `self` a `other` přímo, jako by šlo o objekty typu `float`. Verze operátoru `OR` jsme vynechali, protože se liší jen názvem (`__or__()` a `__ior__()`) a tím, že místo funkce `min()` používají funkci `max()`.

```
def __repr__(self):
    return ("{0}({1})".format(self.__class__.__name__,
                              super().__repr__()))
```

Metodu `__repr__()` musíme reimplementovat, protože verze bázové třídy `float.__repr__()` vrací řetězec obsahující pouze číslo, kdežto my potřebujeme název třídy, aby byla tato reprezentace vhodná pro funkci `eval()`. U druhého argumentu metody `str.format()` nemůžeme jen předat `self`, poněvadž to by vedlo k nekonečné rekurzi volání metod `__repr__()`. Proto raději zavoláme implementaci v bázové třídě.

Metodu `__str__()` reimplementovat nemusíme, protože verze bázové třídy `float.__str__()` je dostatečná a použije se v případě nepřítomnosti reimplementace `FuzzyBool.__str__()`.

```
def __bool__(self):
    return self > 0.5

def __int__(self):
    return round(self)
```

Když se v kontextu logického výrazu použije objekt typu `float`, pak se vyhodnotí na `False`, je-li jeho hodnota `0.0`, nebo se v opačném případě vyhodnotí na `True`. To není vhodné chování pro objekty typu `FuzzyBools`, takže jsme museli tuto metodu reimplementovat. Podobně by použití funkce `int(self)` vedlo k prostému oříznutí, takže by se vše kromě `1.0` změnilo na `0`, a proto zde používáme funkci `round()` k vytvoření `0` pro hodnoty do `0.5` a `1` pro hodnoty do `1.0` včetně.

Metodu `__hash__()`, `__format__()` ani žádnou další metodu poskytující porovnávací operátory jsme také reimplementovat nemuseli, protože všechny tyto metody poskytované bázovou třídou `float` fungují pro objekty typu `FuzzyBools` správně.

Metody, které jsme reimplementovali, poskytují kompletní implementaci třídy `FuzzyBool` a vyžadovaly mnohem méně kódu než implementace předchozí verze. Nicméně třída `FuzzyBool` zdědila více než 30 metod, které pro objekty typu `FuzzyBool` nedávají smysl. Například žádný ze základních číselných operátorů ani z operátorů bitového posunu (+, -, *, /, <<, >> atd.) nelze smysluplně aplikovat na objekty typu `FuzzyBool`. Zde je způsob, kterým bychom mohli zahájit „odimplementování“:

```
    def __add__(self, other):
        raise NotImplementedError()
```

Pro úplné zamezení sčítání bychom museli stejný kód použít také pro metody `__iadd__()` a `__radd__()`. (Je třeba poznamenat, že `NotImplementedError` je standardní výjimka, která je odlišná od vestavěného objektu `NotImplemented`.) Alternativou k vyvolávání výjimky `NotImplementedError` je vyvolat výjimku `TypeError`, zvláště pak v situaci, kdy chceme těsněji napodobit chování vestavěných tříd Pythonu. Níže uvedeným způsobem můžeme zajistit, aby se metoda `FuzzyBool.__add__()` chovala podobně jako vestavěné třídy, které čelí neplatné operaci:

```
def __add__(self, other):
    raise TypeError("unsupported operand type(s) for +: "
                    "'{0}' and '{1}'".format(
                self.__class__.__name__, other.__class__.__name__))
```

Pro unární operace, které chceme odimplementovat způsobem, který napodobuje chování vestavěných typů, je kód malinko jednodušší:

```
def __neg__(self):
    raise TypeError("bad operand type for unary -: '{0}'".format(
                    self.__class__.__name__))
```

Pro porovnávací operátory máme mnohem jednodušší způsob. Například k odimplementování operátoru == stačí napsat:

```
def __eq__(self, other):
    return NotImplemented
```

Pokud metoda implementující porovnávací operátor (<, <=, ==, !=, >=, >) vrátí vestavěný objekt `NotImplemented` a dojde k pokusu o použití této metody, Python se nejprve pokusí obrátit porovnání prohozením operandů (pro případ, že objekt `other` má vhodnou porovnávací metodu, která v objektu `self` chyběla), a pokud ani toto nefunguje, vyvolá se výjimka `TypeError` se zprávou, která vysvětlu-

je, že operace není podporována pro operandy s použitým typem. Ovšem u všech ostatních metod, které nechceme, musíme vyvolat buď výjimku `NotImplementedError`, nebo výjimku `TypeError`, jak jsme provedli ve výše uvedených metodách `__add__()` a `__neg__()`.

Bylo by únavné provádět odimplementování každé metody, kterou nechceme, dosavadním způsobem, i když by to samozřejmě šlo a navíc by to bylo velice srozumitelné. Nyní si ukážeme na pokročilejší techniku pro odimplementování metod (používá se v modulu `FuzzyBoolAlt`), avšak pravděpodobně lepší je přeskočit do následující části (na stranu 255) a vrátit se sem až v okamžiku, kdy to bude vyžadovat praxe.

Zde je kód pro odimplementování dvou unárních operací, které nechceme:

```
for name, operator in (("__neg__", "-"),
                       ("__index__", "index()")):
    message = ("bad operand type for unary {0}: '{{self}}'"
               .format(operator))
    exec("def {0}(self): raise TypeError(\"{1}\".format("
         "self=self.__class__.__name__))".format(name, message))
```

Dynamické programování ➢ 338

Vestavěná funkce `exec()` dynamicky provádí kód ze zadaného objektu. V tomto případě jsme funkci předali řetězec, stejně jí ale můžeme předat i jiné druhy objektů. Kód se standardně provede v kontextu aktuálního oboru platnosti, kterým je v tomto případě definice třídy `FuzzyBool`, takže provedené příkazy `def` vytvoří metody třídy `FuzzyBool`, což potřebujeme. Tento kód se provede pouze jednou při importu modulu `FuzzyBoolAlt`. Zde je kód, který se vygeneruje pro první n-tici `("__neg__", "-")`:

```
def __neg__(self):
    raise TypeError("bad operand type for unary -: '{self}'"
                    .format(self=self.__class__.__name__))
```

Kód jsme napsali tak, aby výjimka a chybová zpráva odpovídala tomu, co používá Python pro své vlastní typy. Kód pro ošetření binárních metod a funkcí s více argumenty (např. `pow()`) funguje podobným způsobem, ovšem s odlišnou chybovou zprávou. Pro úplnost zde uvádíme námi použitý kód:

```
for name, operator in (("__xor__", "^"), ("__ixor__", "^="),
        ("__add__", "+"), ("__iadd__", "+="), ("__radd__", "+"),
        ("__sub__", "-"), ("__isub__", "-="), ("__rsub__", "-"),
        ("__mul__", "*"), ("__imul__", "*="), ("__rmul__", "*"),
        ("__pow__", "**"), ("__ipow__", "**="),
        ("__rpow__", "**"), ("__floordiv__", "//"),
        ("__ifloordiv__", "//="), ("__rfloordiv__", "//"),
        ("__truediv__", "/"), ("__itruediv__", "/="),
        ("__rtruediv__", "/"), ("__divmod__", "divmod()"),
        ("__rdivmod__", "divmod()"), ("__mod__", "%"),
        ("__imod__", "%="), ("__rmod__", "%"),
        ("__lshift__", "<<"), ("__ilshift__", "<<="),
        ("__rlshift__", "<<"), ("__rshift__", ">>"),
        ("__irshift__", ">>="), ("__rrshift__", ">>")):
```

```
        message = ("unsupported operand type(s) for {0}: "
                   "'{{self}}'{{join}} {{args}}".format(operator))
        exec("def {0}(self, *args):\n"
             "    types = [\"'\" + arg.__class__.__name__ + \"'\" "
             "for arg in args]\n"
             "    raise TypeError(\"{1}\".format("
             "self=self.__class__.__name__, "
             "join=(\" and\" if len(args) == 1 else \",\"),"
             "args=\", \".join(types)))".format(name, message))
```

Tento kód je trošku složitější než dříve, protože pro napodobení vestavěného chování musíme u binárních operací vypisovat zprávy, v nichž jsou dva typy, jako *type1* a *type2*, ale pro tři a více typů musíme uvádět *type1*, *type2*, *type3*. Zde je kód, který se vygeneruje pro první n-tici `("__xor__", "^")`:

```
def __xor__(self, *args):
    types = ["'" + arg.__class__.__name__ + "'" for arg in args]
    raise TypeError("unsupported operand type(s) for ^: "
                    "'{self}'{join} {args}".format(
                    self=self.__class__.__name__,
                    join=(" and" if len(args) == 1 else ","),
                    args=", ".join(types)))
```

Dva zde použité bloky cyklů `for ... in` lze jednoduše zkopírovat a vložit a poté můžeme v prvním přidat nebo odstranit unární operátory a metody a ve druhém přidat nebo odstranit binární operátory a metody nebo operátory a metody s více argumenty, čímž odimplementujeme metody, které nejsou vyžadovány.

Poslední kousek kódu máme na místě, takže pokud nyní máme dva objekty *f* a *g* typu `FuzzyBool` a pokusíme se je pomocí `f + g` sečíst, obdržíme výjimku `TypeError` se zprávou `"unsupported operand type(s) for +: 'FuzzyBool' and 'FuzzyBool'"`, což je přesně to chování, které chceme.

Tvorba tříd způsobem, který jsme použili pro první implementaci třídy `FuzzyBool`, je mnohem častější a pro většinu případů dostatečná. Pokud ovšem potřebujeme vytvořit neměnitelnou třídu, pak musíme reimplementovat metodu `object.__new__()` zděděnou od jednoho z neměnitelných typů jazyka Python, jako je `float`, `int`, `str` nebo `tuple`, a poté implementovat všechny ostatní metody, které potřebujeme. Nevýhoda tohoto postupu tkví v tom, že některé metody možná budeme muset odimplementovat, což ale ničí polymorfizmus. Pro většinu případů je tedy mnohem vhodnější sáhnout po agregaci, jak jsme učinili v první implementaci třídy `FuzzyBool`.

Vlastní třídy představující kolekce

V této části se podíváme na vlastní třídy, které jsou odpovědné za velké množství dat. První třída nese název Image a stará se o obrazová data. Jedná se o typického zástupce vlastních tříd uchovávajících data, protože neposkytuje jen přístup k datům v paměti, ale také metody pro jejich ukládání na disk a načítání z disku. Dále prostudujeme třídy `SortedList` a `SortedDict`, které jsou navržené pro vyplnění ojedinělé a překvapivé mezery ve standardní knihovně Pythonu pro datové typy představující skutečně seřazené kolekce.

Tvorba tříd agregujících kolekce

Jednoduchým způsobem reprezentace dvourozměrného barevného obrázku je dvourozměrné pole, kde každý prvek představuje barvu. Pro reprezentaci obrázku v rozlišení 100×100 tedy musíme uložit 10 000 barev. V případě třídy `Image` (v souboru `Image.py`) je náš postup potenciálně efektivnější. Třída `Image` uchovává jedinou barvu pozadí plus barvy těch bodů v obrázku, které se od barvy pozadí liší. K tomuto účelu používáme slovník jako jakýsi druh řídkého pole, kde je každý klíč souřadnicí (x, y) a odpovídající hodnota barvou v tomto bodě. Máme-li obrázek 100×100, v němž tvoří polovina bodů pozadí, stačí nám pro jeho uložení 5000 + 1 barev, což je znatelná úspora paměti.

Naložené objekty ➤ 285

Modul `Image.py` je napsán nyní již docela známým postupem. Začíná řádkem shebang, pak následují v komentářích informace o copyrightu, poté dokumentační řetězce modulu s dokumentačními testy a dále import, v tomto případě modulů `os` a `pickle`. Použití modulu `pickle` si stručně vysvětlíme při probírání ukládání a načítání obrázků. Po příkazech import vytváříme několik vlastních tříd představujících výjimky:

```
class ImageError(Exception): pass
class CoordinateError(ImageError): pass
```

Ukázali jsme si pouze první dvě třídy představující výjimky. Ostatní (`LoadError`, `SaveError`, `ExportError` a `NoFilenameError`) jsou vytvořeny stejným způsobem a jsou odvozeny od třídy `ImageError`. Uživatelé třídy Image mohou testovat kteroukoli ze specifických výjimek nebo jen výjimku ve formě bázové třídy `ImageError`.

Zbývající část modulu tvoří třída `Image`, přičemž na konci jsou standardní tři řádky pro spuštění dokumentačních testů modulu. Před pohledem na třídu a její metody se podíváme na to, jak se používá:

```
border_color = "#FF0000"     # červená
square_color = "#0000FF"     # modrá
width, height = 240, 60
midx, midy = width // 2, height // 2
image = Image.Image(width, height, "square_eye.img")
for x in range(width):
    for y in range(height):
        if x < 5 or x >= width - 5 or y < 5 or y >= height - 5:
            image[x, y] = border_color
        elif midx - 20 < x < midx + 20 and midy - 20 < y < midy + 20:
            image[x, y] = square_color
image.save()
image.export("square_eye.xpm")
```

Všimněte si, že pro nastavení barev v obrázku můžeme použít operátor pro přístup k prvkům (`[]`). Hranaté závorky můžeme použít i pro získávání nebo mazání (kterým vlastně nastavíme barvu na barvu pozadí) barvy na určité souřadnici (x, y). Souřadnice se předávají jako jeden objekt n-tice (díky operátoru čárka), což je stejné, jako kdybychom napsali `image[(x, y)]`. Tento druh hladkého integrování syntaxe lze v Pythonu provést velmi snadno. Stačí jen implementovat příslušné speciální metody, kterými jsou v případě operátoru pro přístup k prvkům metody `__getitem__()`, `_setitem__()` a `__delitem__()`.

Třída `Image` používá pro reprezentaci barev řetězce s hexadecimálními hodnotami ve stylu HTML. Barva pozadí musí být nastavena při vytváření obrázku, jinak se použije výchozí bílá barva. Třída Image ukládá a načítá obrázky ve svém vlastním formátu, dokáže je ale také exportovat do formátu `.xpm`, kterému rozumí řada aplikací pro zpracování obrazu. Obrázek `.xpm` vytvořený výše uvedeným úryvkem kódu si můžete prohlédnout na obrázku 6.6.

Obrázek 6.6: Obrázek square_eye.xpm

Nyní se podíváme na metody třídy `Image`, přičemž začneme řádkem s příkazem `class` a inicializační metodou:

```
class Image:

    def __init__(self, width, height, filename="",
                 background="#FFFFFF"):
        self.filename = filename
        self.__background = background
        self.__data = {}
        self.__width = width
        self.__height = height
        self.__colors = {self.__background}
```

Při vytváření objektu typu Image musí uživatel (tj. uživatel třídy) zadat šířku a výšku, název souboru a barva pozadí jsou však volitelné, protože mají uvedeny výchozí hodnoty. Klíči slovníku `self.__data` jsou souřadnice (x, y) a jeho hodnotami řetězce s barvou. Množina `self.__colors` je inicializována barvou pozadí. Používá se pro sledování jedinečných barev používaných v obrázku.

Všechny datové atributy kromě názvu souboru jsou soukromé, a proto musíme poskytnout nějaké prostředky, pomocí nichž k nim budou moci uživatelé třídy přistupovat. To lze snadno provést pomocí vlastností.*

```
    @property
    def background(self):
        return self.__background

    @property
    def width(self):
        return self.__width

    @property
    def height(self):
```

* V lekci 8 si ukážeme zcela odlišný přístup k poskytování přístupu k atributům, který spočívá v použití speciálních metod `__getattr__()` a `__setattr__()` a který je v některých situacích užitečný.

```
        return self.__height

    @property
    def colors(self):
        return set(self.__colors)
```

Kopírování kolekcí ➢ 146

Při vracení datového atributu z objektu je nutné myslet na to, zda se jedná o neměnitelný či měnitelný typ. Vracení neměnitelného atributu je vždy bezpečné, protože jej nelze změnit, avšak v případě měnitelných atributů musíme zvážit několik věcí. Vracení odkazu na měnitelný atribut je velice rychlé a efektivní, protože nedochází k žádnému kopírování. Na druhou stranu to ale znamená, že volající má nyní přístup k internímu stavu daného objektu, který může změnit takovým způsobem, který objekt uvede do neplatného stavu. Je tedy třeba vzít v potaz zásadu vracet vždy kopii měnitelných datových atributů, dokud profilace neukáže výrazně negativní dopad na výkon. (V tomto případě bychom mohli místo uchovávání množiny jedinečných barev vracet `set(self.__data.values()) | {self.__background}`, kdykoli bude potřeba přistupovat k množině barev. K této problematice se za okamžik vrátíme.)

```
    def __getitem__(self, coordinate):
        assert len(coordinate) == 2, "souřadnice musí být n-tice se dvěma prvky"
        if (not (0 <= coordinate[0] < self.width) or
            not (0 <= coordinate[1] < self.height)):
            raise CoordinateError(str(coordinate))
        return self.__data.get(tuple(coordinate), self.__background)
```

Tato metoda vrací barvu pro zadanou souřadnici pomocí operátoru pro přístup k prvkům (`[]`). Speciální metody pro operátory pro přístup k prvkům a několik dalších speciálních metod souvisejících s kolekcemi uvádí tabulka 6.4.

Tabulka 6.4: Speciální metody pro kolekce

Speciální metoda	Použití	Popis
`__contains__(self, x)`	`x in y`	Vrací hodnotu `True`, pokud se objekt `x` nachází v posloupnosti `y` nebo pokud je objekt `x` klíčem v mapování `y`.
`__delitem__(self, k)`	`del y[k]`	Vymaže prvek na pozici `k` posloupnosti `y` nebo prvek s klíčem `k` v mapování `y`.
`__getitem__(self, k)`	`y[k]`	Vrací prvek na pozici `k` posloupnosti `y` nebo prvek s klíčem `k` v mapování `y`.
`__iter__(self)`	`for x in y: pass`	Vrací iterátor pro prvky posloupnosti `y` nebo pro klíče mapování `y`.
`__len__(self)`	`len(y)`	Vrací počet prvků v `y`.

Speciální metoda	Použití	Popis
`__reversed__(self)`	`reversed(y)`	Vrací zpětný iterátor pro prvky posloupnosti y nebo pro klíče mapování y.
`__setitem__(self, k, v)`	`y[k] = v`	Nastaví prvek na pozici k posloupnosti y nebo hodnotu pro klíč k v mapování y na v.

Rozhodli jsme se aplikovat na přístup k prvkům dvě zásady. První spočívá v tom, že předběžnou podmínkou pro použití metody pro přístup k prvku je to, že zadaná souřadnice je posloupnost délky 2 (obvykle tedy n-tice se dvěma prvky), což zajišťujeme příkazem `assert`. Druhá zásada říká, že přijatelné jsou libovolné hodnoty souřadnic, je-li však jedna z nich mimo rozsah, vyvoláme vlastní výjimku.

K získání barvy pro zadanou souřadnici používáme metodu `dict.get()` s výchozí hodnotou nastavenou na barvu pozadí. Díky tomu máme jistotu, že pokud pro zadanou souřadnici nebyla nastavena žádná barva, vrátí se místo vyvolání výjimky `KeyError` barva pozadí.

```
def __setitem__(self, coordinate, color):
    assert len(coordinate) == 2, "souřadnice musí být n-tice se dvěma prvky"
    if (not (0 <= coordinate[0] < self.width) or
        not (0 <= coordinate[1] < self.height)):
        raise CoordinateError(str(coordinate))
    if color == self.__background:
        self.__data.pop(tuple(coordinate), None)
    else:
        self.__data[tuple(coordinate)] = color
        self.__colors.add(color)
```

Pokud uživatel nastaví hodnotu souřadnice na barvu pozadí, pak můžeme klidně vymazat odpovídající prvek slovníku, protože u libovolné souřadnice neuvedené ve slovníku použijeme barvu pozadí. Musíme ale místo příkazu `del` použít metodu `dict.pop()` s falešným druhým argumentem, protože jen tak se vyhneme vyvolání výjimky `KeyError` v případě, že zadaný klíč (souřadnice) není ve slovníku.

Pokud se barva liší od barvy pozadí, nastavíme ji pro zadanou souřadnici a přidáme do množiny jedinečných barev používaných v obrázku.

```
def __delitem__(self, coordinate):
    assert len(coordinate) == 2, "souřadnice musí být n-tice se dvěma prvky"
    if (not (0 <= coordinate[0] < self.width) or
        not (0 <= coordinate[1] < self.height)):
        raise CoordinateError(str(coordinate))
    self.__data.pop(tuple(coordinate), None)
```

Je-li barva na určité souřadnici smazána, zůstane na této souřadnici barva pozadí. K odstranění prvku používáme opět metodu `dict.pop()`, která funguje dle našich představ správně bez ohledu na to, zda se prvek se zadanou souřadnicí nachází ve slovníku.

Obě metody `__setitem__()` a `__delitem__()` mají potenciál naplnit množinu barev více barvami, než kolik jich obrázek skutečně používá. Pokud například na určitém pixelu vymažeme jedinečnou barvu odlišnou od barvy pozadí, zůstane tato barva v množině barev, přestože se již nepoužívá. A podobně pokud má nějaký pixel jedinečnou barvu odlišnou od barvy pozadí a je nastaven na barvu pozadí, jedinečná barva se již nepoužívá, ale zůstává v množině barev. To znamená, že v nejhorším případě by množina barev mohla obsahovat více barev, než kolik jich ve skutečnosti obrázek používá (nikdy však méně).

Rozhodli jsme se přijmout záležitost s případným větším počtem barev v množině barev, než kolik se jich skutečně používá, z důvodu lepšího výkonu, tj. kvůli maximalizaci rychlosti nastavování a mazání barev, zvláště pak s přihlédnutím k faktu, že uchovávání několika barev navíc obvykle nepředstavuje velký problém. Pokud chceme mít jistotu, že je množina barev synchronizovaná, pak bychom samozřejmě mohli buď vytvořit další metodu, kterou bychom volali dle potřeby, nebo přijmout vyšší nároky na výkon a provádět výpočet automaticky, kdykoliv by to bylo zapotřebí. V obou případech je kód velice jednoduchý (a používá se při načítání nového obrázku):

```
self.__colors = (set(self.__data.values()) | {self.__background})
```

Tímto způsobem jednoduše přepíšeme množinu barev množinou barev skutečně používaných v obrázku sjednocenou s barvou pozadí.

Metodu `__len__()` jsme neimplementovali, protože to by pro dvourozměrný objekt nedávalo smysl. Dále nemůžeme poskytovat reprezentační formu, poněvadž třídu Image nelze plně vytvořit jen na základě volání `Image()`, a proto neposkytujeme implementace pro metodu `__repr__()` (nebo `__str__()`). Pokud uživatel zavolá na objektu typu `Image` funkci `repr()` nebo `str()`, vrátí implementace metody `object.__repr__()` v bázové třídě vhodný řetězec, například `'<Image.Image object at 0x9c794ac>'`. Jedná se o standardní formát používaný pro objekty nevhodné pro funkci `eval()`. Šestnáctkové číslo představuje ID objektu, které je sice jedinečné (obvykle jde o adresu objektu v paměti), ale jen dočasné.

Chceme, aby uživatelé třídy `Image` byly schopni ukládat a načítat svá obrazová data, a proto poskytujeme dvě metody `save()` a `load()`, které mají tyto činnosti na starost.

Data jsme se rozhodli ukládat pomocí *nakládání* (pickling). V řeči Pythonu je nakládání jistým způsobem serializace (převedení na posloupnost bajtů nebo na řetězec) objektu Pythonu. Silnou zbraní nakládání je to, že naložený objekt může mít datový typ představující kolekci, jako je seznam nebo slovník, a i kdyby měl v sobě naložený objekt další objekty (včetně dalších kolekcí, které mohou obsahovat další kolekce atd.), naloží se s celým obsahem, a navíc bez duplicitních objektů, které se objevují více než jednou.

Upozornění: Naložený objekt lze načíst zpět přímo do proměnné Pythonu, přičemž nemusíme sami provádět žádnou analýzu ani jiné interpretace. Použití naložených objektů je tedy ideální pro ukládání a načítání ad hoc kolekcí dat, zvláště u malých prográmků a u programů vytvořených pro osobní použití. Nicméně naložené objekty nemají žádné bezpečnostní mechanizmy (žádné šifrování, žádný digitální podpis), takže načítání naloženého objektu, který pochází z nedůvěryhodného zdroje, může být nebezpečné. Z tohoto důvodu je u programů, které nejsou určeny čistě pro osobní použití, nejlepší vytvořit vlastní formát souboru, který je specifický pro daný program. V lekci 7 si ukážeme, jak číst a zapisovat vlastní souborové formáty pro binární data, text a kód jazyka XML.

```
    def save(self, filename=None):
        if filename is not None:
            self.filename = filename
        if not self.filename:
            raise NoFilenameError()

        fh = None
        try:
            data = [self.width, self.height, self.__background,
                    self.__data]
            fh = open(self.filename, "wb")
            pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)
        except (EnvironmentError, pickle.PicklingError) as err:
            raise SaveError(str(err))
        finally:
            if fh is not None:
                fh.close()
```

V první části této funkce se zabýváme výhradně názvem souboru. Pokud byl objekt typu Image vytvořen bez názvu souboru a dosud nebyl žádný název souboru nastaven, pak je nutné předat metodě `save()` explicitní název souboru (v tomto případě se chová jako příkaz „uložit jako“ a aktualizuje interně používaný název souboru). Není-li zadán název souboru, použije se aktuální, a pokud aktuální název souboru není k dispozici a žádný nebyl zadán, vyvolá se výjimka.

Vytváříme seznam (`data`) pro uchovávání objektů, které chceme uložit, což zahrnuje slovník `self.__data` s prvky souřadnice-barva, ale bez množiny jedinečných barev, kterou lze zrekonstruovat. Poté otevíráme soubor pro zápis v binárním režimu a voláme funkci `pickle.dump()`, která do něj zapíše objekt `data`. A to je vše!

Modul `pickle` umí serializovat data pomocí nejrůznějších formátů (označovaných v dokumentaci jako protokoly), z nichž jeden lze uvést jako třetí argument funkce `pickle.dump()`. Protokol 0 znamená ASCII a používá se pro ladění. My jsme použili protokol 3 (`pickle.HIGHEST_PROTOCOL`), což je kompaktní binární formát, kvůli kterému jsme museli otevřít soubor v binárním režimu. Při čtení naložených objektů se žádný protokol neuvádí. Funkce `pickle.load()` je dostatečně chytrá, aby si protokol zjistila sama.

```
    def load(self, filename=None):
        if filename is not None:
            self.filename = filename
        if not self.filename:
            raise NoFilenameError()

        fh = None
        try:
            fh = open(self.filename, "rb")
            data = pickle.load(fh)
            (self.__width, self.__height, self.__background,
```

```
             self.__data) = data
            self.__colors = (set(self.__data.values()) |
                             {self.__background})
        except (EnvironmentError, pickle.UnpicklingError) as err:
            raise LoadError(str(err))
        finally:
            if fh is not None:
                fh.close()
```

Na začátku této funkce se snažíme získat název souboru, který se má načíst, což probíhá stejně jako ve funkci `save()`. Soubor musíme otevřít pro čtení v binárním režimu a data načteme pomocí jediného příkazu `data = pickle.load(fh)`. Objekt data je přesnou rekonstrukcí toho, který jsme uložili, takže v tomto případě jde o seznam obsahující celá čísla reprezentující šířku a výšku, řetězec s barvou pozadí a slovník s prvky souřadnice-barva. Pro přiřazení každého prvku slovníku `data` do příslušné proměnné používáme rozbalení n-tice, takže jakákoli dříve uložená obrazová data jsou (správně) ztracena.

Množinu jedinečných barev zrekonstruujeme vytvořením množiny všech barev uložených ve slovníku souřadnice-barva, k níž přidáme barvu pozadí.

```
    def export(self, filename):
        if filename.lower().endswith(".xpm"):
            self.__export_xpm(filename)
        else:
            raise ExportError("nepodporovaný formnát pro export: " +
                              os.path.splitext(filename)[1])
```

Implementovali jsme jednu generickou metodu pro export, která podle přípony souboru určí, která soukromá metoda se má zavolat, nebo v případě formátu souboru, do něhož nelze exportovat, vyvolá výjimku. V tomto případě podporujeme pouze ukládání do souborů `.xpm` (a navíc pouze pro obrázky s méně než 8 930 barvami). Metodu `__export_xpm()` zde neuvádíme, protože s látkou probíranou v této lekci příliš nesouvisí. Samozřejmě ji ale najdete ve zdrojovém kódu dodávaném k této knize.

Dokončili jsme výklad naší vlastní třídy `Image`. Tato třída je typickým zástupcem tříd používaných pro uchovávání dat specifických pro určitý program. Poskytuje přístup k datovým prvkům, které obsahuje, a umožňuje všechna svá data ukládat na disk a načítat z disku, přičemž nabízí pouze základní metody, které potřebuje. V následujícím textu si ukážeme, jak vytvořit dva vlastní generické typy, které představují kolekci a nabízejí kompletní rozhraní API.

Tvorba tříd představujících kolekce pomocí agregace

V tomto oddílu vyvineme vlastní kompletní datový typ `SortedList` představující kolekci, který uchovává seznam prvků v seřazeném pořadí. Tyto prvky jsou seřazeny pomocí jejich operátoru menší než (`<`) poskytovaného speciální metodou `__lt__()` nebo pomocí klíčové funkce, je-li zadána. U této třídy se pokoušíme vytvořit rozhraní API shodné s vestavěnou třídou `list`, aby se co nejsnadněji osvojovala a používala, některé metody ale musíme vypustit, protože by nedávaly smysl. Například výsledkem operátoru spojení (`+`) mohou být neseřazené prvky, a proto jej implementovat nebudeme.

Jako vždy při vytváření vlastních tříd se musíme i nyní rozhodnout, zda použít odvození od podobné třídy nebo vytvořit třídu úplně od začátku a agregovat instance dalších potřebných tříd uvnitř anebo tyto dva postupy nějak zkombinovat. Pro třídu `SortedList` v tomto oddílu použijeme agregaci (a samozřejmě implicitní odvození od třídy `object`), přičemž v následujícím oddílu vytvoříme třídu `SortedDict`, která bude používat agregaci i dědičnost (odvodíme ji od třídy `dict`).

V lekci 8 uvidíme, že třídy se mohou tvářit, že poskytují různá rozhraní API. Například třída list nabízí rozhraní API `MutableSequence`, což znamená, že podporuje operátor in, vestavěné funkce `iter()` a `len()`, operátor pro přístup k prvkům (`[]`) pro čtení, zapisování a mazání prvků a metodu `insert()`. Zde implementovaná třída `SortedList` nepodporuje zapisování prvků a nemá metodu `insert()`, takže rozhraní API `MutableSequence` nenabízí. Pokud bychom třídu `SortedList` vytvořili odvozením od třídy list, mohla by výsledná třída tvrdit, že je měnitelnou posloupností, kompletní rozhraní API `MutableSequence` by ale neměla. Z tohoto důvodu třída `SortedList` nedědí od třídy list, a proto o svém rozhraní API nic netvrdí. Na druhou stranu třída `SortedDict` implementovaná v následujícím oddílu podporuje kompletní rozhraní API `MutableMapping`, které poskytuje třída `dict`, a proto ji můžeme odvodit od třídy `dict`.

Zde je několik základních příkladů použití třídy `SortedList`:

```
letters = SortedList.SortedList(("H", "c", "B", "G", "e"), str.lower)
# str(letters) == "['B', 'c', 'e', 'G', 'H']"
letters.add("G")
letters.add("f")
letters.add("A")
# str(letters) == "['A', 'B', 'c', 'e', 'f', 'G', 'G', 'H']"
letters[2] # vrátí: 'c'
```

Objekt typu `SortedList` agreguje (tvoří jej) dva soukromé atributy, funkci `self.__key()` (uchovávanou jako odkaz na objekt `self.__key`) a seznam `self.__list`.

Klíčová funkce se předává jako druhý argument (nebo pomocí klíčovaného argumentu `key`, není-li zadána úvodní posloupnost). Pokud není zadána žádná klíčová funkce, použije se následující soukromá funkce modulu:

Lambda funkce ➢ 180

```
_identity = lambda x: x
```

Jedná se o funkci představující identitu, takže vrací svůj argument beze změny. Při jejím použití v podobě klíčové funkce třídy `SortedList` to tedy znamená, že řadicím klíčem pro každý objekt seznamu je samotný objekt.

Typ `SortedList` nepovoluje operátoru pro přístup k prvkům (`[]`) tyto prvky měnit (neimplementuje tedy speciální metodu `__setitem__()`) ani neposkytuje metody `append()` nebo `extend()`, protože by mohly narušit řazení. Jedinou možností, jak přidávat prvky, je předat posloupnosti při vytváření objektu typu `SortedList` nebo později použít metodu `SortedList.add()`. Na druhou stranu můžeme bez obav použít operátor pro přístup k prvkům pro čtení a mazání prvků na zadané indexové pozici, neboť žádná z těchto operací nemá vliv na řazení prvků, a proto implementujeme obě speciální metody `__getitem__()` a `__delitem__()`.

Nyní se podíváme na jednotlivé metody třídy a začneme tradičně řádkem s příkazem `class` a inicializační metodou třídy:

```
class SortedList:

    def __init__(self, sequence=None, key=None):
        self.__key = key or _identity
        assert hasattr(self.__key, "__call__")
        if sequence is None:
            self.__list = []
        elif (isinstance(sequence, SortedList) and
              sequence.key == self.__key):
            self.__list = sequence.__list[:]
        else:
            self.__list = sorted(list(sequence), key=self.__key)
```

Název funkce je odkazem na objekt (ukazující na tuto funkci), a proto můžeme uchovávat funkce v proměnných stejně jako jakýkoliv jiný odkaz na objekt. Zde je v proměnné `self.__key` uložen odkaz na klíčovou funkci, která byla zadána, nebo na funkci představující identitu. První příkaz metody se spoléhá na skutečnost, že operátor `or` vrací svůj první operand, má-li v logickém kontextu hodnotu `True` (což má funkce, která je odlišná od hodnoty `None`) nebo druhý operand v opačném případě. Malinko delší, ale zato srozumitelnější by byl příkaz `self.__key = key if key is not None else _identity`.

Jakmile máme klíčovou funkci, použijeme příkaz `assert`, abychom se ujistili, že je volatelná. Vestavěná funkce `hasattr()` vrátí hodnotu `True`, má-li objekt předaný jako první argument atribut, jehož název je zadán jako druhý argument. K dispozici jsou také odpovídající funkce `setattr()` a `delattr()`, kterým se budeme věnovat v lekci 8. Všechny volatelné objekty (například funkce a metody) mají atribut `__call__`.

Aby se tvorba objektů typu `SortedList` maximálně podobala tvorbě objektů typu list, přidali jsme volitelný argument `sequence`, který odpovídá jedinému volitelnému argumentu přijímanému funkcí `list()`. Třída `SortedList` agreguje kolekci typu list v soukromé proměnné `self.__list` a prvky v agregovaném seznamu udržuje v seřazeném pořadí pomocí zadané klíčové funkce.

V klauzuli `elif` používáme testování typu pro zjištění, zda je zadaná posloupnost typu `SortedList`, a je-li tomu tak, zda má stejnou klíčovou funkci jako tento seřazený seznam. Jsou-li tyto podmínky splněny, pak provedeme jen mělkou kopii seznamu, který již nemusíme seřadit. Je-li většina klíčových funkcí vytvářena až v místě použití pomocí lambda funkcí, pak i přesto, že nějaké dvě mohou mít stejný kód, nebudou si při porovnání rovny, a proto v praxi nemusí vůbec dojít k nárůstu výkonu.

```
    @property
    def key(self):
        return self.__key
```

Po vytvoření seřazeného seznamu je jeho klíčová funkce zafixována, takže ji uchováváme v soukromé proměnné, aby ji uživatelé nemohli měnit. Avšak někteří uživatelé na ni mohou chtít získat

odkaz (jak uvidíme v následujícím oddílu), a proto jsme ji zpřístupnili prostřednictvím vlastnosti `key` určené pouze pro čtení.

```
def add(self, value):
    index = self.__bisect_left(value)
    if index == len(self.__list):
        self.__list.append(value)
    else:
        self.__list.insert(index, value)
```

Při zavolání této metody je nutné zadanou hodnotu vložit do soukromého seznamu `self.__list` na správnou pozici, aby seznam zůstal seřazen. Soukromá metoda `SortedList.__bisect_left()` vrací požadovanou indexovou pozici, jak uvidíme za okamžik. Je-li nová hodnota větší než jakákoli jiná hodnota v seznamu, uložíme ji na konec, takže indexová pozice bude odpovídat délce seznamu (indexové pozice seznamu jsou od `0` do `len(L) - 1`). V tomto případě novou hodnotu připojíme, jinak ji vložíme na zadanou indexovou pozici, což bude `0`, je-li nová hodnota menší než všechny ostatní hodnoty v seznamu.

```
def __bisect_left(self, value):
    key = self.__key(value)
    left, right = 0, len(self.__list)
    while left < right:
        middle = (left + right) // 2
        if self.__key(self.__list[middle]) < key:
            left = middle + 1
        else:
            right = middle
    return left
```

Tato soukromá metoda vypočítá indexovou pozici v seznamu, do níž náleží zadaná hodnota, což je indexová pozice, kde se tato hodnota nachází (je-li v seznamu) nebo kde by měla být (není-li v seznamu). Metoda odvodí porovnávací klíč pro zadanou hodnotu pomocí klíčové funkce seřazeného seznamu a srovnává jej s odvozeným porovnávacím klíčem zkoumaných prvků. Používá algoritmus s názvem *binární hledání* (označovaný též jako *binární sek*), který má skvělý výkon i v případě velmi rozsáhlých seznamů. Například pro nalezení pozice pro zadanou hodnotu v seznamu s 1 000 000 prků stačí 21 porovnání.* Pro srovnání si můžeme představit holý neseřazený seznam, který používá lineární hledání a který pro nalezení hodnoty v seznamu s 1 000 000 prvků potřebuje v průměru 500 000 a nejhůře 1 000 000 porovnání.

```
def remove(self, value):
    index = self.__bisect_left(value)
    if index < len(self.__list) and self.__list[index] == value:
        del self.__list[index]
```

* Modul `bisect` Pythonu nabízí funkci `bisect.bisect_left()` a několik dalších, avšak v době psaní této knihy nedokáže žádná z funkcí modulu bisect pracovat s klíčovou funkcí.

```
        else:
            raise ValueError("{0}.remove(x): x not in list".format(
                             self.__class__.__name__))
```

Tato metoda se používá k odstranění prvního výskytu zadané hodnoty. Pro nalezení indexové pozice, kam tato hodnota patří, použije metodu `SortedList.__bisect_left()` a poté provede testy pro zjištění, zda tato indexová pozice leží uvnitř seznamu a zda je prvek na této pozici stejný jako zadaná hodnota. Jsou-li tyto podmínky splněny, daný se prvek odstraní. V opačném případě se vyvolá výjimka `ValueError` (což ve stejné situaci provede také metoda `list.remove()`).

```
    def remove_every(self, value):
        count = 0
        index = self.__bisect_left(value)
        while (index < len(self.__list) and
               self.__list[index] == value):
            del self.__list[index]
            count += 1
        return count
```

Tato metoda je podobná metodě `SortedList.remove()` a představuje rozšíření rozhraní API seznamu. Začíná vyhledáním indexové pozice, na níž se nachází první výskyt hodnoty v seznamu, a poté se provádí cyklus až do okamžiku, kdy je indexová pozice uvnitř seznamu a prvek na indexové pozici je stejný jako zadaná hodnota. Tento kód je malinko zákeřný, protože v každé iteraci je nalezený prvek vymazán, načež se na indexové pozici vymazaného prvku objeví prvek, který následuje po něm.

```
    def count(self, value):
        count = 0
        index = self.__bisect_left(value)
        while (index < len(self.__list) and
               self.__list[index] == value):
            index += 1
            count += 1
        return count
```

Tato metoda vrací počet výskytů zadané hodnoty v seznamu (což může být i `0`). Používá velice podobný algoritmus jako metoda `SortedList.remove_every()`, avšak zde musíme po každé iteraci indexovou pozici inkrementovat.

```
    def index(self, value):
        index = self.__bisect_left(value)
        if index < len(self.__list) and self.__list[index] == value:
            return index
        raise ValueError("{0}.index(x): x not in list".format(
                         self.__class__.__name__))
```

Třída `SortedList` představuje uspořádaný seznam, takže můžeme pro nalezení (či nenalezení) hodnoty v seznamu použít rychlé binární hledání.

```
    def __delitem__(self, index):
        del self.__list[index]
```

Speciální metoda `__delitem__()` poskytuje podporu pro syntaxi `del L[n]`, kde *L* je seřazený seznam a *n* je celé číslo představující indexovou pozici. Test, zda je index mimo rozsah, neprovádíme, protože pokud takový index použijeme při volání `self.__list[index]`, dojde k vyvolání výjimky `IndexError`, což je chování, které chceme.

```
    def __getitem__(self, index):
        return self.__list[index]
```

Tato metoda poskytuje podporu pro syntaxi `x = L[n]`, kde *L* je seřazený seznam a *n* je celé číslo představující indexovou pozici.

```
    def __setitem__(self, index, value):
        raise TypeError("pro vložení hodnoty použijte add() a nechte seznam, "
                        " aby ji umístil na správné místo")
```

Nechceme, aby uživatel prvek na zadané pozici měnil (takže příkaz `L[n] = x` nelze použít), jinak by mohlo dojít k narušení řazení seznamu. Výjimka `TypeError` se používá k signalizaci, že daná operace není určitým datovým typem podporována.

```
    def __iter__(self):
        return iter(self.__list)
```

Implementace této metody je snadná, protože stačí jen pomocí vestavěné funkce `iter()` vrátit iterátor soukromého seznamu. Tato metoda se používá k podpoře syntaxe `for` *`hodnota`* `in` *`iterovatelný_objekt`*.

Je třeba poznamenat, že pokud je požadována posloupnost, pak se použije právě tato metoda. Pro převod objektu *L* typu `SortedList` na holý seznam tedy zavoláme `list(L)` a Python v pozadí zavolá metodu `SortedList.__iter__(L)`, která poskytuje posloupnost požadovanou funkcí `list()`.

```
    def __reversed__(self):
        return reversed(self.__list)
```

Tato metoda poskytuje podporu pro vestavěnou funkci `reversed()`, takže můžeme například napsat `for` *`hodnota`* `in reversed(`*`iterovatelný_prvek`*`)`.

```
    def __contains__(self, value):
        index = self.__bisect_left(value)
        return (index < len(self.__list) and
                self.__list[index] == value)
```

Metoda `__contains__()` poskytuje podporu pro operátor `in`. Opět je třeba zdůraznit, že oproti pomalému lineárnímu vyhledávání obyčejného seznamu můžeme použít rychlé binární hledání.

```
    def clear(self):
        self.__list = []
```

```
    def pop(self, index=-1):
        return self.__list.pop(index)

    def __len__(self):
        return len(self.__list)

    def __str__(self):
        return str(self.__list)
```

Metoda `SortedList.clear()` zahodí stávající seznam a nahradí jej novým prázdným seznamem. Metoda `SortedList.pop()` odstraní a vrátí prvek na zadané indexové pozici nebo vyvolá výjimku `IndexError`, je-li tato indexová pozice mimo rozsah. U metod `pop()`, `__len__()` a `__str__()` jen předáváme práci agregovanému objektu `self.__list`.

Reimplementaci speciální metody `__repr__()` nemáme, a proto se v okamžiku, kdy uživatel napíše `repr(L)`, kde `L` je objekt typu `SortedList`, zavolá metoda `object.__ repr__()` bázové třídy. Tím se vytvoří řetězec, jako je například `'<SortedList.SortedList object at 0x97e7cec>'`, ovšem s tím, že šestnáctkové ID může být samozřejmě jiné. Smysluplnou implementaci metody `__repr__()` poskytnout nemůžeme, protože bychom potřebovali uvést také klíčovou funkci, avšak odkaz na funkční objekt nelze převést na řetězec vhodný pro funkci `eval()`.

Metody `insert()`, `reverse()` a `sort()` jsme neimplementovali, protože žádná z nich není pro naši třídu vhodná. Pokud se některá z nich zavolá, vyvolá se výjimka `AttributeError`.

Pokud kopírujeme seřazený seznam pomocí řezu `L[:]`, neobdržíme objekt typu `SortedList`, ale objekt typu `list`. Nejsnadnějším způsobem k provedení kopie je importovat modul `copy` a použít funkci `copy.copy()`, která je dostatečně chytrá na to, aby bez jakékoli pomoci zkopírovala seřazený seznam (a instance většiny dalších vlastních tříd). Nicméně jsme se rozhodli poskytnout explicitní metodu `copy()`:

```
    def copy(self):
        return SortedList(self, self.__key)
```

Předáním `self` jako prvního argumentu zajistíme, že se `self.__list` nebude muset kopírovat a opětovně řadit, ale že se provede mělká kopie (díky testování typu v klauzuli `elif` v metodě `__init__()`). Teoretická výhoda vyššího výkonu tohoto způsobu kopírování není funkci `copy.copy()` dostupná, což snadno napravíme přidáním následujícího řádku:

```
    __copy__ = copy
```

Při zavolání funkce `copy.copy()` se tato pokusí použít speciální metody objektu `__copy__()`, přičemž se vrátí pět ke svému kódu, pokud tato metoda neexistuje. Díky tomuto řádku může nyní funkce `copy.copy()` použít metodu `SortedList.copy()` pro seřazené seznamy. (Dále je možné implementovat speciální metodu `__deepcopy__()`, která je ale poněkud komplikovanější – veškeré podrobnosti najdete ve webové dokumentaci k modulu `copy`.)

Dokončili jsme implementaci třídy `SortedList`. V následujícím oddílu ji použijeme jako seřazený seznam klíčů pro třídu `SortedDict`.

Tvorba tříd představujících kolekce pomocí dědičnosti

collections.OrderedDict ➤ 136

Třída `SortedDict`, kterou si ukážeme v tomto oddílu, se pokouší v maximální možné míře napodobit třídu `dict`. Hlavní odlišností je to, že klíče objektu typu `SortedDict` jsou vždy seřazené na základě zadané klíčové funkce nebo funkce představující identitu. Třída `SortedDict` nabízí stejné rozhraní API jako třída `dict` (tedy až na to, že při aplikaci funkce `repr()` neposkytuje reprezentaci vhodnou pro funkci `eval()`) plus dvě metody navíc, které mají smysl pouze pro uspořádanou kolekci.* (Je třeba poznamenat, že Python 3.1 zavádí třídu `collections.OrderedDict`, která se od naší třídy `SortedDict` liší, protože nemá seřazené klíče, ale je seřazena podle pořadí, v jakém jsou prvky vkládány.)

Zde je několik příkladů použití, z nichž si můžeme udělat představu o tom, jak třída `SortedDict` funguje:

```
d = SortedDict.SortedDict(dict(s=1, A=2, y=6), str.lower)
d["z"] = 4
d["T"] = 5
del d["y"]
d["n"] = 3
d["A"] = 17
str(d) # vrátí: "{'A': 17, 'n': 3, 's': 1, 'T': 5, 'z': 4}"
```

Při implementace třídy `SortedDict` používáme agregaci i dědičnost. Seřazený seznam klíčů je agregován jako proměnná instance, zatímco samotná třída `SortedDict` je odvozena od třídy `dict`. Prohlídku kódu začneme pohledem na řádek s příkazem `class` a na inicializační metodu a poté se postupně podíváme na všechny ostatní metody.

```
class SortedDict(dict):

    def __init__(self, dictionary=None, key=None, **kwargs):
        dictionary = dictionary or {}
        super().__init__(dictionary)
        if kwargs:
            super().update(kwargs)
        self.__keys = SortedList.SortedList(super().keys(), key)
```

V příkazu `class` uvádíme bázovou třídu `dict`. Inicializační metoda se pokouší napodobit funkci `dict()`, přidává ale druhý argument pro klíčovou funkci. Voláním `super().__init__()` inicializujeme objekt typu `SortedDict` pomocí metody bázové třídy `dict.__init__()`. Podobně pokud byly použity klíčované argumenty, použijeme metodu bázové třídy `dict.update()` pro jejich přidání do slovníku. (Je třeba poznamenat, že je přijímán pouze jeden výskyt klíčovaného argumentu, takže žádný z klíčů v klíčovaných argumentech `kwargs` nesmí být „dictionary“ nebo „key“.)

Kopii všech klíčů slovníku udržujeme v seřazeném seznamu v proměnné `self.__keys`. Klíče slovníku předáváme pomocí metody bázové třídy `dict.keys()` pro inicializaci seřazeného seznamu. Meto-

* Zde prezentovaná třída `SortedDict` je odlišná od té, která je uvedena v knize s původním názvem „Rapid GUI Programming with Python and Qt“ (ISBN 0132354187) od téhož autora, a také od té, která je k dispozici přes Seznam balíčků pro jazyk Python (Python Package Index).

du `SortedDict.keys()` použít nemůžeme, protože ta se opírá o proměnnou `self.__keys`, která ale bude existovat až po vytvoření klíčů objektu typu `SortedList`.

```
def update(self, dictionary=None, **kwargs):
    if dictionary is None:
        pass
    elif isinstance(dictionary, dict):
        super().update(dictionary)
    else:
        for key, value in dictionary.items():
            super().__setitem__(key, value)
    if kwargs:
        super().update(kwargs)
    self.__keys = SortedList.SortedList(super().keys(),
                                        self.__keys.key)
```

Tato metoda se používá pro aktualizaci jedněch prvků slovníku jinými prvky slovníku, klíčovanými argumenty nebo obojím. Prvky, které existují pouze ve druhém slovníku, se přidají do tohoto slovníku a pro prvky, jejichž klíče se objevují v obou slovnících, nahradíme původní hodnoty hodnotami z druhého slovníku. Toto chování jsme museli malinko rozšířit v tom, že neměníme klíčovou funkci původního slovníku, a to ani tehdy, je-li druhým slovníkem objekt typu `SortedDict`.

Aktualizace probíhá ve dvou fázích. Nejdříve aktualizujeme prvky slovníku. Je-li zadaný slovník odvozený od tříd `dict` (což samozřejmě zahrnuje také třídu `SortedDict`), použijeme k provedení aktualizace metodu bázové třídy `dict.update()`. Použití verze z bázové třídy je zásadní pro zamezení rekurzivního volání metody `SortedDict.update()` končícího nekonečným cyklem. Pokud slovník není typu `dict`, procházíme jeho prvky a nastavujeme každou dvojici klíč-hodnota samostatně. (Pokud objekt slovníku není typu `dict` a nemá metodu `items()`, vyvoláme po právu výjimku `AttributeError`.) Jsou-li použity klíčované argumenty, začleníme je opětovným zavoláním metody bázové třídy `update()`.

Důsledkem aktualizování je to, že seznam `self.__keys` začne být zastaralý, a proto jej nahradíme novým objektem typu `SortedList` s klíči slovníku (které opět získáme z bázové třídy, poněvadž metoda `SortedDict.keys()` se opírá o seznam `self.__keys`, který právě aktualizujeme) a klíčovou funkcí původního seřazeného seznamu.

```
@classmethod
def fromkeys(cls, iterable, value=None, key=None):
    return cls({k: value for k in iterable}, key)
```

Rozhraní API třídy `dict` obsahuje metodu třídy `dict.fromkeys()`. Tato metoda se používá k vytvoření nového slovníku na základě iterovatelného objektu. Každý prvek v iterovatelném objektu se stane klíčem a hodnota každého klíče bude `None` nebo zadaná hodnota.

Vzhledem k tomu, že se jedná o metodu třídy, je prvním argumentem třída, kterou automaticky poskytuje Python. Pro třídu `dict` bude třída `dict` a pro `SortedDict` bude `SortedDict`. Návratovou hodnotou je slovník zadané třídy. Například:

```
class MyDict(SortedDict.SortedDict): pass
d = MyDict.fromkeys("VEINS", 3)
str(d)              # vrátí: "{'E': 3, 'I': 3, 'N': 3, 'S': 3, 'V': 3}"
d.__class__.__name__ # vrátí: 'MyDict'
```

Takže při zavolání zděděné metody třídy se její proměnná `cls` nastaví na správnou třídu, tedy stejně jako v případě zavolání běžných metod, při němž se jejich proměnná `self` nastaví na aktuální objekt. Ve srovnání se statickou metodou jde o odlišné chování, které je lepší, protože statická metoda je svázána s konkrétní třídou a neví, zda byla provedena v kontextu své původní třídy nebo nějaké podtřídy.

```
    def __setitem__(self, key, value):
        if key not in self:
            self.__keys.add(key)
        return super().__setitem__(key, value)
```

Tato metoda implementuje syntaxi *d*`[`*klíč*`] = `*hodnota*. Pokud klíč není ve slovníku, přidáme jej do seznamu klíčů, přičemž se spoléháme na to, že jej objekt typu `SortedList` umístí na správné místo. Poté zavoláme metodu bázové třídy a její výsledek vrátíme volajícímu kvůli podpoře řetězením, například `x = d[klíč] = hodnota`.

Všimněte si, že v příkazu `if` kontrolujeme, zda klíč již v objektu typu `SortedDict` existuje, příkazem `not in self`. Třída `SortedDict` je odvozena od třídy `dict`, a proto ji lze použít všude tam, kde je očekáván typ `dict`, přičemž v tomto případě je `self` typu `SortedDict`. Když při reimplementaci metod třídy `dict` ve třídě `SortedDict` potřebujeme zavolat implementaci v bázové třídě, musíme dbát na to, abychom tuto metodu zavolali pomocí funkce `super()`, jak jsme to provedli v posledním příkazu výše uvedené metody. Díky tomu máme jistotu, že reimplementovaná metoda nezavolá sebe samu, což by vedlo k nekonečné rekurzi.

Metoda `__getitem__()` není reimplementována, protože verze v bázové třídě nám vyhovuje a na řazení klíčů nemá žádný vliv.

```
    def __delitem__(self, key):
        try:
            self.__keys.remove(key)
        except ValueError:
            raise KeyError(key)
        return super().__delitem__(key)
```

Tato metoda poskytuje syntaxi `del` *d*`[`*klíč*`]`. Pokud klíč neexistuje, způsobí volání metody `SortedList.remove()` vyvolání výjimky `ValueError`. V takovém případě výjimku zachytíme a vyvoláme místo ní výjimku `KeyError`, čímž se přiblížíme k chování rozhraní API třídy `dict`. V opačném případě vrátíme výsledek volání implementace v bázové třídě, která vymaže prvek se zadaným klíčem ze samotného slovníku.

```
    def setdefault(self, key, value=None):
        if key not in self:
            self.__keys.add(key)
        return super().setdefault(key, value)
```

Tato metoda vrátí hodnotu pro zadaný klíč, nachází-li se ve slovníku, jinak vytvoří nový prvek se zadaným klíčem a hodnotou a vrátí tuto hodnotu. U třídy `SortedDict` musíme zajistit, aby se klíč, který dosud není ve slovníku, přidal do seznamu klíčů.

```
def pop(self, key, *args):
    if key not in self:
        if len(args) == 0:
            raise KeyError(key)
        return args[0]
    self.__keys.remove(key)
    return super().pop(key, args)
```

Je-li zadaný klíč ve slovníku, pak tato metoda vrátí odpovídající hodnotu a odstraní prvek klíč-hodnota ze slovníku. Klíč je též nutné odstranit ze seznamu klíčů.

Tato implementace je docela zákeřná, protože metoda `pop()` musí pro napodobení metody `dict.pop()` podporovat dvě různá chování. První je *`d`*`.pop(`*`k`*`)`. Zde se vrací hodnota pro klíč *`k`* nebo se vyvolá výjimka `KeyError`, pokud neexistuje. Druhé je *`d`*`.pop(`*`k, v`*`)`. Zde se vrací hodnota pro klíč *`k`* nebo hodnota *`v`* (která může být `None`), pokud neexistuje. V obou případech se klíč *`k`* odstraní, pokud existuje.

```
def popitem(self):
    item = super().popitem()
    self.__keys.remove(item[0])
    return item
```

Metoda `dict.popitem()` odstraní ze slovníku a vrátí náhodný prvek klíč-hodnota. Nejdříve musíme zavolat verzi z bázové třídy, protože nevíme dopředu, který prvek se má odstranit. Odstraníme klíč prvku ze seznamu klíčů a poté prvek vrátíme.

```
def clear(self):
    super().clear()
    self.__keys.clear()
```

Zde odstraníme všechny prvky slovníku a všechny prvky seznamu klíčů.

```
def values(self):
    for key in self.__keys:
        yield self[key]

def items(self):
    for key in self.__keys:
        yield (key, self[key])

def __iter__(self):
    return iter(self.__keys)

keys = __iter__
```

Slovníky obsahují čtyři metody, které vracejí iterátory: `dict.values()` pro hodnoty slovníku, `dict.items()` pro prvky klíč-hodnota slovníku, `dict.keys()` pro klíče a speciální metodu `__iter__()`, která poskytuje podporu pro syntaxi `iter(d)` a pracuje na klíčích. (Verze těchto metod v bázové třídě ve skutečnosti vracejí slovníkové pohledy, avšak pro většinu účelů je zde implementované chování iterátorů stejné.)

Metody `__iter__()` a `keys()` mají identické chování, a proto stačí místo implementace metody `keys()` jednoduše vytvořit odkaz na objekt s názvem `keys` a nastavit jej tak, aby ukazoval na metodu `__iter__()`. Díky tomu mohou uživatelé třídy `SortedDict` volat pro získání iterátoru přes klíče slovníku `d.keys()` nebo `iter(d)`, tedy stejně, jako mohou voláním `d.values()` získat iterátor přes hodnoty slovníku. Metody `values()` a `items()` jsou generátorové metody. Stručné vysvětlení generátorových metod najdete v panelu „Generátorové funkce“. V obou případech procházejí seznam seřazených klíčů, takže vždy vracejí iterátory, které procházejí prvky v pořadí daném pořadím klíčů (které závisí na klíčové funkce zadané inicializační metodě). V metodách `items()` a `values()` se hodnoty hledají pomocí syntaxe `d[k]` (která v pozadí používá metodu `dict.__getitem__()`), protože můžeme s objektem `self` zacházet jako s objektem typu `dict`.

Generátory ➤ 332

Generátorové funkce

Generátorová funkce nebo *generátorová metoda* je taková funkce či metoda, která obsahuje výraz `yield`. Generátorová funkce při svém zavolání vrátí iterátor. Hodnoty se extrahují z tohoto iterátoru vždy po jedné voláním metody `__next__()`. Při každém zavolání metody `__next__()` se vrátí hodnota výrazu `yield` generátorové funkce (nebo `None`, pokud není uveden). Pokud generátorová funkce skončí nebo provede příkaz return, vyvolá se výjimka `StopIteration`.

V praxi jen málokdy voláme metodu `__next__()` nebo zachytáváme výjimku `StopIteration`. Spíše používáme generátor jako jakýkoliv jiný iterovatelný prvek. Zde jsou dvě téměř ekvivalentní funkce. Funkce na levé straně vrací seznam a funkce na pravé straně vrací generátor.

```
# Sestaví a vrátí seznam
def letter_range(a, z):
    result = []
    while ord(a) < ord(z):
        result.append(a)
        a = chr(ord(a) + 1)
    return result
```

```
# Vrací každou hodnotu na požádání
def letter_range(a, z):
    while ord(a) < ord(z):
        yield a
        a = chr(ord(a) + 1)
```

Výsledek volání kterékoli z těchto funkcí můžeme procházet pomocí cyklu `for`, například `for letter in letter_range("m", "v"):`. Pokud ale chceme seznam výsledných písmen, bude nám v případě funkce na levé straně stačit volání `letter_range("m", "v")`, pro funkci na pravé straně ale musíme použít `list(letter_range("m", "v"))`.

Generátorovým funkcím a metodám (a také generátorovým výrazům) se budeme podrobně věnovat v lekci 8.

```
    def __repr__(self):
        return object.__repr__(self)

    def __str__(self):
        return ("{" + ", ".join(["{0!r}: {1!r}".format(k, v)
                                 for k, v in self.items()]) + "}")
```

Reprezentaci objektu typu `SortedDict` vhodnou pro funkci `eval()` poskytnout nemůžeme, protože nemůžeme vytvořit reprezentaci klíčové funkce vhodnou pro funkci `eval()`. Proto v reimplementaci metody `__repr__()` přeskakujeme metodu `dict.__repr__()` a místo ní voláme verzi nejzákladnější bázové třídy `object.__repr__()`. Tímto způsobem obdržíme řetězec, který se používá pro reprezentace nevhodné pro funkci `eval()`, například `'<SortedDict.SortedDict object at 0xb71fff5c>'`.

Metodu `SortedDict.__str__()` jsme implementovali sami, protože chceme, aby výstup zobrazil prvky seřazené podle klíčů. Tuto metodu bychom mohli napsat také takto:

```
items = []
for key, value in self.items():
    items.append("{0!r}: {1!r}".format(key, value))
return "{" + ", ".join(items) + "}"
```

Použití seznamové komprehenze je ale kratší a nenutí nás přidávat dočasnou proměnnou `items`.

Metody bázové třídy `dict.get()`, `dict.__getitem__()` (pro syntaxi `v = d[k]`), `dict.__len__()` (pro `len(d)`) a `dict.__contains__()` (pro `x in d`) fungují dobře tak, jak jsou, a nemají žádný vliv na uspořádání klíčů, takže je nebylo nutné reimplementovat.

Poslední metodou třídy `dict`, kterou musíme reimplementovat, je metoda `copy()`:

```
    def copy(self):
        d = SortedDict()
        super(SortedDict, d).update(self)
        d.__keys = self.__keys.copy()
        return d
```

Nejjednodušší reimplemetnace by měla tvar `def copy(self): return SortedDict(self)`. My jsme však zvolili trošku komplikovanější řešení, díky kterému se vyhneme opětovnému řazení již seřazených klíčů. Vytváříme prázdný seřazený slovník, který poté aktualizujeme prvky v původním seřazeném slovníku pomocí metody bázové třídy `dict.update()`, čímž se vyhneme reimplementované verzi `SortedDict.update()`, a objekt slovníku `self.__keys` typu `SortedList` nahradíme mělkou kopií původního.

Když zavoláme funkci `super()` bez argumentů, pak pracuje s bázovou třídou a objektem `self`. Můžeme ji však explicitním zadáním třídy a objektu přimět pracovat s jakoukoli třídou a jakýmkoli objektem. Při použití této syntaxe pracuje volání funkce `super()` s *bezprostřední* bázovou třídou zadané třídy, také v tomto případě má uvedený kód stejný efekt (a mohl by být zapsán) jako `dict.update(d, self)`.

Vzhledem ke skutečnosti, že řadicí algoritmus Pythonu je velice rychlý a zvláště dobře optimalizovaný pro částečně seřazené seznamy, je nárůst výkonu snad až na obrovské slovníky jen malý nebo vůbec žádný. Nicméně z implementace je patrné, že alespoň z principialního hlediska může být vlastní metoda `copy()` efektivnější než používání příkazu ve stylu `kopie_objektu_x = TřídaX(x)`, který podporují vestavěné typy jazyka Python. A stejně jako v případě třídy `SortedList` jsme i zde provedli přiřazení `__copy__ = copy`, takže funkce `copy.copy()` používá místo svého kódu naši vlastní kopírovací metodu.

```
def value_at(self, index):
    return self[self.__keys[index]]

def set_value_at(self, index, value):
    self[self.__keys[index]] = value
```

Tyto dvě metody představují rozšíření rozhraní API třídy `dict`. Třída `SortedDict` je na rozdíl od holé třídy `dict` seřazená, z čehož vyplývá, že můžeme používat indexové pozice klíčů. Například první prvek ve slovníku je na indexové pozici `0` a poslední na pozici `len(d)-1`. Obě metody pracují na prvcích slovníku, jejichž klíč je v seřazeném seznamu klíčů na zadané indexové pozici. Díky dědičnosti můžeme hodnoty v objektu typu `SortedDict` vyhledat aplikací operátoru pro přístup k prvkům (`[]`) přímo na objekt `self`, který je zároveň typu `dict`. Pokud uživatel zadá index mimo rozsah, vyvolají metody výjimku `IndexError`.

Dokončili jsme implementaci třídy `SortedDict`. Nebývá příliš často nutné vytvářet kompletní třídy představující generickou kolekci, jako je například tato třída. Když už se ale do této činnosti musíme pustit, můžeme naši třídu díky speciálním metodám Pythonu plně integrovat, takže její uživatelé s ní mohou pracovat jako s jakoukoli jinou vestavěnou třídou nebo třídou standardní knihovny.

Shrnutí

V této lekci jsme probírali základy podpory jazyka Python pro objektově orientované programování. Na začátku jsme si ukázali několik nevýhod čistě procedurálního přístupu a také způsob, jak bychom se jim mohli pomocí objektové orientace vyhnout. Pak jsme si popsali některé pojmy z nejčastěji používané terminologie objektově orientovaného programování včetně pojmů „duplicitních“, jako je například *bázová třída* a *nadtřída*.

Viděli jsme, jak vytvořit jednoduché třídy s datovými atributy a vlastními metodami. Dále jsme viděli, jak lze třídy odvozovat od jiných tříd, jak přidávat dodatečné atributy a metody a jak lze metody „odimplementovat“. Odimplementace je potřebná, když dědíme od nějaké třídy a zároveň chceme omezit metody poskytované naší podtřídou. Tuto techniku bychom však měli používat s opatrností, protože porušuje očekávání, že podtřídu lze použít všude tam, kde lze použít některou z jejích bázových tříd – jinými slovy porušuje polymorfismus.

Vlastní třídy lze hladce integrovat, takže podporují stejné syntaxe jako vestavěné třídy a třídy standardní knihovny Pythonu. Této integrace docílíme implementováním speciálních metod. Ukázali jsme si, jak implementovat speciální metody na podporu porovnávání, jak poskytovat reprezentační a řetězcové formy a jak v případě smysluplného výsledku poskytovat převod na jiné typy, jako je

`int` a `float`. Dále jsme si ukázali, jak implementovat metodu `__hash__()`, aby bylo možné instance vlastní třídy použít jako klíče slovníku nebo prvky množiny.

Samotné datové atributy nabízejí mechanizmus pro zajištění, že jsou nastaveny na platné hodnoty. Viděli jsme, jak je snadné nahradit datové atributy vlastnostmi, což nám umožňuje vytvářet vlastnosti určené pouze pro čtení a u zapisovatelných vlastností snadno provádět validaci.

Námi vytvářené třídy jsou většinou „nekompletní", poněvadž máme sklon poskytovat pouze ty metody, které skutečně potřebujeme. To sice v Pythonu funguje skvěle, kromě toho je ale možné vytvářet vlastní kompletní třídy, které poskytují každou relevantní metodu. Ukázali jsme si, jak to provést pro třídy s jedinou hodnotou s použitím agregace nebo kompaktněji pomocí dědičnosti. Dále jsme viděli, jak to provést pro třídy s více hodnotami (tj. pro třídy představující kolekce). Vlastní třídy představující kolekce mohou poskytovat stejné možnosti jako vestavěné třídy představující kolekce, a to včetně podpory pro funkce `len()`, `iter()`, `reversed()` a pro operátor pro přístup k prvkům (`[]`).

Dozvěděli jsme si, že vytváření a inicializace objektů jsou samostatné operace a že Python nám umožňuje obě kontrolovat, ačkoliv v drtivé většině případů si vystačíme s přizpůsobením inicializace. Dále jsme se dozvěděli, že ačkoliv je vždy bezpečné vracet neměnitelné datové atributy objektu, běžně bychom měli vždy vracet jen kopie měnitelných datových atributů objektu, čímž zamezíme úniku interního stavu objektu a jeho neúmyslné změně.

Python nabízí běžné metody, statické metody, metody třídy a funkce modulu. Viděli jsme, že většina metod jsou běžné metody, přičemž občas se hodí také metody třídy. Statické metody se používají jen zřídka, protože metody třídy nebo funkce modulu jsou téměř vždy lepší alternativou.

Vestavěná metoda `repr()` volá speciální metodu objektu `__repr__()`. Je-li možné, tak platí `eval(repr(x)) == x`, přičemž jsme si ukázali, jak podporu pro funkci `eval()` realizovat. Pokud nelze vytvořit řetězcovou reprezentaci vhodnou pro funkci `eval()`, použijeme metodu bázové třídy `object.__repr__()` pro vytvoření reprezentace ve standardním formátu, která se pro funkci `eval()` nehodí.

Testování typu pomocí vestavěné funkce `isinstance()` může nabídnout určité výhody v možnosti zlepšit výkon, ačkoliv objektově orientovaní puristé by se jí zajisté velkým obloukem vyhnuli. Pro přístup k metodám bázové třídy slouží vestavěná funkce `super()`, bez níž bychom se nevyhnuli rekurzi v situaci, kdy potřebujeme zavolat metodu bázové třídy uvnitř reimplementace této metody v podtřídě.

Generátorové funkce a metody provádějí líné vyhodnocování, takže na požádání vracejí (přes příkaz `yield`) vždy jedinou hodnotu z generovaných hodnot, a když (pokud vůbec) tyto hodnoty vyčerpají, vyvolají výjimku `StopIteration`. Generátory lze použít všude tam, kde je očekáván iterátor a pro končené generátory platí, že všechny jejich hodnoty lze extrahovat do n-tice nebo seznamu předáním iterátoru vraceného generátorem funkci `tuple()` nebo `list()`.

Objektově orientovaný přístup vede ve srovnání s čistě procedurálním přístupem téměř vždy k jednoduššímu kódu. Pomocí vlastních tříd můžeme zaručit, že k dispozici jsou pouze platné operace (neboť implementujeme pouze vhodné metody) a že žádná operace nemůže umístit objekt do neplatného stavu (např. aplikací validace prostřednictvím vlastností). Jakmile začneme používat objektovou orientaci, náš styl programování se pravděpodobně změní z globálních datových struktur a globálních funkcí aplikovaných na tato data na vytváření tříd a implementování metod, které jsou pro ně

vhodné. Díky objektové orientaci je možné zabalit dohromady data a metody, které jsou pro tato data smysluplné. To nám pomáhá vyhnout se míchání všech našich dat a funkcí dohromady a usnadňuje vytvářet udržovatelné programy, protože funkčnost je udržována odděleně v jednotlivých třídách.

Cvičení

První dvě cvičení zahrnují úpravu tříd, které jsme probírali v této lekci, a v posledních dvou cvičeních vytvoříte nové dvě třídy úplně od začátku.

1. Upravte třídu Point (ze souboru `Shape.py` nebo `ShapeAlt.py`) tak, aby podporovala následující operace, kde `p`, `q` a `r` jsou objekty typu `Point` a `n` je číslo:

```
p = q + r      # Point.__add__()
p += q         # Point.__iadd__()
p = q - r      # Point.__sub__()
p -= q         # Point.__isub__()
p = q * n      # Point.__mul__()
p *= n         # Point.__imul__()
p = q / n      # Point.__truediv__()
p /= n         # Point.__itruediv__()
p = q // n     # Point.__floordiv__()
p //= n        # Point.__ifloordiv__()
```

   Metody pro místní (in-place) operátory mají včetně řádku `def` čtyři řádky a ostatní metody mají včetně řádku `def` po dvou řádcích a všechny jsou samozřejmě velice podobné a docela jednoduché. S minimálním popisem a dokumentačním testem každé přidávané metodě se celý kód nezvětší více než o 130 řádků. Modelové řešení je k dispozici v souboru `Shape_ans.py`, stejný kód je k dispozici také v souboru `ShapeAlt_ans.py`.

2. Upravte třídu `Image` (v souboru `Image.py`) tak, aby poskytovala metodu `resize(width, height)`. Je-li nová výška nebo šířka menší než aktuální hodnota, je nutné všechny barvy mimo nové hranice vymazat. Má-li výška, resp. šířka, hodnotu `None`, použije se stávající výška, resp. šířka. Na konci nezapomeňte znovu vygenerovat množinu `self.__colors`. Metoda vrátí logickou hodnotu signalizující, zda změny byly či nebyly provedeny. Tuto metodu lze implementovat na méně než 20 řádcích (méně než 35 řádků včetně dokumentačního řetězce a jednoduchého dokumentačního testu). Řešení je k dispozici v souboru `Image_ans.py`.

3. Implementujte třídu `Transaction`, která přijímá částku, datum a měnu (výchozí „EUR“), kurz vůči euru (výchozí `1`) a popis (výchozí `None`). Všechny tyto datové atributy musejí být soukromé. Třída bude dále poskytovat tyto vlastnosti určené pouze pro čtení: `amount` (částka), `date` (datum), `currecny` (měna), `eur_conversion_rate` (kurz vůči euru), `description` (popis) a `eur` (částka vypočítaná jako `amount * eur_conversion_rate`). Tato třída může být implementována přibližně na 60 řádcích včetně několik jednoduchých dokumentačních testů. Modelové řešení pro toto cvičení (a následující) je k dispozici v souboru `Account.py`.

4. Implementuje třídu `Account`, která uchovává číslo účtu, název účtu a seznam objektů typu `Transactions`. Číslo účtu (`number`) by měla být vlastnost určená pouze pro čtení. Název (`name`) by měla být vlastnost s povoleným čtením i zápisem a s příkazem `assert` zajišťujícím, že název bude mít minimálně čtyři znaky. Třída by měla podporovat vestavěnou funkci `len()` (vracející počet transakcí) a měla by poskytovat dvě odvozené vlastnosti: vlastnost `balance` by měla vracet zůstatek v eurech a vlastnost `all_eur` by měla vracet hodnotu `True`, jsou-li všechny transakce v eurech. K dispozici by měly být další tři metody: metoda `apply()` pro aplikaci (přidání) transakce, `save()` a `load()`. Metody `save()` a `load()` by měly používat binární naložený objekt (modul `pickle`) s názvem souboru odpovídajícím číslu účtu s příponou `.acc`. Tyto metody by měly ukládat a načítat číslo účtu, název a všechny transakce. Tuto třídu lze implementovat přibližně na 90 řádcích s několika jednoduchými dokumentačními testy, které zahrnují ukládání a načítání – pro vytvoření vhodného názvu pro dočasný soubor použijte kód ve stylu `name = os.path.join(tempfile.gettempdir(), account_name)` a nezapomeňte po dokončení testů dočasný soubor vymazat. Modelové řešení je k dispozici v souboru `Account.py`.

LEKCE 7

Práce se soubory

V této lekci:

- Zapisování a čtení binárních dat
- Zapisování a analyzování textových souborů
- Zapisování a analyzování souborů XML
- Binární soubory s náhodným přístupem

Většina programů potřebuje informace, jako jsou data nebo stavové údaje, ukládat do souborů a načítat ze souborů. Python nabízí mnoho způsobů, jak takové ukládání a načítání provést. Práci s textovými soubory jsme si již stručně probrali v lekci 3 a na naložené objekty (pickles) jsme se letmo podívali v předchozí lekci. V této lekci se budeme věnovat práci se soubory mnohem podrobněji.

Všechny techniky uvedené v této lekci jsou nezávislé na platformě. To znamená, že soubor uložený pomocí jednoho z ukázkových programů na jedné kombinaci operačního systémů a procesoru lze načíst stejným programem na jiné kombinaci operačního systému a procesoru. A to též může platit i u vašich programů, pokud použijete stejné techniky jako v těchto ukázkových programech.

První tři části této lekce se věnují běžnému ukládání na disk a načítání z disku celých kolekcí dat. V první části si ukážeme, jak toto realizovat pomocí binárního formátu souborů, přičemž v jednom oddílu použijeme naložené objekty (volitelně komprimované) a ve druhém vše provede ručně. Ve druhé části si ukážeme, jak pracovat s textovými soubory. Zapisování textu je jednoduché, ale jeho čtení může být složité, pokud potřebujeme zpracovat netextová data, jako jsou čísla nebo kalendářní data. Ukážeme si dva přístupy k analýze textu, ručně a s použitím regulárních výrazů. Ve třetí části se podíváme na to, jak číst a zapisovat soubory XML. V této části se budeme věnovat zapisování a analýze pomocí stromů elementů, zapisování a analýze pomocí modelu DOM (Document Object Model – objektový model dokument) a ručnímu zapisování a analýze s použitím rozhraní SAX (Simple API for XML – jednoduché rozhraní API pro jazyk XML).

Ve čtvrté části si ukážeme, jak pracovat s binárními soubory s náhodným přístupem. To je užitečné v situaci, kdy má každý datový prvek stejnou velikost a počet těchto prvků je větší, než kolik chceme (nebo můžeme) udržovat v paměti.

Který souborový formát je nejlepší pro uchovávání celých kolekcí – binární, textový, nebo XML? Jaký je nejlepší způsob práce s každým z těchto formátů? Tyto otázky jsou pro jednu definitivní odpověď příliš závislé na kontextu, zejména proto, že každý formát a každý způsob práce s tímto formátem má své klady a zápory. Ukážeme si všechny, což nám pomůže provádět informovaná rozhodnutí na základě konkrétního případu.

Binární formáty jsou obvykle velice rychlé při ukládání a načítání a dokážou být velice kompaktní. Binární data nepotřebují analýzu, protože každý datový typ je uložen pomocí své přirozené reprezentace. Binární data nejsou vhodná pro čtení a úpravu člověkem a bez detailní znalosti použitého formátu není možné vytvářet samostatné nástroje pracující s určitými binárními daty.

Textové formáty mohou číst a upravovat lidé, což usnadňuje zpracování textových souborů samostatnými nástroji nebo jejich změnu pomocí textového editoru. Textové formáty může být obtížné analyzovat a není vždy jednoduché poskytnout srozumitelné chybové zprávy, je-li formát textového souboru narušen (např. nedbalou úpravou).

Formáty XML mohou číst a upravovat lidé, i když většinou mají sklon k upovídanosti a k vytváření obrovských souborů. Podobně jako textové formáty lze také formáty XML zpracovat pomocí samostatných nástrojů. Analýza kódu jazyka XML je jednoduchá (tedy za předpokladu, že ji neprovádíme ručně, ale požijeme analyzátor kódu jazyka XML), přičemž některé analyzátory mají srozumitelné hlášení chyb. Analyzátory XML mohou být pomalé, takže čtení velmi velkých souborů XML může trvat mnohem déle než čtení ekvivalentního binárního či textového souboru. Jazyk XML obsahuje

metadata, jako je kódování znaků (implicitně či explicitně), což v textových souborech často není k dispozici. Díky tomu je kód jazyka XML přenositelnější než textové soubory.

Textové formáty jsou pro koncové uživatele obvykle nejpohodlnější, někdy jsou ale problémy s výkonem takové, že jedinou rozumnou volbou zůstává binární formát. Vždy je však užitečné poskytnout také operace pro import a export v XML, díky kterým je možné zpracovat formát souboru i nástroji třetích stran, aniž bychom museli upustit od použití toho optimálního textového či binárního formátu pro běžné zpracování v našem programu.

V prvních třech částech této lekce budeme používat stejnou kolekci dat: množinu záznamů o leteckých nehodách. Tabulka 7.1 uvádí názvy, datové typy a validační omezení pro záznamy o leteckých nehodách. Na tom, jaká data zpracováváme, ve skutečnosti vůbec nezáleží. Podstatné je, že se naučíme zpracovat základní datové typy včetně řetězců, celých čísel, čísel s pohyblivou řádovou čárkou, logických hodnot a kalendářních dat, protože pokud dokážeme pracovat s těmito typy, pak dokážeme pracovat s jakýmkoli druhem dat.

Tabulka 7.1: Záznam o letecké nehodě

Název	Datový typ	Poznámky
`report_id`	`str`	Minimální délka 8 a bez bílých míst
`date`	`datetime.date`	
`airport`	`str`	Neprázdný a bez znaků nového řádku
`aircraft_id`	`str`	Neprázdný a bez znaků nového řádku
`aircraft_type`	`str`	Neprázdný a bez znaků nového řádku
`pilot_percent_hours_on_type`	`float`	Rozsah `0.0` až `100.0`
`pilot_total_hours`	`int`	Kladné a nenulové
`midair`	`bool`	
`narrative`	`str`	Víceřádkový

Při použití stejné množiny dat se záznamy o leteckých nehodách pro formáty binární, textové a XML lze porovnat různé formáty a kód nezbytný pro jejich zpracování. Tabulka 7.2 nabízí počet řádků kódu pro čtení a zápis každého formátu a celkové souhrny.

Tabulka 7.2: Porovnání způsobů čtení a zápisu souborových formátů se záznamy o leteckých nehodách

Formát	Způsob čtení a zápisu	Řádků kódu pro čtení + zápis	Řádků kódu celkem	Velikost výstupního souboru (~KB)
Binární	Pickle (komprese gzip)	20 + 16 =	36	160
Binární	Pickle	20 + 16 =	36	416
Binární	Manuální (komprese gzip)	60 + 34 =	94	132

Formát	Způsob čtení a zápisu	Řádků kódu pro čtení + zápis	Řádků kódu celkem	Velikost výstupního souboru (~KB)
Binární	Manuální	60 + 34 =	94	356
Holý text	Regex pro čtení a manuální zápis	39 + 28 =	67	436
Holý text	Manuální	53 + 28 =	81	436
XML	Strom elementů	37 + 27 =	64	460
XML	Model DOM	44 + 36 =	80	460
XML	Rozhraní SAX pro čtení a manuální zápis	55 + 37 =	92	464

Velikosti souborů jsou přibližné a jsou založeny na jistém vzorku 596 záznamů leteckých nehod.* Velikosti komprimovaných binárních souborů pro stejná data uložená pod jinými názvy souborů se mohou o několik bajtů lišit, protože název souboru je obsažen v komprimovaných datech a jeho délka se různí. Podobně se malinko liší i velikosti souborů XML, protože některé rutiny pro zápis XML používají entity (`&quot;` for " a `&apos;` pro ') pro uvozovky uvnitř textových dat a jiné nikoli.

V prvních třech částech pracujeme s kódem stejného programu: `convert-incidents.py`. Tento program se používá pro čtení dat o leteckých nehodách v jednom formátu a následný zápis ve formátu jiném. Zde je text nápovědy z konzoly programu:

```
Usage: convert-incidents.py [volby] vstupní_soubor výstupní_soubor

Načte data leteckých nehod ze vstupního souboru a zapíše je do
výstupního souboru. Použité datové formáty závisejí na příponě souboru:
.aix je XML, .ait je text (kódování UTF-8), .aib je binární,
.aip je pickle a .html je HTML (pouze pro výstupní soubor).
Všechny formáty jsou nezávislé na platformě.

Options:
  -h, --help            show this help message and exit
  -f, --force           zapíše výstupní soubor i v případě, že již existuje
                        [default: off]
  -v, --verbose         vypíše výsledky [default: off]
  -r READER, --reader=READER
                        způsob čtení (XML): 'dom', 'd', 'etree', 'e', 'sax',
                        's' čtení (text): 'manual', 'm', 'regex', 'r'
                        [default: etree pro XML, manual pro text]
  -w WRITER, --writer=WRITER
                        způsob zápisu (XML): 'dom', 'd', 'etree', 'e',
                        'manual', 'm' [default: manual]
  -z, --compress        zkomprimuje výstupní soubor .aib/.aip [default: off]
```

* Tato data jsou založena na skutečných datech o leteckých nehodách dostupných z amerického leteckého úřadu FAA (Federal Aviation Administration, *www.faa.gov*).

```
  -t, --test            provede dokumetační testy a ukončí (pro výpis výsledků
                        použijte s -v)
```

Volby jsou ve srovnání s běžnými požadavky trošku složitější, protože koncový uživatel se za normálních okolností nestará o to, který způsob čtení či zápisu se pro určitý formát použije. V realističtější verzi programu by proto volby pro způsob čtení a zápisu neexistovaly a my bychom pro každý formát implementovali pouze jeden z nich. Také volba `test` existuje jen k tomu, aby nám pomohla otestovat kód a v ostré verzi by neměla být přítomna.

Program definuje jednu vlastní výjimku:

```
class IncidentError(Exception): pass
```

Letecké nehody uchováváme v objektech typu `Incident`. Zde je příslušný řádek s příkazem `class` a inicializační metoda:

```
class Incident:

    def __init__(self, report_id, date, airport, aircraft_id,
                 aircraft_type, pilot_percent_hours_on_type,
                 pilot_total_hours, midair, narrative=""):
        assert len(report_id) >= 8 and len(report_id.split()) == 1, \
               "neplatné ID hlášení"
        self.__report_id = report_id
        self.date = date
        self.airport = airport
        self.aircraft_id = aircraft_id
        self.aircraft_type = aircraft_type
        self.pilot_percent_hours_on_type = pilot_percent_hours_on_type
        self.pilot_total_hours = pilot_total_hours
        self.midair = midair
        self.narrative = narrative
```

Při vytváření objektu typu `Incident` kontrolujeme identifikační číslo hlášení, které je dostupné jako vlastnost `report_id` určená pouze pro čtení. Všechny ostatní datové atributy jsou vlastnosti, u nichž je povoleno čtení i zápis. Zde je kupříkladu kód vlastnosti `date`:

```
    @property
    def date(self):
        return self.__date

    @date.setter
    def date(self, date):
        assert isinstance(date, datetime.date), "neplatné datum"
        self.__date = date
```

Všechny ostatní vlastnosti jsou napsané stejným stylem a liší se pouze v detailech příkazů `assert`, takže si je zde ukazovat nebudeme. Používáme příkazy `assert`, a proto náš program selže, pokud

dojde k pokusu o vytvoření objektu typu Incident s neplatnými daty nebo k nastavení některé ze stávajících vlastností s povoleným čtením i zápisem na neplatnou hodnotu. Tento nekompromisní přístup jsme zvolili proto, abychom měli jistotu, že ukládaná a načítaná data jsou vždy platná, a pokud nejsou, tak nechceme, aby program tiše pokračoval dál, ale aby skončil a nahlásil chybu.

Kolekci nehod uchovává třída `IncidentCollection`. Tato třída je odvozena od třídy `dict`, takže díky dědičnosti získáváme spoustu funkcí, mezi něž patří například podpora pro operátor přístupu k prvkům (`[]`) pro získávání, nastavování a mazání záznamů o nehodách. Zde je řádek s příkazem `class` a několik metod třídy:

```
class IncidentCollection(dict):

    def values(self):
        for report_id in self.keys():
            yield self[report_id]

    def items(self):
        for report_id in self.keys():
            yield (report_id, self[report_id])

    def __iter__(self):
        for report_id in sorted(super().keys()):
            yield report_id

    keys = __iter__
```

Inicializační metodu reimplementovat nemusíme, protože metoda `dict.__init__()` nám stačí. Klíči jsou identifikační čísla hlášení a hodnotami objekty typu `Incident`. Metody `values()`, `items()` a `keys()` jsme museli reimplementovat, aby jejich iterátory byly seřazeny podle identifikačních čísel hlášení. To není problém, protože metody `values()` a `items()` procházejí přes klíče vrácené metodou `IncidentCollection.keys()`. Tato metoda (což je jen jiný název pro `IncidentCollection.__iter__()`) zase v seřazeném pořadí prochází přes klíče poskytované metodou bázové třídy `dict.keys()`.

Třída `IncidentCollection` dále obsahuje metody `export()` a `import_()`. (Pro odlišení této metody od vestavěného příkazu `import` používáme koncové podtržítko.) Metoda `export()` přijímá název souboru a volitelně způsob zápisu a příkaz komprese. Na základě názvu souboru a způsobu zápisu předá práci specifičtější metodě, jako je `export_xml_dom()` nebo `export_xml_etree()`. Metoda `import_()` přijímá název souboru a volitelně způsob čtení a pracuje podobně. Metodám pro import, které čtou binární formáty, nemusíme říkat, zda je soubor komprimovaný. Očekáváme od nich, že to samy zjistí a zachovají se vhodným způsobem.

Zapisování a čtení binárních dat

Binární formáty, a to i bez komprese, obvykle zabírají nejméně místa na disku a nejrychleji se ukládají a načítají. Nejjednodušší je použít naložené objekty (pickles), ačkoliv ruční práce s binárními daty by měla vést k nejmenším velikostem souborů.

Naložené objekty s volitelnou kompresí

Upozornění: Naložené objekty nabízejí nejjednodušší přístup k ukládání a načítání dat v programech napsaných v jazyku Python, jak jsme si ale řekli už v předchozí lekci, naložené objekty nemají žádný bezpečnostní mechanizmus (žádné šifrování ani digitální podpis), takže načítání naloženého objektu, který pochází z nedůvěryhodného zdroje, může být nebezpečné. Otázka bezpečnosti je na místě, protože naložené objekty mohou importovat libovolný modul a volat libovolné funkce, takže můžeme obdržet naložený objekt, jehož data byla upravena takovým způsobem, aby například interpret při jeho načtení spustil něco škodlivého. Nicméně naložené objekty jsou často ideální pro práci s ad hoc daty, a to především v programech pro osobní použití.

Při vytváření souborových formátů je obvykle snazší napsat před kódem pro načítání kód pro ukládání, a proto si i my nejdříve ukážeme, jak uložit záznamy o nehodách do naloženého objektu.

```
def export_pickle(self, filename, compress=False):
    fh = None
    try:
        if compress:
            fh = gzip.open(filename, "wb")
        else:
            fh = open(filename, "wb")
        pickle.dump(self, fh, pickle.HIGHEST_PROTOCOL)
        return True
    except (EnvironmentError, pickle.PicklingError) as err:
        print("{0}: chyba při exportu: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
    finally:
        if fh is not None:
            fh.close()
```

Pokud byla požadována komprese, použijeme k otevření souboru funkci `gzip.open()` modulu `gzip`. V opačném případě použijeme vestavěnou funkci `open()`. Při nakládání dat v binárním formátu musíme použít režim `"wb"` („write binary“ – binární zápis). V Pythonu 3.0 a 3.1 představuje konstanta `pickle.HIGHEST_PROTOCOL` protokol 3, což je kompaktní binární formát pro naložený objekt. Jedná se o nejlepší protokol pro data sdílená mezi programy napsanými v Pythonu 3.*

Co se zpracování chyb týče, rozhodli jsme se je hlásit uživateli ihned, jakmile k nim dojde, a vracet volajícímu logickou hodnotu signalizující úspěch či neúspěch. Pro zajištění, aby se na konci soubor uzavřel bez ohledu na to, zda došlo či nedošlo k chybě, používáme blok `finally`. V lekci 8 použijeme pro zajištění, aby se soubory uzavřely, kompaktnější styl, který nepotřebuje blok `finally`.

Správci kontextu ➤ 357

* Protokol 3 je specifický pro Python 3. Pokud chceme naložené objekty, které budou čitelné a zapisovatelné pro Python 2 i Python 3, pak musíme použít protokol 2. Je však třeba poznamenat, že soubory využívající protokol 2 zapsané Pythonem 3.1 lze číst Pythonem 3.1 a Pythonem 2.x, ale ne Pythonem 3.0!

3.x

Tento kód je velice podobný tomu, který jsme viděli v předchozí lekci, je zde ale jedno zákeřné místo, na které je nutné upozornit. Naloženými daty je objekt `self`, který je typu `dict`, avšak hodnotami tohoto slovníku jsou objekty typu `Incident`, což jsou instance naší vlastní třídy. Modul `pickle` je dostatečně chytrý, aby byl schopen ukládat objekty většiny vlastních tříd bez nutnosti našeho zásahu.

`__dict__` ➤ 352

Nakládat (vytvářet naložené objekty) lze logické hodnoty, čísla a řetězce stejně jako instance tříd včetně vlastních tříd, ovšem za předpokladu, že jejich soukromý atribut `__dict__` je možné nakládat. Kromě toho lze nakládat jakékoli vestavěné typy představující kolekce (n-tice, seznamy, množiny, slovníky), ovšem za předpokladu, že obsahují pouze objekty, které je možné nakládat (což zahrnuje také typy představující kolekce, takže rekurzivní struktury jsou podporovány). Dále je možné nakládat i ty druhy objektů či instancí vlastních tříd, které není možné normálně naložit (např. proto, že obsahují atribut, který nelze naložit). To lze provést buď tak, že modulu `pickle` poskytneme určitou nápovědu, nebo tak, že implementujeme vlastní funkce pro naložení a vytažení. Všechny podrobnosti najdete ve webové dokumentaci k modulu `pickle`.

Pro opětovné načtení naložených dat je nutné rozlišit mezi komprimovaným a nekomprimovaným naloženým objektem. Jakýkoliv soubor, který je zkomprimovaný pomocí komprese gzip, začíná určitým *magickým číslem*. Magické číslo je posloupnost jednoho či více bajtů na začátku souboru, která se používá k označení typu souboru. U souborů gzip má toto číslo dva bajty 0x1F 0x8B, které uchováváme v proměnné typu `bytes`:

```
GZIP_MAGIC = b"\x1F\x8B"
```

Více informací o datovém typu `bytes` najdete v panelu „Datové typy bytes a bytearray“ a v tabulce 7.3 (na straně 289), které uvádějí jejich metody.

Datové typy bytes a bytearray

Python nabízí dva datové typy pro práci s holými bajty: typ `bytes`, který je neměnitelný, a typ `bytearray`, který je měnitelný. Oba typy uchovávají posloupnost nula či více 8bitových bezznaménkových celých čísel (bajtů), kde každý bajt uchovává hodnotu v rozsahu 0 až 255.

`str.translate()` ➤ 82

Oba typy jsou velice podobné řetězcům a poskytují řadu stejných metod, včetně podpory pro řezání. Kromě toho objekty typu `bytearray` nabízejí metody podobné metodám seznamu pro změnu prvků. Všechny metody jsou uvedené v tabulce 7.3 (strana 289) a 7.4 (strana 317).

Zatímco řez aplikovaný na objekt typu `bytes` nebo `bytearray` vrací objekt stejného typu, přístup k jedinému bajtu pomocí operátoru pro přístup k prvkům (`[]`) vrací objekt typu `int` obsahující hodnotu daného bajtu. Například:

```
word = b"Animal"
x = b"A"
word[0] == x      # vrátí: False   # word[0] == 65; x == b"A"
word[:1] == x     # vrátí: True    # word[:1] == b"A"; x == b"A"
word[0] == x[0]   # vrátí: True    # word[0] == 65; x[0] == 65
```

Zde je několik dalších ukázek použití typů `bytes` a `bytearray`:

```
data = b"5 Hills \x35\x20\x48\x69\x6C\x6C\x73"
data.upper()                          # vrátí: b'5 HILLS 5 HILLS'
data.replace(b"ill", b"at")           # vrátí: b'5 Hats 5 Hats'
bytes.fromhex("35 20 48 69 6C 6C 73")  # vrátí: b'5 Hills'
bytes.fromhex("352048696C6C73")       # vrátí: b'5 Hills'
data = bytearray(data)                # data jsou nyní typu bytearray
data.pop(10)                          # vrátí: 72 (ord("H"))
data.insert(10, ord("B"))             # data == b'5 Hills 5 Bills'
```

Metody, které mají smysl jen u řetězců, jako je `bytes.upper()`, předpokládají, že bajty jsou zakódovány pomocí znakové sady ASCII. Metoda třídy `bytes.fromhex()` ignoruje bílé místo a interpretuje každý podřetězec se dvěma číslicemi jako šestnáctkové číslo, takže například z `"35"` se stane bajt s hodnotou `0x35`.

Zde je kód pro čtení souboru s naloženými záznamy nehod:

```
def import_pickle(self, filename):
    fh = None
    try:
        fh = open(filename, "rb")
        magic = fh.read(len(GZIP_MAGIC))
        if magic == GZIP_MAGIC:
            fh.close()
            fh = gzip.open(filename, "rb")
        else:
            fh.seek(0)
        self.clear()
        self.update(pickle.load(fh))
        return True
    except (EnvironmentError, pickle.UnpicklingError) as err:
        print("{0}: chyba při importu: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
    finally:
        if fh is not None:
            fh.close()
```

Nevíme, zda je zadaný soubor komprimován. V každém případě začínáme jeho otevřením v režimu „read binary“ (binární čtení) a poté přečteme první dva bajty. Jsou-li tyto bajty stejné jako magické číslo formátu gzip, soubor zavřeme a vytvoříme nový objekt souboru pomocí funkce `gzip.open()`. A pokud soubor není komprimován, použijeme objekt souboru vrácený funkcí `open()` a voláním jeho metody `seek()` vrátíme ukazatel souboru na začátek, takže následující čtení (provedené uvnitř funkce `pickle.load()`) začne od začátku souboru.

Do objektu `self` přiřazovat nemůžeme, protože tím bychom zničili objekt typu `IncidentCollection`, který se právě používá. Proto raději vymažeme všechny záznamy o nehodách (tím slovník vyprázdníme) a poté slovník znovu metodou `dict.update()` naplníme vše záznamy o nehodách ze slovníku typu `IncidentCollection` načteného z naloženého objektu.

Všimněte si, že vůbec nezáleží na tom, zda proces zpracovává bajty v pořadí Big Endian nebo Little Endian, protože u magického čísla čteme jednotlivé bajty a u dat se o aktuální pořadí bajtů postará modul `pickle`.

Holá binární data s volitelnou kompresí

Napsáním vlastního kódu pro práci s holými binárními daty získáme úplnou kontrolu nad formátem souboru. Tento postup je také bezpečnější než používání naložených objektů, protože zlovolná neplatná data nespustí interpret, ale budou zpracována naším kódem.

Při vytváření vlastního binárního souborového formátu je rozumné vytvořit si magické číslo, které bude identifikovat váš typ souboru, a číslo verze, které bude identifikovat verzi používaného souborového formátu. Zde jsou definice použité v programu `convert-incidents.py`:

```
MAGIC = b"AIB\x00"
FORMAT_VERSION = b"\x00\x01"
```

Pro magické číslo jsme použili čtyři bajty a pro číslo verze dva bajty. Způsob uložení bajtů není problém, protože je budeme zapisovat po jednotlivých bajtech, a ne jako bajtovou reprezentaci celých čísel, takže pořadí bajtů budou vždy stejné na libovolné procesorové architektuře.

Pro zápis a čtení holých binárních dat musíme mít nějaké prostředky pro převod objektů Pythonu na vhodné binární reprezentace zpět. Většinu potřebné funkčnosti zajišťuje modul `struct`, který je stručně popsán v panelu „Modul struct“, a datové typy `bytes` a `bytearray`, jejichž stručný popis nabízí panel „Datové typy bytes a bytearray“ (strana 286). Metody tříd `bytes` a `bytearray` jsou uvedeny v tabulce 7.3 (na straně 289).

Modul struct

Modul `struct` nabízí funkce `struct.pack()`, `struct.unpack()` a několik dalších a třídu `struct.Struct()`. Funkce `struct.pack()` přijímá formátovací řetězec modulu `struct` a jednu či více hodnot a vrací objekt typu `bytes`, který uchovává všechny tyto hodnoty reprezentované podle zadaného formátu. Funkce `struct.unpack()` přijímá formát a objekt typu `bytes` nebo `bytearray` a vrací n-tici hodnot, které byly původně zabaleny funkcí `struct.pack()` s použitím zadaného formátu. Například:

```
data = struct.pack("<2h", 11, -9)    # data == b'\x0b\x00\xf7\xff'
items = struct.unpack("<2h", data)   # items == (11, -9)
```

Formátovací řetězce se skládají z jednoho či více znaků. Většina znaků představuje hodnotu určitého typu. Pokud potřebujeme více než jednu hodnotu nějakého typu, pak můžeme buď zapsat daný znak tolikrát, kolik máme hodnot s tímto typem (`"hh"`), nebo před znak s označením typu umístit počet, jak jsme učinili ve výše uvedeném příkladu (`"2h"`).

Spousta formátovacích znaků je popsána ve webové nápovědě modulu `struct`. Mezi tyto znaky patří „b“ (8bitové celé číslo se znaménkem), „B“ (8bitové celé číslo bez znaménka), „h“ (16bitové celé číslo se znaménkem – použili jsme je ve výše uvedeném příkladu), „H“ (16bitové celé číslo bez znaménka), „i“ (32bitové celé číslo se znaménkem), „I“ (32bitové celé číslo bez znaménka), „q“ (64bitové celé číslo se znaménkem), „Q“ (64bitové celé číslo bez znaménka), „f“ (32bitové číslo s pohyblivou řádovou čárkou), „d“ (64bitové číslo s pohyblivou řádovou čárkou – odpovídá typu `float` v jazyku Pythonu), „?“ (logická hodnota), „s“ (objekt typu `bytes` nebo `bytearray` – řetězce bajtů) a řada dalších.

Pro některé datové typy, jako jsou vícebajtová celá čísla, má na pořadí bajtů vliv způsob uložení bajtů v procesoru. Vynucení určitého pořadí bajtů bez ohledu na architekturu procesoru provedeme zahájením formátovacího řetězce znakem představujícím požadovaný způsob uložení bajtů. V této knize budeme vždy používat znak „<“, který představuje formát Little Endian, což je nativní způsob uložení bajtů pro široce rozšířené procesory Intel a AMD. Formát Big Endian (označovaný též jako síťové pořadí bajtů) představuje znak „>“ (nebo „!“). Pokud není uveden žádný způsob uložení bajtů, použije se takový, který se používá na daném stroji. Doporučujeme vždy uvádět způsob uložení bajtů, a to i tehdy, je-li stejné jako na používaném stroji, protože díky tomu budou vaše data přenositelná.

Funkce `struct.calcsize()` přijímá formát a vrací počet bajtů, které bude daný formát zabírat. Formát lze také uložit vytvořením objektu typu `struct.Struct()`, kterému předáme formát jako argument, přičemž velikost toho objektu je obsažena v jeho atributu `size`. Například:

```
TWO_SHORTS = struct.Struct("<2h")
data = TWO_SHORTS.pack(11, -9)   # data == b'\x0b\x00\xf7\xff'
items = TWO_SHORTS.unpack(data)  # items == (11, -9)
```

V obou příkladech číslo `11` představuje `0x000b`, což je ale transformováno na bajty `0x0b 0x00`, protože pro řazení bajtů používáme formát Little Endian.

Tabulka 7.3: Metody tříd bytes a bytearray

Syntaxe	Popis
ba.append(i)	Připojí číslo `i` typu `int` (v rozsahu 0 až 255) do objektu ba typu `bytearray`.
b.capitalize()	Vrátí kopii objektu b typu `bytes` nebo `bytearray` s prvním znakem převedeným na velké písmeno (jedná-li se o písmeno znakové sady ASCII).
b.center(*šířka*, *bajt*)	Vrátí kopii objektu `b` vycentrovanou na zadanou šířku a doplněnou mezerami nebo volitelně zadaným bajtem.
b.count(*x*, *začátek*, *konec*)	Vrátí počet výskytů objektu *x* typu `bytes` nebo `bytearray` v objektu *b* typu `bytes` nebo `bytearray` (nebo v řezu *začátek*:*konec* objektu *b*).
b.decode(*kódování*, *chyby*)	Vrátí objekt typu `str`, který reprezentuje bajty pomocí kódování UTF-8 nebo pomocí zadaného *kódování*, přičemž chyby se zpracují podle volitelného argumentu *chyby*.

Kódování znaků ➤ 95

Syntaxe	Popis
b.endswith(*x*, *začátek*, *konec*)	Vrátí hodnotu `True`, pokud objekt *b* (nebo řez *začátek*:*konec* objektu *b*) končí objektem *x* typu `bytes` nebo `bytearray` nebo kterýmkoli z objektů typu `bytes` nebo `bytearray` v n-tici *x*. V opačném případě vrátí hodnotu `False`.
b.expandtabs(*počet*)	Vrátí kopii objektu *b* typu `bytes` nebo `bytearray` s tabulátory nahrazenými vždy 8 mezerami nebo volitelně zadaným počtem mezer.
ba.extend(*posloupnost*)	Rozšíří objekt *ba* typu `bytearray` o čísla typu `int` v zadané posloupnosti. Všechna tato čísla musejí být v rozsahu 0 až 255.
b.find(*x*, *začátek*, *konec*)	Vrátí nejlevější pozici objektu *x* typu `bytes` nebo `bytearray` v objektu *b* (nebo v řezu *začátek*:*konec* objektu *b*) nebo -1, pokud není nalezen. Pro nalezení nejpravější pozice se používá metoda `rfind()`.
b.fromhex(*h*)	Vrátí objekt typu `bytes` s bajty odpovídajícími šestnáctkovým číslům v řetězci *h*.
b.index(*x*, *začátek*, *konec*)	Vrátí nejlevější pozici objektu *x* typu `bytes` nebo `bytearray` v objektu *b* (nebo v řezu *začátek*:*konec* objektu *b*) nebo vyvolá výjimku `ValueError`, pokud není nalezen. Pro nalezení nejpravější pozice se používá metoda `rindex()`.
ba.insert(*p*, *i*)	Vloží celé číslo *i* (v rozsahu 0 až 255) na pozici *p* v objektu `ba`.
b.isalnum()	Vrátí hodnotu `True`, pokud je objekt *b* typu `bytes` nebo `bytearray` neprázdný a pokud je každý znak v *b* alfanumerickým znakem znakové sady ASCII.
b.isalpha()	Vrátí hodnotu `True`, pokud je objekt *b* typu `bytes` nebo `bytearray` neprázdný a pokud je každý znak v *b* abecedním znakem znakové sady ASCII.
b.isdigit()	Vrátí hodnotu `True`, pokud je objekt *b* typu `bytes` nebo `bytearray` neprázdný a pokud je každý znak v *b* celým číslem znakové sady ASCII.
b.islower()	Vrátí hodnotu `True`, pokud objekt *b* typu `bytes` nebo `bytearray` má alespoň jeden znak znakové sady ASCII převoditelný na malé písmeno a pokud jsou všechny takto převoditelné znaky malými písmeny.
b.isspace()	Vrátí hodnotu `True`, pokud je objekt *b* typu `bytes` nebo `bytearray` neprázdný a pokud je každý znak v *b* znakem znakové sady ASCII představujícím bílé místo.
b.istitle()	Vrátí hodnotu `True`, je-li objekt b neprázdný, každé slovo začíná velkým písmenem a všechna ostatní jsou malá.
b.isupper()	Vrátí hodnotu `True`, pokud objekt *b* typu `bytes` nebo `bytearray` má alespoň jeden znak znakové sady ASCII převoditelný na velké písmeno a pokud jsou všechny takto převoditelné znaky velkými písmeny.

Syntaxe	Popis
b.join(*posloupnost*)	Vrátí spojení každého objektu typu `bytes` nebo `bytearray` v zadané posloupnosti s tím, že mezi každý umístí objekt `b` (který může být prázdný).
b.ljust(šířka, bajt)	Vrátí kopii objektu `b` zarovnanou doleva na zadanou šířku doplněnou mezerami nebo volitelně zadaným bajtem. Pro zarovnání doprava se používá metoda `rjust()`.
b.lower()	Vrátí kopii objektu b typu `bytes` nebo `bytearray` se všemi znaky ASCII převedenými na malá písmena.
b.partition(*oddělovač*)	Vrátí n-tici tří objektů typu `bytes`: část `b` před nejlevějším výskytem objektu *oddělovač* typu `bytes` nebo `bytearray`, samotný objekt *oddělovač* a část objektu `b` za objektem *oddělovač*. Pokud *oddělovač* není v objektu `b`, pak vrátí `b` a dva prázdné objekty typu `bytes`. Pro vyhledání nejpravějšího výskytu oddělovače se používá metoda `rpartition()`.
ba.pop(*p*)	Odstraní a vrátí celé číslo na indexové pozici *p* v objektu `ba`.
ba.remove(*i*)	Odstraní první výskyt čísla typu `int` z objektu `ba` typu `bytearray`.
b.replace(*x*, *y*, *n*)	Vrátí kopii objektu `b` s každým výskytem (nebo maximální s *n* výskyty, je-li n zadáno) objektu *x* typu `bytes` nebo `bytearray` nahrazeným objektem *y*.
ba.reverse()	Obrátí na místě pořadí bajtů objektu `ba` typu `bytearray`.
b.split(*x*, *n*)	Vrátí seznam bajtů rozdělených nejvýše *n*-krát podle *x*. Není-li *n* zadáno, pak dělení probíhá v maximálním možné míře. Není-li zadáno `t`, pak se řetězec rozdělí podle mezer. Není-li zadáno *x*, probíhá dělení podle bílých míst. Pro dělení zprava se používá metoda`rsplit()`.
b.splitlines(*f*)	Vrátí seznam řádků, který je výsledkem rozdělení `b` podle oddělovačů řádků. Nemá-li `f` hodnotu `True`, pak se oddělovače řádků odstraní.
b.startswith(*x*, *začátek*, *konec*)	Vrátí hodnotu `True`, pokud objekt `b` typu `bytes` nebo `bytearray` (nebo jeho řez *začátek*`:`*konec*) začíná objektem *x* typu `bytes` nebo `bytearray` nebo některým z objektů typu `bytes` nebo `bytearray` v n-tici *x*.
b.strip(*x*)	Vrátí kopii `b` bez počátečního a koncového bílého znaku (nebo bez bajtů v objektu *x* typu `bytes` nebo `bytearray`). Metoda `lstring()` odstraňuje znaky pouze na začátku a metoda `rstrip()` pouze na konci.
b.swapcase()	Vrátí kopii `b`, která má místo malých písmen znakové sady ASCII písmena velká a místo velkých písmen znakové sady ASCII malá.

Syntaxe	Popis
b.title()	Vrátí kopii b, v níž je první písmeno znakové sady ASCII každého slova velké a všechna ostatní písmena znakové sady ASCII jsou malá.
b.translate(*bt*, *d*)	Vrátí kopii b, která nemá žádné bajty z d a kde je každý další bajt nahrazen bajtem z objektu bt typu bytes na pozici určené nahrazovaným bajtem.
b.upper()	Vrátí kopii objektu b typu bytes nebo bytearray se všemi znaky ASCII převedenými na velká písmena.
b.zfill(*w*)	Vrátí kopii b, která, je-li kratší než *w*, je na začátku doplněna nulami (znaky 0x30) na délku *w* bajtů.

Modul `struct` naneštěstí dokáže pracovat pouze s řetězci zadané délky, my však potřebujeme pro identifikační čísla hlášení a letadla řetězce proměnné délky, které jsou nezbytné také pro letiště, typ letadla a popis. K tomuto účelu jsme vytvořili funkci `pack_string()`, která přijímá řetězec a vrací objekt typu `bytes`, který obsahuje dvě složky. První složkou je celé číslo představující délku a druhou je posloupnost bajtů o této délce, které jsou zakódovány pomocí UTF-8 a které představují text řetězce.

Lokální funkce ➤ 341

Jediným místem, kde funkci `pack_string()` potřebujeme, je funkce `export_binary()`, a proto jsme její definici umístili přímo do funkce `export_binary()`. To znamená, že funkce `pack_string()` není mimo funkci `export_binary()` viditelná, a je jasné, že se jedná pouze o lokální pomocnou funkci. Zde je začátek funkce `export_binary()` a kompletní vnořená funkce `pack_string()`:

```
def export_binary(self, filename, compress=False):


    def pack_string(string):
        data = string.encode("utf8")
        format = "<H{0}s".format(len(data))
        return struct.pack(format, len(data), data)
```

Kódování znaků ➤ 95

Metoda `str.encode()` vrací objekt typu `bytes` s řetězcem zakódovaným podle zadaného kódování. UTF-8 je skutečně vhodné kódování, protože dokáže reprezentovat libovolný znak znakové sady Unicode a při reprezentaci znaků ASCII je zvláště kompaktní (pro každý jen jeden znak). Proměnná `format` uchovává formát pro modul `struct` vytvořený podle délky řetězce. Například při zadání řetězce „cs.wikipedia.org“ bude formát `"<H16s"` (pořadí bajtů Little Endian, celé číslo bez znaménka délky 2 bajty, bajtový řetězec délky 16 bajtů) a vrácený objekt bude `b'\x10\x00cs.wikipedia.org'`. Python standardně zobrazuje objekty typu `bytes` v kompaktní formě pomocí tisknutelných znaků ASCII a ostatní znaky zobrazí pomocí zakódovaných šestnáctkových hodnot.

Funkce `pack_string()` dokáže zpracovat řetězce až do délky 65 535 znaků v kódování UTF-8. Pro počet bajtů lze snadno použít i jiný druh celého čísla, například 4bajtové celé číslo se znaménkem (formátovací znak „i“) nám umožňuje zpracovat řetězce do délky 2^{31}-1 znaků (tj. více než 2 miliardy znaků).

Modul `struct` nabízí podobný vestavěný formát „p“, který uchovává jediný bajt jako počet znaků následovaný maximálně 255 znaky. Pro zabalování je kód používající formát „p“ trošku jednodušší při porovnání s tím, když si vše uděláme sami. Avšak formát „p“ nám omezuje na maximálně 255 znaků v kódování UTF-8 a při rozbalování nenabízí téměř žádnou výhodu. (Pro srovnání jsou ve zdrojovém souboru `convert-incidents.py` obsaženy verze funkcí `pack_string()` a `unpack_string()`, které používají formát „p“.)

Nyní můžeme pozornost obrátit ke zbývající části kódu metody `export_binary()`.

```
        fh = None
        try:
            if compress:
                fh = gzip.open(filename, "wb")
            else:
                fh = open(filename, "wb")
            fh.write(MAGIC)
            fh.write(FORMAT_VERSION)
            for incident in self.values():
                data = bytearray()
                data.extend(pack_string(incident.report_id))
                data.extend(pack_string(incident.airport))
                data.extend(pack_string(incident.aircraft_id))
                data.extend(pack_string(incident.aircraft_type))
                data.extend(pack_string(incident.narrative.strip()))
                data.extend(NumbersStruct.pack(
                                incident.date.toordinal(),
                                incident.pilot_percent_hours_on_type,
                                incident.pilot_total_hours,
                                incident.midair))
                fh.write(data)
            return True
```

Bloky `exept` a `finally` jsme vynechali, protože jsou až na konkrétní typy výjimek zachytávaných v bloku `except` stejné jako ty, které jsme si ukázali v předchozím oddílu.

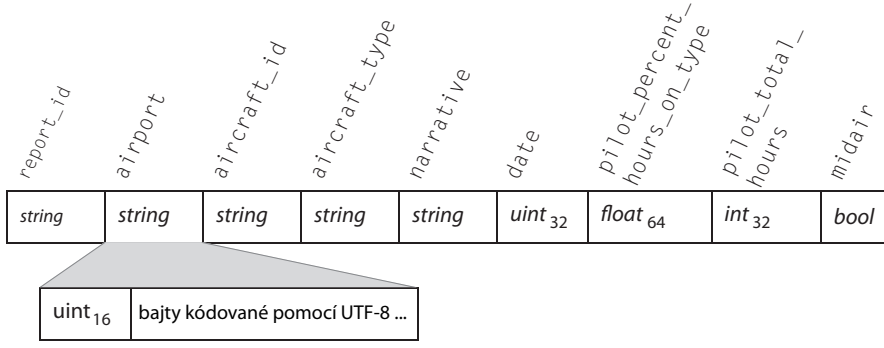
Začínáme otevřením souboru v režimu binárního zápisu, a to v závislosti na příznaku `compress` buď obyčejného souboru, nebo souboru s kompresí gzip. Poté zapíšeme 4bajtové magické číslo, které je (jak doufáme) jedinečné pro náš program, a 2bajtové číslo verze.* Použití čísla verze nám usnadní změnu formátu v budoucnu. Při načtení čísla verze můžeme určit, který kód se má použít pro čtení dat.

Dále procházíme všechny záznamy o nehodách a pro každý z nich vytváříme objekt typu `bytearray`. Každý prvek dat počínaje řetězci s proměnnou délkou přidáváme do pole bajtů. Metoda `date.toordinal()` vrací jediné číslo reprezentující uložené datum, které lze později obnovit předáním tohoto čísla metodě `datetime.date.fromordinal()`. Objekt `NumbersStruct` definujeme v dřívější části programu následujícím příkazem:

```
NumbersStruct = struct.Struct("<Idi?")
```

* Tato magická čísla na rozdíl od názvů domén žádný centrální repositář nemají, takže jejich jedinečnost nemůžeme nikdy zcela zaručit.

Tento formát stanoví pořadí bajtů Little Endian, 32bitové celé číslo bez znaménka (pro číselnou hodnotu kalendářního data), 64bitové číslo s pohyblivou řádovou čárkou (pro procentní podíl nalétaných hodin na daném typu), 32bitové celé číslo se znaménkem (pro celkový počet nalétaných hodin) a logickou hodnotu (signalizující, zda k nehodě došlo ve vzduchu). Strukturu celého záznamu o letecké nehodě schematicky znázorňuje obrázek 7.1.



Obrázek 7.1: Struktura binárního záznamu o letecké nehodě

Jakmile má objekt typu `bytearray` všechna data pro jednu nehodu, zapíšeme je na disk. A jakmile zapíšeme na disk všechny nehody, vrátíme hodnotu `True` (za předpokladu, že nedojde k chybě). Blok `finally` zajistí, aby se soubor před vrácením hodnoty uzavřel. Čtení dat není tak jednoduché jako jejich zápis, už jen kvůli tomu, že může dojít k více chybám, které musíme ošetřit. Kromě toho čtení řetězců s proměnnou délkou je malinko záludné. Zde je začátek metody `import_binary()` a kompletní vnořená funkce `unpack_string()`, kterou používáme pro čtení řetězců proměnné délky:

```
def import_binary(self, filename):

    def unpack_string(fh, eof_is_error=True):
        uint16 = struct.Struct("<H")
        length_data = fh.read(uint16.size)
        if not length_data:
            if eof_is_error:
                raise ValueError("missing or corrupt string size")
            return None
        length = uint16.unpack(length_data)[0]
        if length == 0:
            return ""
        data = fh.read(length)
        if not data or len(data) != length:
            raise ValueError("chybějící nebo porušený řetězec")
        format = "<{0}s".format(length)
        return struct.unpack(format, data)[0].decode("utf8")
```

Každý záznam o nehodě začíná řetězcem s identifikačním číslem jeho hlášení, takže se po úspěšném pokusu o přečtení tohoto řetězce nacházíme na začátku nového záznamu. Pokud ale čtení není úspěšné, tak jsme dorazili na konec souboru a můžeme skončit. Při pokusu o čtení identifikačního čísla hlášení nastavujeme příznak `eof_is_error` na hodnotu `False`, protože pokud nejsou k dispozici žádná data, tak to prostě znamená, že jsme skončili. U všech ostatních řetězců ponecháváme výchozí hodnotu `True`, protože nemá-li kterýkoli z ostatních řetězců žádná data, znamená to chybu. (Dokonce i prázdný řetězec bude předcházet 16bitové celé číslo bez znaménka představující délku.)

Začínáme pokusem o přečtení délky řetězce. Pokud dojde k chybě, vrátíme hodnotu `None` signalizující konec souboru (pokud se pokoušíme přečíst nový záznam o nehodě) nebo vyvoláme výjimku `ValueError` signalizující poškozená či chybějící data. Funkce `struct.unpack()` a metoda `struct.Struct.unpack()` vždy vracejí n-tici, a to i tehdy, obsahuje-li pouze jedinou hodnotu. Rozbalíme data s délkou a číslo, které reprezentují, uložíme do proměnné `length`. Je-li délka nulová, pak jednoduše vrátíme prázdný řetězec. V opačném případě se pokusíme přečíst uvedený počet bajtů. Pokud neobdržíme žádná data nebo pokud tato data nemají očekávanou délku (tj. jsou příliš malá), vyvoláme výjimku `ValueError`.

Pokud máme správný počet bajtů, můžeme pro funkci `struct.unpack()` vytvořit vhodný formátovací řetězec a poté vrátit řetězec, který je výsledkem rozbalení dat a dekódování bajtů v kódování UTF-8. (Teoreticky bychom mohli poslední dva řádky nahradit příkazem `return data.decode("utf8")`, my si ale raději projdeme proces rozbalení, protože je možné, i když nepravděpodobné, že formát „s“ provede na našich datech nějaké transformace, které je nutné při opětovném čtení provést obráceně.)

Nyní se podíváme na zbytek metody `import_binary()`, kterou si pro snazší výklad rozdělíme na dvě části.

```
fh = None
try:
    fh = open(filename, "rb")
    magic = fh.read(len(GZIP_MAGIC))
    if magic == GZIP_MAGIC:
        fh.close()
        fh = gzip.open(filename, "rb")
    else:
        fh.seek(0)
    magic = fh.read(len(MAGIC))
    if magic != MAGIC:
        raise ValueError("neplatný formát souboru .aib")
    version = fh.read(len(FORMAT_VERSION))
    if version > FORMAT_VERSION:
        raise ValueError("neznámý formát souboru .aib")
    self.clear()
```

Soubor může, ale také nemusí být zkomprimován, takže k otevření souboru funkcí `gzip.open()` nebo vestavěnou funkcí `open()` sáhneme po stejné technice, kterou jsme použili pro čtení naložených objektů.

Jakmile je soubor otevřený a my se nacházíme na jeho začátku, tak přečteme první čtyři bajty (`len(MAGIC)`). Pokud nesouhlasí s naším magickým číslem, pak víme, že se nejedná o binární soubor s daty záznamů o nehodách, a proto vyvoláme výjimku `ValueError`. Dále načteme 2bajtové číslo verze. Na tomto místě bychom mohli použít odlišný kód pro čtení v závislosti na této verzi. Zde jen zkontrolujeme, zda se nejedná o pozdější verzi, než kterou dokáže náš program přečíst.

Pokud je magické číslo správné a uvedenou verzi dokážeme zpracovat, jsme připraveni přečíst data, a proto začneme vymazáním stávajících záznamů o nehodách, aby byl slovník prázdný.

```
while True:
    report_id = unpack_string(fh, False)
    if report_id is None:
        break
    data = {}
    data["report_id"] = report_id
    for name in ("airport", "aircraft_id",
                 "aircraft_type", "narrative"):
        data[name] = unpack_string(fh)
    other_data = fh.read(NumbersStruct.size)
    numbers = NumbersStruct.unpack(other_data)
    data["date"] = datetime.date.fromordinal(numbers[0])
    data["pilot_percent_hours_on_type"] = numbers[1]
    data["pilot_total_hours"] = numbers[2]
    data["midair"] = numbers[3]
    incident = Incident(**data)
    self[incident.report_id] = incident
return True
```

Cyklus `while` probíhá stále dokola, dokud nezpracujeme všechna data. Začínáme pokusem o získání identifikačního čísla hlášení. Pokud obdržíme hodnotu `None`, pak víme, že jsme na konci souboru, a proto cyklus přerušíme. V opačném případě vytvoříme slovník s názvem `data`, který bude uchovávat data pro jednu nehodu, a pokusíme se získat zbývající údaje o nehodě. Pro řetězce použijeme metodu `unpack_string()` a ostatní data načteme najednou pomocí objektu `NumbersStruct`. Datum jsme uložili jako číslici, a proto musíme pro jeho opětovné získání provést převod obráceným směrem. Pro ostatní prvky nám ale stačí použít rozbalená data přímo, přičemž nemusíme provádět žádnou validaci ani převod, protože jsme zapsali správné datové typy a pomocí formátu uloženého v objektu `NumbersStruct` jsme načetli stejné datové typy.

Pokud dojde k nějakým chybám (nebude-li například možné rozbalit všechna čísla), pak vyvoláme výjimku, kterou zachytíme v bloku `except`. (Bloky `except` a `finally` jsme si zde neukázali, protože jsou co do struktury stejné, jako ty, které jsme si ukázali v předchozím oddílu u metody `import_pickle()`.)

Rozbalení mapování ➤ 177

U konce výše uvedeného úryvku kódu používáme pro vytvoření objektu typu Incident standardní syntaxi pro rozbalení mapování a tento objekt pak uložíme do slovníku se záznamy nehod.

Kromě zpracování řetězců proměnné délky je díky modulu `struct` ukládání a načítání dat v binárním formátu velice jednoduché. A pro řetězce proměnné délky jsou pro většinu případů naprosto dostatečné výše uvedené metody `pack_string()` a `unpack_string()`.

Zapisování a analyzování textových souborů

Zapisování textu je jednoduché, jeho opětovné čtení ale může být docela problematické, a proto je nutné pečlivě zvolit strukturu, díky které nebude analýza textu tak obtížná.* Na obrázku 7.2 je ukázka záznamu letecké nehody v textovém formátu, který budeme používat. Při zápisu záznamů o nehodě do souboru budeme za každý záznam přidávat prázdný řádek, avšak při analýze souboru budeme předpokládat, že mezi záznamy může být nula či více prázdných řádků.

```
[20070927022009C]
date=2007-09-27
aircraft_id=1675B
aircraft_type=DHC-2-MK1
airport=MERLE K (MUDHOLE) SMITH
pilot_percent_hours_on_type=46.1538461538
pilot_total_hours=13000
midair=0
.NARRATIVE_START.
    ACCORDING TO THE PILOT, THE DRAG LINK FAILED DUE TO AN OVERSIZED
    TAIL WHEEL TIRE LANDING ON HARD SURFACE.
.NARRATIVE_END.
```

Obrázek 7.2: Ukázka textového formátu pro záznam o letecké nehodě

Zapisování textu

Každý záznam o nehodě začíná identifikačním číslem hlášení v hranatých závorkách (`[]`). Za ním následují všechny jednořádkové datové prvky zapsané ve tvaru *`klíč=hodnota`*. Před víceřádkový text popisu umísťujeme značku začátku (`.NARRATIVE_START.`) a na jeho konec značku konce (`.NARRATIVE_END.`), přičemž veškerý text uvnitř těchto značek odsadíme, abychom měli jistotu, že se žádný řádek textu nebude plést se značkou začátku či konce.

Zde je kód pro funkci `export_text()`, ovšem bez bloků `except` a `finally`, které jsou stejné jako ty, které jsme viděli výše, samozřejmě tedy kromě zpracovávaných výjimek:

```
def export_text(self, filename):
    wrapper = textwrap.TextWrapper(initial_indent="    ",
                                   subsequent_indent="    ")
    fh = None
    try:
        fh = open(filename, "w", encoding="utf8")
        for incident in self.values():
            narrative = "\n".join(wrapper.wrap(
                                incident.narrative.strip()))
            fh.write("[{0.report_id}]\n"
```

* V lekci 14 si představíme několik technik pro analýzu textu, mezi něž patří dva moduly třetích stran s otevřeným zdrojovým kódem, které analyzování textu značně usnadňují.

```
                    "date={0.date!s}\n"
                      "aircraft_id={0.aircraft_id}\n"
                      "aircraft_type={0.aircraft_type}\n"
                      "airport={airport}\n"
                      "pilot_percent_hours_on_type="
                      "{0.pilot_percent_hours_on_type}\n"
                      "pilot_total_hours={0.pilot_total_hours}\n"
                      "midair={0.midair:d}\n"
                      ".NARRATIVE_START.\n{narrative}\n"
                      ".NARRATIVE_END.\n\n".format(incident,
                airport=incident.airport.strip(),
                narrative=narrative))
        return True
```

Konce řádků v textu popisu nejsou podstatné, takže můžeme text zalomit podle potřeby. Za normálních okolností bychom k tomuto účelu využili funkci `textwrap.wrap()` z modulu `textwrap`, avšak zde potřebujeme kromě zalomení také odsazení, a proto začínáme vytvořením objektu typu `textwrap.TextWrap`, který inicializujeme odsazením, které chceme použít (čtyři mezery pro první a následující řádky). Tento objekt ve výchozím nastavení zalamuje řádky na šířku 70 znaků, což můžeme změnit předáním dalšího klíčovaného argumentu.

modul datetime ➤ 212

Zápis bychom sice mohli provést také s použitím řetězce s trojitými uvozovkami, my ale raději vkládáme znaky nových řádků ručně. Objekt typu `textwrap.TextWrapper` nabízí metodu `wrap()`, která jako svůj vstup přijímá řetězec, kterým je v tomto případě text popisu, a vrací seznam řetězců s vhodným odsazením, z nichž každý není delší než šířka pro zalomení řádku. Tento seznam pak spojíme do jediného řetězce a jako oddělovač použijeme znak nového řádku.

str.format() ➤ 83

__format__() ➤ 249

Datum nehody je uchováváno jako objekt typu `datetime.date`. Při zápisu data jsme metodu `str.format()` přinutili použít řetězcovou reprezentaci, která má tvar RRRR-MM-DD, což odpovídá standardu ISO 8601. Pro metodu `str.format()` jsme stanovili, aby zapsala atribut `midair` typu `bool` jako celé číslo, což znamená 1 pro hodnotu `True` a 0 pro hodnotu `False`. Obecně je zápis textu pomocí metody `str.format()` velice jednoduchý, protože tato metoda se automaticky postará o všechny datové typy jazyka Python (a také o naše vlastní datové typy, pokud implementujeme speciální metodu `__str__()` nebo `__format__()`).

Analyzování textu

Metoda pro čtení a analyzování textového formátu záznamů o leteckých nehodách je delší a složitější než ta, kterou používáme pro zápis. Při čtení souboru se můžeme nacházet v jednom z několika stavů. Můžeme být uprostřed čtení řádků popisu, můžeme být na řádku s údaji *klíč=hodnota* anebo můžeme být na řádku s identifikačním číslem hlášení, na začátku nového záznamu o nehodě. Metodu `import_text_manual()` si tedy rozdělíme na pět částí.

```
    def import_text_manual(self, filename):
        fh = None
        try:
            fh = open(filename, encoding="utf8")
```

```
        self.clear()
        data = {}
        narrative = None
```

Tato metoda začíná otevřením souboru v režimu „čtení textu“. Poté vymažeme slovník s nehodami a stejným způsobem, jako při čtení binárních záznamů o nehodě vytvoříme datový slovník uchovávající data pro jedinou nehodu. Proměnnou `narrative` používáme ke dvěma účelům: jako indikátor stavu a pro uložení textu popisu aktuálního záznamu o nehodě. Má-li proměnná `narrative` hodnotu `None`, znamená to, že zatím popis nečteme. Pokud ale obsahuje řetězec (i prázdný), pak to znamená, že jsme v procesu čtení řádků s popisem.

```
        for lino, line in enumerate(fh, start=1):
            line = line.rstrip()
            if not line and narrative is None:
                continue
            if narrative is not None:
                if line == ".NARRATIVE_END.":
                    data["narrative"] = textwrap.dedent(
                                                narrative).strip()
                    if len(data) != 9:
                        raise IncidentError("na řádku {0} "
                                            "chybějí data".format(lino))
                    incident = Incident(**data)
                    self[incident.report_id] = incident
                    data = {}
                    narrative = None
                else:
                    narrative += line + "\n"
```

Vzhledem k tomu, že text čteme řádek po řádku, máme přehled o aktuálním číslu řádku, což můžeme využít k poskytnutí více informativní chybové zprávy než v případě čtení binárních souborů. Začínáme ořezáním případného bílého místa z konce řádku. Pokud nám zůstane prázdný řádek (a za předpokladu, že nejsme uprostřed popisu), jednoduše přeskočíme na další řádek. To znamená, že na počtu prázdných řádků mezi záznamy o nehodách nezáleží, ale prázdné řádky uvnitř textu popisu zachováme.

Má-li proměnná `narrative` hodnotu jinou než `None`, pak víme, že se nacházíme v textu popisu. Je-li daný řádek koncová značka popisu, je jisté, že jsme nejen dokončili čtení popisu, ale že jsme také dokončili čtení všech dat pro aktuální záznam nehody. V takovém případě umístíme text popisu do slovníku `data` (přičemž funkcí `textwrap.dedent()` odstraníme odsazení) a za předpokladu, že máme všech devět nezbytných kousků dat, vytvoříme nový objekt typu `Incident` a uložíme jej do slovníku. Pak slovník `data` vymažeme a resetujeme proměnnou `narrative` a tím jsme připraveni na další záznam. Ale pokud řádek není značka konce popisu, připojíme jej k popisu, a to včetně znaku nového řádku, který jsme na začátku ořízli.

```
elif (not data and line[0] == "["
                and line[-1] == "]"):
    data["report_id"] = line[1:-1]
```

Má-li proměnná `narrative` hodnotu `None`, pak se buď nacházíme u nového identifikačního čísla hlášení, nebo čteme některá z dalších dat. U nového identifikačního čísla hlášení bychom mohli být pouze tehdy, je-li slovník `data` prázdný (protože je prázdný na začátku a protože jej po přečtení každého záznamu o nehodě vymažeme) a řádek začíná `[` a končí `]`. V takovém případě umístíme identifikační číslo hlášení do slovníku `data`. To znamená, že tato podmínka `elif` nebude mít hodnotu `True`, dokud zase slovník `data` nevymažeme.

```
elif "=" in line:
    key, value = line.split("=", 1)
    if key == "date":
        data[key] = datetime.datetime.strptime(value,
                                    "%Y-%m-%d").date()
    elif key == "pilot_percent_hours_on_type":
        data[key] = float(value)
    elif key == "pilot_total_hours":
        data[key] = int(value)
    elif key == "midair":
        data[key] = bool(int(value))
    else:
        data[key] = value
elif line == ".NARRATIVE_START.":
    narrative = ""
else:
    raise KeyError("chyba při analýze na řádku {0}".format(
                   lino))
```

Pokud se nenacházíme v popisu a nečteme identifikační číslo nového hlášení, pak nám zbývají pouze tři možnosti: čteme prvky *`klíč=hodnota`*, jsme na počáteční značce popisu nebo je něco špatně.

V případě čtení řádku s daty *`klíč=hodnota`* tento řádek rozdělíme podle prvního znaku =, a stanovíme maximálně jedno dělení, což znamená, že *`hodnota`* může bez problémů obsahovat znaky =. Všechna načítaná data jsou ve formě řetězců ve znakové sadě Unicode, takže v případě datových typů představujících datum, číslo a logickou hodnotu musíme tuto hodnotu převést náležitým způsobem zpět.

Pro kalendářní data používáme funkci `datetime.datetime.strptime()` („string parse time“), která přijímá formátovací řetězec a vrací objekt typu `datetime.datetime`. Používáme formátovací řetězec, který odpovídá formátu data definovaného standardem ISO 8601, a pro získání objektu typu `datetime.date` z výsledného objektu typu `datetime.datetime` použijeme funkci `datetime.datetime.date()`, protože chceme pouze datum, ne datum a čas. Pro převod číselných hodnot využíváme vestavěné funkce Pythonu `float()` a `int()`. Je však třeba poznamenat, že kupříkladu `int("4.0")` vyvolá chybu `ValueError`. Pokud chceme být při přijímání celých čísel velkorysí, pak můžeme použít `int(float("4.0"))` nebo `round(float("4.0"))`, chceme-li výsledek místo ořezání zaokrouhlit.

Získání logické hodnoty je trošku komplikovanější. Například `bool("0")` vrací `True` (neprázdný řetězec má hodnotu `True`), takže řetězec musíme nejdříve převést na typ `int`.

Hodnoty, které jsou neplatné, chybějící nebo mimo rozsah, způsobí vždy vyvolání výjimky. Pokud kterýkoli z převodů selže, vyvolá se výjimka `ValueError`. A je-li některá hodnota mimo rozsah, vyvolá se při použití dat pro vytvoření odpovídajícího objektu typu `Incident` výjimka `IncidentError`.

Pokud řádek znak = neobsahuje, zkontrolujeme, zda jsme přečetli počáteční značku popisu. Pokud ano, tak nastavíme proměnnou `narrative` na prázdný řetězec. To znamená, že první podmínka `if` bude mít pro následné řádky hodnotu `True`, dokud nepřečteme koncovou značku popisu.

Nemá-li žádná z podmínek `if` nebo `elif` hodnotu `True`, tak došlo k chybě, a proto v poslední klauzuli `else` vyvoláme výjimku `KeyError`.

```
            return True
        except (EnvironmentError, ValueError, KeyError,
                IncidentError) as err:
            print("{0}: chyba při importu: {1}".format(
                  os.path.basename(sys.argv[0]), err))
            return False
        finally:
            if fh is not None:
                fh.close()
```

Po přečtení všech řádků vrátíme volajícímu hodnotu `True`, pokud ovšem nedošlo k nějaké výjimce, kterou zachytíme v bloku `except`, kde vypíšeme chybovou zprávu pro uživatele a vrátíme hodnotu `False`. A bez ohledu na to, co se stalo, se soubor na konci uzavře, pokud byl dříve otevřen.

Analyzování textu pomocí regulárních výrazů

Čtenářům neznalým regulárních výrazů („regex“) doporučujeme před touto částí nejdříve prostudovat lekci 13 nebo rovnou přeskočit na následující část (na stranu 303) a vrátit se sem v případě potřeby až později.

Použití regulárních výrazů pro analýzu textových souborů vede často ke kratšímu kódu než v případě ručního zpracování, které jsme si ukázali v předchozím oddílu. Na druhou stranu může být mnohem obtížnější implementovat kvalitní hlášení chyb. Metodu `import_text_regex()` si rozdělíme na dvě části. Nejdříve se podíváme na regulární výrazy a potom na samotné analyzování; vynecháme bloky `except` a `finally`, protože pro nás neobsahují nic nového.

```
    def import_text_regex(self, filename):
        incident_re = re.compile(
                        r"\[(?P<id>[^]]+)\](?P<keyvalues>.+?)"
                        r"^\.NARRATIVE_START\.$(?P<narrative>.*?)"
                        r"^\.NARRATIVE_END\.$",
                        re.DOTALL|re.MULTILINE)
        key_value_re = re.compile(r"^\s*(?P<key>[^=]+?)\s*=\s*"
                                  r"(?P<value>.+?)\s*$", re.MULTILINE)
```

Holé řetězce ➤ 72

Regulární výrazy se často zapisují jako holé řetězce. Díky tomu nemusíme každé zpětné lomítko zdvojovat (psát každé `\` jako `\\`). Například bez použití holých řetězců bychom museli druhý regulární výraz zapsat jako `"^\\s*(?P<key>[^=]+?)\\s*=\\s*(?P<value>.+?)\\s*$"`. V této knize budeme pro regulární výrazy vždy používat holé řetězce.

První regulární výraz `incident_re` používáme pro zachycení celého záznamu o nehodě. Důsledkem je, že jakýkoli rušivý text *mezi* záznamy bude ignorován. Tento regulární výraz má ve skutečnosti dvě části. První část `\[(?P<id>[^]]+)\](?P<keyvalues>.+?)` se shoduje se znakem `[`, pak se shoduje s libovolným množstvím znaků odlišných od znaku `]`, které zachytí do skupiny `id`, pak se shoduje se znakem `[` (čímž získáme identifikační číslo hlášení) a potom se shoduje s minimálním (ale s alespoň jedním) množstvím znaků (díky příznaku `re.DOTALL` včetně znaků nového řádku), které zachytí do skupiny `keyvalues`. Znaky zachycené do skupiny `keyvalues` představují minimum nezbytné k tomu, abychom se přesunuli do druhé části tohoto regulárního výrazu.

Druhá část prvního regulárního výrazu je `^\.NARRATIVE_START\.$ (?P<narrative>.*?)^\.NARRATIVE_END\.$` a shoduje se s textovým literálem `.NARRATIVE_START.`, pak s minimálně možným množstvím znaků, které zachytí do skupiny `narrative`, a potom s textovým literálem `.NARRATIVE_END.`, který představuje konec záznamu o nehodě. Příznak `re.MULTILINE` znamená, že v tomto regulárním výrazu se `^` shoduje se začátkem každého řádku (a ne jen na začátku řetězce) a `$` se shoduje s koncem každého řádku (a ne jen s koncem každého řetězce), takže shoda s počátečními a koncovými značkami popisu je možná jen na začátku řádků.

Druhý regulární výraz `key_value_re` se používá pro zachycení řádků *`klíč=hodnota`* a shoduje se se začátkem každého řádku v zadaném textu, kde tento řádek začíná libovolným množstvím (včetně žádného) bílého místa, za nímž následují znaky odlišné od `=`, které se zachytí do skupiny `key`, a potom znak `=` následovaný všemi zbývajícími znaky na řádku (včetně případného úvodního a koncového bílého místa), které se zachytí do skupiny `value`.

Základní logika použití pro analýzu souboru je stejná jako při manuální analýze textu, kterou jsme si ukázali v předchozím oddíle, avšak nyní neextrahujeme záznamy o nehodách a data v těchto záznamech čtením řádek po řádku, ale pomocí regulárních výrazů.

```
fh = None
try:
    fh = open(filename, encoding="utf8")
    self.clear()
    for incident_match in incident_re.finditer(fh.read()):
        data = {}
        data["report_id"] = incident_match.group("id")
        data["narrative"] = textwrap.dedent(
                    incident_match.group("narrative")).strip()
        keyvalues = incident_match.group("keyvalues")
        for match in key_value_re.finditer(keyvalues):
            data[match.group("key")] = match.group("value")
        data["date"] = datetime.datetime.strptime(
                            data["date"], "%Y-%m-%d").date()
        data["pilot_percent_hours_on_type"] = (
```

```
                float(data["pilot_percent_hours_on_type"]))
            data["pilot_total_hours"] = int(
                data["pilot_total_hours"])
            data["midair"] = bool(int(data["midair"]))
            if len(data) != 9:
                raise IncidentError("missing data")
            incident = Incident(**data)
            self[incident.report_id] = incident
        return True
```

Metoda `re.finditer()` vrací iterátor, který postupně produkuje nepřekrývající se shody. Stejně jako předtím vytváříme i nyní slovník `data` pro uchovávání dat o nehodě, ovšem tentokrát získáváme identifikační číslo hlášení a text popisu z nalezených skupin regulárního výrazu `incident_re`. Poté s použitím skupiny `keyvalues` naráz extrahujeme všechny řetězce *`klíč=hodnota`* a pomocí metody `re.finditer()` regulárního výrazu `key_value_re` procházíme jednotlivé řádky *`klíč=hodnota`*. Každou nalezenou dvojici (klíč, hodnota) umístíme do slovníku data, takže všechny hodnoty mají podobu řetězců. Potom hodnoty, které nemají být řetězce, nahradíme hodnotou příslušného typu; provádíme to stejným způsobem jako při ruční analýze textu.

Přidali jsme kontrolu pro zajištění, že slovník data má devět prvků, protože pokud by byl nějaký záznam o nehodě porušený, mohl by iterátor `key_value.finditer()` nalézt shodu pro příliš mnoho nebo málo řádků tvaru *`klíč=hodnota`*. Konec je stejný jako dříve – vytváříme nový objekt typu `Incident`, umístíme jej do slovníku se záznamy o nehodách a poté vrátíme hodnotu `True`. Pokud něco neproběhlo správně, vypíše sada `except` vhodnou chybovou zprávu a vrátí hodnotu `False`; sada `finally` se postará o uzavření souboru.

Jednou z věcí, díky které jsou ruční analyzátory textu i analyzátory textu využívající regulární výrazy tak krátké a přímočaré, je zpracování výjimek v jazyku Python. Tyto analyzátory nemusejí kontrolovat žádné převody řetězců na kalendářní data, čísla či logické hodnoty a dále nemusejí provádět žádnou kontrolu rozsahu hodnot (o to se postará třída `Incident`). Pokud některá z těchto věcí selže, vyvolá se výjimka a my pak všechny výjimky elegantně zpracujeme na jednom místě na konci. Další výhoda výjimek oproti explicitní kontrole spočívá ve skvělé škálovatelnosti kódu. Dokonce ani tehdy, když do formátu záznamu přibudou nové datové prvky, není nutné kód pro ošetření chyb žádným způsobem rozšiřovat.

Zapisování a analyzování souborů XML

Některé programy používají souborový formát XML pro všechna data, s nimiž pracují, zatímco jiné používají XML jako příhodný formát pro import a export. Schopnost importovat a exportovat XML je užitečná a stojí vždy za uvážení, dokonce i tehdy, je-li hlavním formátem programu textový či binární formát.

Python nabízí tři způsoby zápisu souborů XML: manuální zápis kódu jazyka XML, vytvoření stromu elementů a použití jeho metody `write()` a vytvoření modelu DOM a použití jeho metody `write()`. Podobně lze pro čtení a analýzu souborů XML použít některý ze čtyř přístupů: manuální čtení a analýzu kódu jazyka XML (nedoporučuje se a ani my se tomuto přístupu zde věnovat nebudeme, protože správné zvládnutí některých méně srozumitelných a pokročilejších detailů může být docela obtíž-

né) nebo použití analyzátoru stromu elementů, modelu DOM či rozhraní SAX. Kromě toho stojí za prozkoumání také tři další knihovny třetích stran pro práci s XML, mezi něž patří i knihovna lxml zmíněná v lekci 5 (strana 222).

Formát XML pro záznam o letecké nehodě si můžete prohlédnout na obrázku 7.3. V této části si ukážeme, jak tento formát zapisovat ručně a jak jej zapisovat pomocí stromu elementů a modelu DOM a také jak tento formát číst a analyzovat pomocí stromu elementů, modelu DOM a rozhraní SAX. Pokud vás nezajímá, který přístup se pro čtení a zápis kódu jazyka XML používá, pak stačí, když si přečtete jen následující oddíl „Stromy elementů“, a poté můžete přeskočit na poslední část této lekce („Binární soubory s náhodným přístupem“ – strana 314).

```
<?xml version="1.0" encoding="UTF-8"?>
<incidents>
<incident report_id="20070222008099G" date="2007-02-22"
    aircraft_id="80342" aircraft_type="CE-172-M"
    pilot_percent_hours_on_type="9.09090909091"
    pilot_total_hours="440" midair="0">
<airport>BOWERMAN</airport>
<narrative>
ON A GO-AROUND FROM A NIGHT CROSSWIND LANDING ATTEMPT THE AIRCRAFT HIT
A RUNWAY EDGE LIGHT DAMAGING ONE PROPELLER.
</narrative>
</incident>
<incident>
    ...
</incident>
    :
</incidents>
```

Obrázek 7.3: Ukázkový formát XML pro záznam o letecké nehodě

Stromy elementů

Zapisování dat pomocí stromu elementů se provádí ve dvou fázích. Nejdříve je nutné vytvořit strom elementů reprezentující data, a pak se tento strom zapíše do souboru. Některé programy mohou používat strom elementů jako svoji datovou strukturu. V takovém případě již strom mají a mohou jednoduše zapsat data. Metodu `export_xml_etree()` si rozdělíme na dvě části:

```
def export_xml_etree(self, filename):
    root = xml.etree.ElementTree.Element("incidents")
    for incident in self.values():
        element = xml.etree.ElementTree.Element("incident",
                report_id=incident.report_id,
                date=incident.date.isoformat(),
                aircraft_id=incident.aircraft_id,
                aircraft_type=incident.aircraft_type,
                pilot_percent_hours_on_type=str(
                        incident.pilot_percent_hours_on_type),
                pilot_total_hours=str(incident.pilot_total_hours),
                midair=str(int(incident.midair)))
```

```
        airport = xml.etree.ElementTree.SubElement(element,
                                                   "airport")
        airport.text = incident.airport.strip()
        narrative = xml.etree.ElementTree.SubElement(element,
                                                     "narrative")
        narrative.text = incident.narrative.strip()
        root.append(element)
    tree = xml.etree.ElementTree.ElementTree(root)
```

Začínáme vytvořením kořenového elementu (`<incidents>`). Pak procházíme všechny záznamy o nehodách. Pro každý z nich vytvoříme element (`<incident>`) pro data dané nehody a pomocí klíčovaných argumentů nastavíme příslušné atributy. Všechny atributy musejí mít podobu textu, a proto musíme datum, čísla a logickou hodnotu odpovídajícím způsobem převést na text. O kódování znaků „&“, „<“ a „>“(nebo o uvozovky v hodnotách atributů) se starat nemusíme, protože modul pro strom elementů – a také moduly pro model DOM a rozhraní SAX – se o tyto detaily postarají zcela automaticky.

Každý element `incident` má dva podelementy, z nichž jeden uchovává název letiště a druhý text popisu. Při vytváření podelementů musíme uvést rodičovský element a název značky. Atribut elementu s názvem `text`, který je určený pro čtení i zápis, se používá pro uchovávání textu.

Jakmile máme element incident vytvořený se všemi atributy a podelementy `airport` a `narrative`, přidáme jej do kořenového elementu hierarchie (`incidents`). Na konci máme hierarchii elementů, jež obsahují data všech záznamů o nehodách, které pak snadno převedeme na strom elementů.

```
    try:
        tree.write(filename, "UTF-8")
    except EnvironmentError as err:
        print("{0}: chyba při exportu: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
    return True
```

Pro zapsání kódu jazyka XML reprezentujícího celý strom elementů stačí už jen tomuto stromu říci, aby se zapsal do zadaného souboru s použitím zadaného kódování.

Až dosud jsme při specifikaci kódování téměř vždy používali řetězec `"utf8"`. Ten funguje skvěle pro vestavěnou funkci Pythonu `open()`, která dokáže přijímat široké spektrum kódování pojmenované nejrůznějšími názvy, jako je například `"UTF-8"`, `"UTF8"`, `"utf-8"` a `"utf8"`. Avšak u souborů XML může mít název kódování pouze oficiální tvar, takže `"utf8"` je nepřijatelné, a proto musíme použít `"UTF-8"`.*

Čtení souboru XML pomocí stromu elementů není o mnoho těžší než jeho zapsání. Opět zde máme dvě fáze. Nejdříve přečteme a analyzujeme soubor XML a poté výsledný strom elementů projdeme a vytáhneme z něj data pro naplnění slovníku se záznamy o nehodách. Ani nyní není druhá fáze

* Více informací o kódování znaků v souborech XML najdete na stránkách *www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl a www.iana.org/assignments/character-sets.*

nutná, pokud se samotný strom elementů používá jako paměťové datové úložiště. Zde je metoda `import_xml_etree()` rozdělená na dvě části.

```
    def import_xml_etree(self, filename):
        try:
            tree = xml.etree.ElementTree.parse(filename)
        except (EnvironmentError,
                xml.parsers.expat.ExpatError) as err:
            print("{0}: chyba při importu: {1}".format(
                  os.path.basename(sys.argv[0]), err))
            return False
```

Analyzátor stromu elementů používá ve výchozím nastavení analyzátor XML expat, a proto musíme být připraveni zachytit jeho výjimky.

```
        self.clear()
        for element in tree.findall("incident"):
            try:
                data = {}
                for attribute in ("report_id", "date", "aircraft_id",
                        "aircraft_type",
                        "pilot_percent_hours_on_type",
                        "pilot_total_hours", "midair"):
                    data[attribute] = element.get(attribute)
                data["date"] = datetime.datetime.strptime(
                                    data["date"], "%Y-%m-%d").date()
                data["pilot_percent_hours_on_type"] = (
                        float(data["pilot_percent_hours_on_type"]))
                data["pilot_total_hours"] = int(
                        data["pilot_total_hours"])
                data["midair"] = bool(int(data["midair"]))
                data["airport"] = element.find("airport").text.strip()
                narrative = element.find("narrative").text
                data["narrative"] = (narrative.strip()
                                     if narrative is not None else "")
                incident = Incident(**data)
                self[incident.report_id] = incident
            except (ValueError, LookupError, IncidentError) as err:
                print("{0}: chyba při importu: {1}".format(
                    os.path.basename(sys.argv[0]), err))
                return False
        return True
```

Jakmile máme strom elementů, můžeme projít všechny elementy `incident` pomocí metody `xml.etree.ElementTree.findall()`. Každý element `incident` obdržíme jako objekt typu `xml.etree.Element`. Pro práci s atributy elementů používáme stejnou techniku jako v předchozí části v metodě `import_text_regex()`. Nejdříve uložíme všechny hodnoty do slovníku `data` a poté převedeme

hodnoty, které představují kalendářní datum, číslo nebo logickou hodnotu na správný typ. Pomocí metody `xml.etree.Element.find()` vyhledáme podelementy `airport` a `narrative` a přečteme jejich atributy `text`. Pokud textový element nemá žádný text, jeho atribut text bude mít hodnotu `None`, s čímž musíme počítat při čtení textu v elementu popisku, který může být prázdný. Hodnoty atributů a obdržený text v každém případě neobsahují zakódované symboly používané pro kód jazyka XML, protože tyto symboly jsou automaticky převedeny zpět.

Stejně jako u všech analyzátorů XML používaných pro zpracování dat o letecké nehodě dojde i zde k výjimce, pokud chybí element `aircraft` či `narrative` nebo některý z atributů, pokud některý z převodu selže nebo pokud jsou číselná data mimo rozsah. Tím máme zajištěno, že neplatná data způsobí zastavení analýzy a vypíše se chybová zpráva. Kód na konci pro vytvoření uložení záznamů o nehodách a pro zpracování výjimek je stejný jako v předchozích případech.

Model DOM (Document Object Model)

Model DOM (objektový model dokumentu) je standardní rozhraní API pro reprezentaci dokumentu XML v paměti a pro manipulaci s tímto dokumentem. Kód pro vytvoření modelu DOM, jeho zapsání do souboru a pro analyzování souboru XML pomocí modelu DOM je svojí strukturou velice podobný kódu se stromem elementů, jen je malinko delší.

Začneme prostudováním metody `export_xml_dom()`, kterou si rozdělíme na dvě části. Tato metoda pracuje ve dvou fázích. Nejdříve se vytvoří model DOM odrážející data o nehodě a poté se tento model DOM zapíše do souboru. Stejně jako u stromu elementů mohou i v tomto případě některé programy používat model DOM jako svoji datovou strukturu. V takovém případě mohou jednoduše zapsat svá data.

```
def export_xml_dom(self, filename):
    dom = xml.dom.minidom.getDOMImplementation()
    tree = dom.createDocument(None, "incidents", None)
    root = tree.documentElement
    for incident in self.values():
        element = tree.createElement("incident")
        for attribute, value in (
                ("report_id", incident.report_id),
                ("date", incident.date.isoformat()),
                ("aircraft_id", incident.aircraft_id),
                ("aircraft_type", incident.aircraft_type),
                ("pilot_percent_hours_on_type",
                 str(incident.pilot_percent_hours_on_type)),
                ("pilot_total_hours",
                 str(incident.pilot_total_hours)),
                ("midair", str(int(incident.midair)))):
            element.setAttribute(attribute, value)
        for name, text in (("airport", incident.airport),
                           ("narrative", incident.narrative)):
            text_element = tree.createTextNode(text)
            name_element = tree.createElement(name)
```

```
            name_element.appendChild(text_element)
            element.appendChild(name_element)
        root.appendChild(element)
```

Tato metoda začíná získáním implementace rozhraní pro práci s modely DOM. Tuto implementaci standardně poskytuje analyzátor XML expat. Modul `xml.dom.minidom` nabízí ve srovnání s modulem `xml.dom` jednodušší a menší implementaci, ačkoliv objekty, které používá, pocházejí z modulu `xml.dom`. Jakmile máme implementaci rozhraní pro práci s modely DOM k dispozici, můžeme vytvořit dokument. Prvním argumentem metody `xml.dom.DOMImplementation.createDocument()` je identifikátor URI oboru názvů, který nepotřebujeme, a proto použijeme hodnotu `None`, druhým argumentem je kvalifikovaný název (název značky pro kořenový element) a třetím argumentem je typ dokumentu, pro který opět použijeme hodnotu `None`, protože typ dokumentu nedefinujeme. Jakmile máme strom reprezentující tento dokument, vezmeme kořenový element a začneme procházet všechny záznamy o nehodách.

Pro každý záznam o nehodě vytvoříme element `incident` a pro každý atribut, který má tento element obsahovat, zavoláme metodu `setAttribute()` s názvem a hodnotou atributu. Stejně jako u stromu elementů se ani zde nemusíme starat o zakódování znaků „&“, „<“ a „>“ (nebo o uvozovky v hodnotách atributů). Pro textové elementy `airport` a `narrative` musíme vytvořit textový element, který bude uchovávat text, a obyčejný element (s příslušným názvem značky) jako rodič textového elementu. Poté obyčejný element (a v něm obsažený textový element) přidáme do aktuálního elementu `incident`. Kompletní element `incident` pak přidáme do kořenového elementu.

```
        fh = None
        try:
            fh = open(filename, "w", encoding="utf8")
            tree.writexml(fh, encoding="UTF-8")
            return True
```

Kódování XML ➢ 305

Vynechali jsme bloky `except` a `finally`, protože jsou stejné jako ty, které jsme již viděli. Na této části kódu je jasně vidět dříve probíraný rozdíl mezi řetězcem s názvem kódování pro vestavěnou funkci `open()` a pro soubory XML.

Importování dokumentu XML do modelu DOM je podobné importování do stromu elementů, avšak podobně jako export vyžaduje více kódu. Funkci `import_xml_dom()` si rozdělíme na tři části, přičemž začneme řádkem s příkazem `def` a vnořenou funkcí `get_text()`.

```
    def import_xml_dom(self, filename):

        def get_text(node_list):
            text = []
            for node in node_list:
                if node.nodeType == node.TEXT_NODE:
                    text.append(node.data)
            return "".join(text).strip()
```

Funkce `get_text()` prochází seznam uzlů (např. potomky nějakého uzlu) a pro ty z nich, které jsou textové, extrahuje jejich text a připojí jej do seznamu textů. Tato funkce na konci vrací veškerý nashromážděný text jako jediný řetězec s ořezaným bílým místem na obou koncích.

```
try:
    dom = xml.dom.minidom.parse(filename)
except (EnvironmentError,
        xml.parsers.expat.ExpatError) as err:
    print("{0}: chyba při importu: {1}".format(
          os.path.basename(sys.argv[0]), err))
    return False
```

Analyzování souboru XML do modelu DOM je jednoduché, protože tento modul provede všechnu obtížnou práci za nás. Musíme ale být připraveni zpracovat chyby knihovny expat, protože stejně jako u stromu elementů je analyzátor XML expat výchozím analyzátorem používaným třídami rozhraní pro práci s modelem DOM.

```
self.clear()
for element in dom.getElementsByTagName("incident"):
    try:
        data = {}
        for attribute in ("report_id", "date", "aircraft_id",
                "aircraft_type",
                "pilot_percent_hours_on_type",
                "pilot_total_hours", "midair"):
            data[attribute] = element.getAttribute(attribute)
        data["date"] = datetime.datetime.strptime(
                            data["date"], "%Y-%m-%d").date()
        data["pilot_percent_hours_on_type"] = (
                float(data["pilot_percent_hours_on_type"]))
        data["pilot_total_hours"] = int(
                data["pilot_total_hours"])
        data["midair"] = bool(int(data["midair"]))
        airport = element.getElementsByTagName("airport")[0]
        data["airport"] = get_text(airport.childNodes)
        narrative = element.getElementsByTagName(
                                            "narrative")[0]
        data["narrative"] = get_text(narrative.childNodes)
        incident = Incident(**data)
        self[incident.report_id] = incident
    except (ValueError, LookupError, IncidentError) as err:
        print("{0}: chyba při importu: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
return True
```

Lokální funkce ➤ 341

Jakmile model DOM existuje, vymažeme aktuální data o nehodách a začneme procházet všechny značky `<incident>`. U každé z nich extrahujeme atributy a datum, číslo a logickou hodnotu převedeme na správné typy naprosto stejným způsobem jako při použití stromu elementů. Jediný skutečně podstatný rozdíl mezi modelem DOM a stromem elementů tkví v práci s textovými uzly. Pro získání elementů potomků se zadaným názvem značky používáme metodu `xml.dom.Element.getElementsByTagName()`. V případě elementů `airport` a `narrative` víme, že potomek bude vždy jeden, a proto bereme první (a jediný) element potomka každého typu. Pak tyto uzly potomků značek pomocí funkce `nested get_text()` procházíme a extrahujeme jejich texty.

Jako obvykle zachytíme v případě chyby relevantní výjimku, vypíšeme chybovou zprávu pro uživatele a vrátíme hodnotu `False`.

Rozdíly v přístupu mezi modelem DOM a stromem elementů nejsou nijak velké, a protože u obou se používá analyzátor expat, jsou oba přiměřeně rychlé.

Ruční zápis kódu jazyka XML

Zápis stávajícího stromu elementů nebo modelu DOM do dokumentu XML lze provést zavoláním jediné metody. Pokud ale naše data dosud nejsou v některé z těchto forem, pak musíme strom elementů nebo model DOM nejdříve vytvořit. V takovém případě může být pohodlnější jednoduše zapsat naše data přímo.

Při vytváření souboru XML musíme zajistit správné zakódování textu a hodnot atributů a zapsat dobře utvořený dokument XML. Zde je metoda `export_xml_manual()` pro zápis záznamů o nehodách do souboru XML:

```
def export_xml_manual(self, filename):
    fh = None
    try:
        fh = open(filename, "w", encoding="utf8")
        fh.write('<?xml version="1.0" encoding="UTF-8"?>\n')
        fh.write("<incidents>\n")
        for incident in self.values():
            fh.write('<incident report_id={report_id} '
                     'date="{0.date!s}" '
                     'aircraft_id={aircraft_id} '
                     'aircraft_type={aircraft_type} '
                     'pilot_percent_hours_on_type='
                     '"{0.pilot_percent_hours_on_type}" '
                     'pilot_total_hours="{0.pilot_total_hours}" '
                     'midair="{0.midair:d}">\n'
                     '<airport>{airport}</airport>\n'
                     '<narrative>\n{narrative}\n</narrative>\n'
                     '</incident>\n'.format(incident,
                report_id=xml.sax.saxutils.quoteattr(
                                    incident.report_id),
                aircraft_id=xml.sax.saxutils.quoteattr(
```

```
                                incident.aircraft_id),
                aircraft_type=xml.sax.saxutils.quoteattr(
                                incident.aircraft_type),
                airport=xml.sax.saxutils.escape(incident.airport),
                narrative="\n".join(textwrap.wrap(
                        xml.sax.saxutils.escape(
                            incident.narrative.strip()), 70))))
        fh.write("</incidents>\n")
        return True
```

Jako obvykle v této lekci jsme i zde bloky `except` a `finally` vynechali.

Soubor zapisujeme s použitím kódování UTF-8, což musíme uvést při volání vestavěné funkce `open()`. Kódování v deklaraci `<?xml?>` ve skutečnosti uvádět nemusíme, protože UTF-8 je výchozí kódování, my se ale snažíme být explicitní. Rozhodli jsme se uzavřít hodnoty všech atributů do dvojitých uvozovek, a proto jsme řetězce, které vkládáme do záznamů o nehodách, uzavřeli do jednoduchých uvozovek, díky čemuž nemusíme používat zpětná lomítka pro potlačení uvozovek.

Funkce `sax.saxutils.quoteattr()` je podobná funkci `sax.saxutils.escape()`, kterou používáme pro text XML, a to v tom, že náležitě zakóduje znaky „&“, „<“ a „>“. Kromě toho zakóduje také uvozovky (je-li to nutné) a vrátí řetězec, který je uzavřen do uvozovek, a je tedy připraven k použití. Z tohoto důvodu jsme nemuseli umisťovat uvozovky kolem identifikačního čísla hlášení a kolem dalších řetězců s hodnotami atributů.

Vkládané znaky nového řádku a zalomení textu pro popis má čistě kosmetický význam. Jejich smyslem je usnadnit člověku čtení a úpravu souboru, klidně bychom je však mohli vypustit.

Zapisování dat ve formátu HTML není příliš odlišné od zapisování kódu jazyka XML. Jednoduchým příkladem je funkce `export_html()` obsažená v programu `convert-incidents.py`. My si ji zde ale neukážeme, protože ve skutečnosti neobsahuje nic nového.

Analýza kódu jazyka XML pomocí rozhraní SAX (Simple API for XML)

Na rozdíl od stromu elementů a modelu DOM, které reprezentují celý dokument XML v paměti, pracují analyzátory SAX inkrementálně, což může být rychlejší a méně náročné na paměti. Nicméně výhodu vyššího výkonu nemůžeme předjímat, zvláště pak proto, že strom elementů i model DOM používají rychlý analyzátor expat.

Analyzátory SAX pracují tak, že když narazí na počáteční značku, koncovou značku a další prvky jazyka XML, tak oznamují „události analýzy“. Pro obsluhu těch událostí, které nás zajímají, musíme vytvořit vhodnou obslužnou třídu a implementovat v ní několik předdefinovaných metod, které se volají v okamžiku, kdy nastanou odpovídající události analýzy. Nejčastěji implementovanou obsluhou je obsluha obsahu. Kromě toho lze implementovat také obsluhy chyb a další obsluhy, chceme-li jemnější kontrolu.

Zde je celá metoda `import_xml_sax()`. Je velice krátká, protože většinu práce provádí naše vlastní třída `IncidentSaxHandler`:

```
    def import_xml_sax(self, filename):
        fh = None
        try:
            handler = IncidentSaxHandler(self)
            parser = xml.sax.make_parser()
            parser.setContentHandler(handler)
            parser.parse(filename)
            return True
        except (EnvironmentError, ValueError, IncidentError,
                xml.sax.SAXParseException) as err:
            print("{0}: chyba při importu: {1}".format(
                  os.path.basename(sys.argv[0]), err))
            return False
```

Vytváříme jedinou obsluhu, kterou chceme použít, a poté vytváříme analyzátor SAX a nastavujeme jeho obsluhu obsahu na naši vytvořenou obsluhu. Pak předáme název souboru metodě `parse()` analyzátoru, která vrátí hodnotu `True`, pokud při analýze nedojde k žádným chybám.

Inicializační metodě naší vlastní třídy `IncidentSaxHandler` předáváme objekt `self` (tj. tento objekt typu `IncidentCollection`, který je odvozen od typu `dict`). Obsluha vymaže staré záznamy o nehodách a poté bude při analyzování souboru sestavovat slovník se záznamy o nehodách. Po dokončení analýzy bude slovník obsahovat záznamy o všech načtených nehodách.

```
class IncidentSaxHandler(xml.sax.handler.ContentHandler):

    def __init__(self, incidents):
        super().__init__()
        self.__data = {}
        self.__text = ""
        self.__incidents = incidents
        self.__incidents.clear()
```

Naše obslužná třída pro události rozhraní SAX musí být odvozena od příslušné bázové třídy. Tím je zajištěno, že se pro jakékoli metody, které nejsou reimplementované (protože nás nezajímají události analýzy, které obsluhují), zavolá verze bázové třídy, která bezpečně neprovede vůbec nic.

Začínáme voláním inicializační metody bázové třídy. Jedná se obecně o vhodný postup pro všechny podtřídy, ačkoliv pro třídy odvozené přímo od třídy `object` to není nutné (je to však neškodné). Slovník `self.__data` se používá pro uložení dat o jedné nehodě, řetězec `self.__text` se používá pro uložení textu s názvem letiště nebo popisu v závislosti na tom, který údaj se čte, a slovník `self.__incidents` je odkaz na objekt ukazující na slovník typu `IncidentCollection`, který tato obsluha aktualizuje přímo. (Alternativní design by mohl spočívat v existenci nezávislého slovníku uvnitř obsluhy, který by se na konci zkopíroval do objektu typu `IncidentCollection` pomocí metod `dict.clear()` a `dict.update()`.)

```
    def startElement(self, name, attributes):
        if name == "incident":
```

```
            self.__data = {}
            for key, value in attributes.items():
                if key == "date":
                    self.__data[key] = datetime.datetime.strptime(
                                               value, "%Y-%m-%d").date()
                elif key == "pilot_percent_hours_on_type":
                    self.__data[key] = float(value)
                elif key == "pilot_total_hours":
                    self.__data[key] = int(value)
                elif key == "midair":
                    self.__data[key] = bool(int(value))
                else:
                    self.__data[key] = value
        self.__text = ""
```

Kdykoli se čte počáteční značka a její atributy, zavolá se metoda `xml.sax.handler.ContentHandler.startElement()` s názvem značky a jejími atributy. V případě souboru XML s leteckými nehodami patří mezi počáteční značky značka `<incidents>`, kterou ignorujeme, dále značka `<incident>`, jejíž atributy používáme k naplnění části slovníku `self.__data`, a značky `<airport>` a `<narrative>`, které též ignorujeme. Při výskytu počáteční značky vždy vymažeme řetězec `self.__text`, protože žádné textové značky nejsou ve formátu souboru XML s leteckými nehodami vnořené.

Ve třídě `IncidentSaxHandler` nezpracováváme žádné výjimky. Pokud dojde k nějaké výjimce, pak bude předána volajícímu, což je v tomto případě metoda `import_xml_sax()`, která ji zachytí a vypíše vhodnou chybovou zprávu.

```
    def endElement(self, name):
        if name == "incident":
            if len(self.__data) != 9:
                raise IncidentError("chybějící data")
            incident = Incident(**self.__data)
            self.__incidents[incident.report_id] = incident
        elif name in frozenset({"airport", "narrative"}):
            self.__data[name] = self.__text.strip()
        self.__text = ""
```

Při načtení koncové značky se zavolá metoda `xml.sax.handler.ContentHandler.endElement()`. Jsme-li na konci záznamu o nehodě, měli bychom mít všechna nezbytná data, takže vytvoříme nový objekt typu `Incident` a přidáme jej do slovníku se záznamy nehod. Jsme-li na konci textového elementu, přidáme prvek do slovníku `self.__data` s textem, který byl až dosud nashromážděn. Na konci řetězec `self.__text` vymažeme, takže bude připraven pro další použití. (My jej přesněji řečeno vymazat nemusíme, protože jej vymažeme při výskytu počáteční značky, jeho vymazání však může být v některých formátech XML důležité, například při možnosti vnořování značek.)

```
    def characters(self, text):
        self.__text += text
```

Když analyzátor SAX přečte text, zavolá metodu `xml.sax.handler.ContentHandler.characters()`. Neexistuje žádná záruka, že se tato metoda zavolá jen jednou s celým textem. Text totiž může přicházet postupně, po částech. Z tohoto důvodu zde máme metodu pro akumulaci textu, který vkládáme do slovníku až tehdy, když dojdeme na konec dané značky. (Efektivnější implementací by bylo mít `self.__text` jako seznam v těle metody `self.__text.append(text)` a v dalších metodách odpovídajícím způsobem upravený kód.)

Používání rozhraní SAX je značně odlišné od používání stromu elementů nebo modelu DOM, je však přinejmenším stejně efektivní. Můžeme doplnit i jiné obsluhy a v obsluze obsahu reimplementovat další metody, čímž získáme tolik kontroly, kolik jen chceme. Samotný analyzátor SAX neudržuje žádnou reprezentaci dokumentu XML. Díky tomu je ideální pro čtení XML do naší vlastní kolekce dat. Dále z toho plyne, že zde nemáme žádný „dokument" SAX, do kterého bychom mohli zapsat kód jazyka XML, pro který tak musíme použít některý z přístupů, které jsme si popsali již dříve v této části.

Binární soubory s náhodným přístupem

V předchozích částech jsme vytvářeli programy, které pracovaly tak, že se všechna data programu načetla naráz do paměti, zpracovala se a potom naráz zapsala. Moderní počítače mají tolik paměti RAM, že můžeme s klidem prohlásit, že tento způsob fungování programů je plně realizovatelný, a to i s rozsáhlými skupinami dat. Jenže v některých situacích může být lepším řešením uchovávat data na disku, číst jen části, které potřebujeme, a zapisovat zpět změny. Techniku na disku založeného náhodného přístupu lze nejsnadněji realizovat pomocí databáze dvojic klíč-hodnota („DBM") nebo pomocí kompletní databáze SQL (obě možnosti budeme probírat v lekci 12). V této části si však ukážeme, jak pracovat se soubory s náhodným přístupem ručně.

Nejdříve si představíme třídu `BinaryRecordFile.BinaryRecordFile`. Instance této třídy představují generický čitelný a zapisovatelný binární soubor strukturovaný jako posloupnost záznamů fixní délky. Poté se podíváme na třídu `BikeStock.BikeStock`, která uchovává kolekci objektů typu `BikeStock.Bike` jako záznamy v objektu typu `BinaryRecordFile.BinaryRecordFile`, a ukážeme si, jak využít binární soubory s náhodným přístupem.

Generická třída BinaryRecordFile

Rozhraní API třídy `BinaryRecordFile.BinaryRecordFile` je podobné seznamu, protože můžeme číst, zapisovat a mazat záznamy na zadané indexové pozici. Při mazání záznamu je daný záznam jen označen jako „vymazán", díky čemuž nemusíme pro zaplnění mezery přesouvat všechny následující záznamy a navíc zůstanou původní indexové pozice platné i po vymazání. Další výhodou je, že vymazání záznamu lze stornovat zrušením příslušného označení. Cena, kterou za to platíme, spočívá v tom, že mazáním záznamů neušetříme žádné místo na disku. Tento problém vyřešíme implementováním metod pro „stlačení" souboru, které odstraní vymazané záznamy (a zneplatní indexové pozice).

Ještě před tím, než si prohlédneme implementaci, podíváme se na základní použití:

```
Contact = struct.Struct("<15si")
contacts = BinaryRecordFile.BinaryRecordFile(filename, Contact.size)
```

Zde vytváříme objekt typu `struct.Struct` (binární struktura – řazení bajtů Little Endian, 15 bajtů dlouhý řetězec bajtů a 4bajtové celé číslo se znaménkem), který použijeme pro reprezentaci každého záznamu. Poté vytváříme instanci třídy `BinaryRecordFile.BinaryRecordFile` s názvem souboru a s velikostí záznamu, která odpovídá námi definované binární struktuře `Contact`. Pokud soubor existuje, bude otevřen a jeho obsah zůstane nedotčen, jinak se daný soubor vytvoří. V každém případě se soubor otevře v režimu binárního čtení a zápisu, takže do něj můžeme zapisovat data:

```
contacts[4] = Contact.pack("Adam Bauer".encode("utf8"), 762)
contacts[5] = Contact.pack("Eva Drobná".encode("utf8"), 987)
```

S tímto souborem můžeme pomocí operátoru pro přístup k prvkům (`[]`) pracovat jako se seznamem. Zde přiřazujeme dva bajtové řetězce (objekty typu `bytes`, z nichž každý obsahuje řetězec a celé číslo) na indexové pozice dvou záznamů v souboru. Tato přiřazení přepíší jakýkoliv stávající obsah. Pokud soubor zatím nemá šest záznamů, budou předchozí záznamy vytvořeny s každým bajtem nastaveným na hodnotu `0x00`.

```
contact_data = Contact.unpack(contacts[5])
contact_data[0].decode("utf8").rstrip(chr(0)) # vrátí: 'Eva Drobná'
```

Řetězec „Eva Drobná“ je kratší než 15 znaků kódovaných UTF-8 v naší binární struktuře, a proto je při zabalení na konci doplněn bajty `0x00`. Při získávání záznamu bude tedy proměnná `contact_data` obsahovat n-tici se dvěma prvky `(b'Eva Drobn\xc3\xa1\x00\x00\x00\x00', 987)`. Pro získání jména musíme UTF-8 dekódovat, čímž získáme řetězec ve znakové sadě Unicode, a oříznout vyplňovací bajty `0x00`.

Nyní již máme představu o tom, jak se tato třída používá, takže se můžeme pustit do prohlídky kódu. Třída `BinaryRecordFile.BinaryRecordFile` se nachází v souboru `BinaryRecordFile.py`, který po obvyklém úvodu začíná definicemi několika soukromých bajtových hodnot:

```
_DELETED = b"\x01"
_OKAY = b"\x02"
```

Každý záznam začíná „stavovým“ bajtem, který je buď `_DELETED`, nebo `_OKAY` (nebo `b"\x00"` v případě prázdných záznamů).

Zde je řádek s příkazem `class` a inicializační metoda:

```
class BinaryRecordFile:

    def __init__(self, filename, record_size, auto_flush=True):
        self.__record_size = record_size + 1
        mode = "w+b" if not os.path.exists(filename) else "r+b"
        self.__fh = open(filename, mode)
        self.auto_flush = auto_flush
```

Existují dvě odlišné velikosti záznamu. Velikost `BinaryRecordFile.record_size` nastavuje uživatel a jedná se o velikost záznamu z pohledu uživatele. Soukromý atribut `BinaryRecordFile.__record_size` představuje skutečnou velikost záznamu a obsahuje stavový bajt.

Musíme myslet na to, abychom soubor, v případě že již existuje, při jeho otevření nezkrátili (použitím režim `"r+b"`) a abychom jej vytvořili, pokud zatím neexistuje (použitím režimu `"w+b"`). Znaménko `"+"` v řetězci definujícím režim označuje čtení a zápis. Má-li logická proměnná `BinaryRecordFile.auto_flush` hodnotu `True`, tak se před každým čtením a po každém zápisu vynutí zápis souboru na disk.

```
@property
def record_size(self):
    return self.__record_size - 1

@property
def name(self):
    return self.__fh.name

def flush(self):
    self.__fh.flush()

def close(self):
    self.__fh.close()
```

Z velikosti záznamu a názvu souboru jsme udělali vlastnosti určené pouze pro čtení. Velikost záznamu hlášená uživateli odpovídá velikosti, kterou uživatel požadoval, a shoduje se s velikostí jeho záznamu. Metody pro vynucený zápis na disk a uzavření souboru jednoduše předávají činnost objektu souboru.

```
def __setitem__(self, index, record):
    assert isinstance(record, (bytes, bytearray)), \
           "vyžadována jsou binární data"
    assert len(record) == self.record_size, (
        "záznam musí mít přesně {0} bajtů".format(
        self.record_size))
    self.__fh.seek(index * self.__record_size)
    self.__fh.write(_OKAY)
    self.__fh.write(record)
    if self.auto_flush:
        self.__fh.flush()
```

Tato metoda podporuje syntaxi *`brf[i] = data`*, kde *`brf`* je binární soubor se záznamy, *`i`* je indexová pozice záznamu a *`data`* bajtový řetězec. Všimněte si, že záznam musí mít stejnou velikost jako velikost stanovená při vytvoření tohoto binární souboru se záznamy.

Jsou-li argumenty v pořádku, posuneme ukazatel pozice v souboru na první bajt záznamu. Všimněte si, že zde používáme skutečnou velikost záznamu, což znamená, že počítáme se stavovým bajtem. Metoda `seek()` přesune ve výchozím stavu ukazatel souboru na absolutní bajtovou pozici. Druhý argument lze použít k relativnímu přesunutí vzhledem k aktuální pozici nebo konci souboru. (Atributy a metody poskytované objekty souboru jsou uvedeny v tabulce 7.2.)

Tabulka 7.2: Atributy a metody objektu souboru

Syntaxe	Popis
f.close()	Uzavře objekt souboru `f` a nastaví atribut `f.closed` na hodnotu `True`.
f.closed	Vrátí hodnotu `True`, je-li soubor uzavřen.
f.encoding	Kódování použité pro převod mezi typy `bytes` a `str`.
f.fileno()	Vrátí deskriptor používaného souboru. (Dostupné pouze pro objekty souboru, které mají deskriptory souboru.)
f.flush()	Vynutí zápis objektu souboru `f` na disk.
f.isatty()	Vrátí hodnotu `True`, je-li objekt souboru spojen s konzolou. (Dostupné pouze pro objekty souboru, které ukazují na skutečné soubory.)
f.mode	Režim, s nímž byl objekt souboru `f` otevřen.
f.name	Název souboru objektu souboru `f` (má-li nějaký).
f.newlines	Druh znaků pro nový řádek, které jsou přítomné v textovém souboru `f`.
f.__next__()	Vrátí další řádek z objektu souboru `f`. Ve většině případů se tato metoda používá implicitně, například `for line in f`.
f.peek(*n*)	Vrátí *n* bajtů bez posunu pozice ukazatele v souboru.
f.read(*počet*)	Přečte nejvýše zadaný počet bajtů ze souboru objektu `f`. Není-li počet zadán, načte se každý bajt od aktuální pozice po konec souboru. Při čtení v binárním režimu vrátí objekt typu `bytes` a při čtení v textovém režimu vrátí objekt typu `str`. Pokud již není co číst (konec souboru), vrátí prázdný objekt typu `bytes` či `str`.
f.readable()	Vrátí hodnotu `True`, pokud byl f otevřen pro čtení.
f.readinto(ba)	Načte nejvýše `len(ba)` bajtů do objektu ba typu `bytearray` a vrátí počet přečtených bajtů (což je na konci souboru 0). (Dostupné pouze v binárním režimu.)
f.readline(*počet*)	Přečte další řádek (nebo nejvýše zadaný počet bajtů dosažitelných před znakem `\n`, je-li počet zadán), a to včetně znaku `\n`.
f.readlines(*velikost*)	Přečte všechny řádky až do konce souboru a vrátí je jako seznam. Je-li zadána velikost, pak přečte data přibližně do zadané velikosti bajtů, pokud to daný objekt souboru podporuje.

Syntaxe	Popis
f.seek(*posun*, *odkud*)	Přesune pozici ukazatele v souboru (místo, kde dojde k dalšímu čtení či zápisu) na zadaný *posun*, není-li parametr *odkud* zadán, nebo má hodnotu `os.SEEK_SET`. Přesune ukazatel v souboru na zadaný *posun* (který může být záporný) relativně vzhledem k aktuální pozici, má-li parametr *odkud* hodnotu `os.SEEK_CUR`, nebo relativně vzhledem ke konci souboru, má-li parametr *odkud* hodnotu `os.SEEK_END`. V režimu připojování (`"a"`) probíhá zápis vždy od konce souboru bez ohledu na to, zda se ukazatel v souboru nachází. Hodnoty vrácené metodou `tell()` by se měly používat pro posun pouze v textovém režimu.
f.seekable()	Vrátí hodnotu `True`, pokud `f` podporuje náhodný přístup.
f.tell()	Vrátí aktuální pozici ukazatele v souboru relativně vzhledem k začátku souboru.
f.truncate(*velikost*)	Zkrátí soubor po aktuální pozici ukazatele v souboru nebo na uvedenou *velikost*, je-li zadána.
f.writable()	Vrátí hodnotu `True`, pokud byl objekt souboru `f` otevřen pro zápis.
f.write(s)	Zapíše do souboru objekt `s` typu `bytes` nebo `bytearray`, je-li otevřený v binárním režimu, nebo objekt `s` typu `str`, je-li otevřený v textovém režimu.
f.writelines(*posl*)	Zapíše do souboru zadanou posloupnost objektů (řetězců u textových souborů nebo bajtových řetězců u binárních souborů).

Vzhledem k tomu, že prvek nastavujeme, tak jistě nebyl smazán, a proto zapíšeme stavový bajt `_OKAY`, a poté zapíšeme data binárního záznamu uživatele. Soubor s binárními záznamy neví nic o používané struktuře záznamu a ani se o ni nestará. Stačí, že záznamy mají správnou velikost.

Kontrolu, zda je index ve správném rozsahu, neprovádíme. Je-li index za koncem souboru, záznam se zapíše na správnou pozici a každý bajt mezi dřívějším koncem souboru a novým záznamem se automaticky nastaví na `b"\x00"`. Stavový bajt u takovýchto prázdných záznamů není _OKAY ani _DELETED, a proto je můžeme v případě potřeby snadno rozpoznat.

```
def __getitem__(self, index):
    self.__seek_to_index(index)
    state = self.__fh.read(1)
    if state != _OKAY:
        return None
    return self.__fh.read(self.record_size)
```

Při získávání záznamu existují čtyři možnosti, s nimiž musíme počítat: záznam neexistuje, což znamená, že zadaný index je za koncem, záznam je prázdný, záznam byl vymazán a záznam je v pořádku. Pokud záznam neexistuje, soukromá metoda `__seek_to_index()` vyvolá výjimku `IndexError`. V opačném případě se přesune na bajt, kde záznam začíná, takže můžeme přečíst stavový bajt. Nemá-li hodnotu `_OKAY`, záznam musí být buď prázdný, nebo vymazaný, a proto vrátíme hodnotu `None`. V opačném případě požadovaný záznam přečteme a vrátíme. (Další možností by bylo pro prázdné

a vymazané záznamy vyvolat místo vrácení hodnoty `None` vlastní výjimku, třeba `BlankRecordError` nebo `DeletedRecordError`.)

```
def __seek_to_index(self, index):
    if self.auto_flush:
        self.__fh.flush()
    self.__fh.seek(0, os.SEEK_END)
    end = self.__fh.tell()
    offset = index * self.__record_size
    if offset >= end:
        raise IndexError("na indexové pozice {0} není záznam".format(
                         index))
    self.__fh.seek(offset)
```

Jedná se o podpůrnou soukromou metodu používanou některými metodami pro přesun pozice ukazatele v souboru na první bajt záznamu se zadanou indexovou pozicí. Začínáme kontrolou, zda je zadaný index v platném rozsahu, což provádíme přesunutím na konec souboru (0 bajtů od konce), kde pomocí metody `tell()` získáme bajtovou pozici, do níž jsme se přesunuli.* Je-li posun záznamu (indexová pozice × skutečná velikost záznamu) na konci nebo za koncem, pak je index mimo rozsah, a proto vyvoláme vhodnou výjimku. V opačném případě se přesuneme na pozici záznamu, který je tak připraven pro čtení či zápis.

```
def __delitem__(self, index):
    self.__seek_to_index(index)
    state = self.__fh.read(1)
    if state != _OKAY:
        return
    self.__fh.seek(index * self.__record_size)
    self.__fh.write(_DELETED)
    if self.auto_flush:
        self.__fh.flush()
```

Nejdříve přesuneme ukazatel pozice v souboru na správné místo. Je-li index v platném rozsahu (tj. nedošlo k výjimce `IndexError`) a není-li záznam prázdný nebo vymazaný, vymažeme záznam přepsáním jeho stavového bajtu hodnotou `_DELETED`.

```
def undelete(self, index):
    self.__seek_to_index(index)
    state = self.__fh.read(1)
    if state == _DELETED:
        self.__fh.seek(index * self.__record_size)
        self.__fh.write(_OKAY)
        if self.auto_flush:
```

* Python 3.0 a 3.1 má konstanty `os.SEEK_SET`, `os.SEEK_CUR` a `os.SEEK_END`. Python 3.1 má navíc tyto konstanty i ve svém modulu `io` (např. `io.SEEK_SET`). **3.x**

```
                self.__fh.flush()
            return True
        return False
```

Tato metoda začíná vyhledáním záznamu a přečtením jeho stavového bajtu. Je-li záznam vymazán, přepíšeme jeho stavový bajt hodnotou `_OKAY` a vrátíme volajícímu hodnotu `True` signalizující úspěšně provedenou operaci. V opačném případě (u prázdných nebo nevymazaných záznamů) vrátíme hodnotu `False`.

```
    def __len__(self):
        if self.auto_flush:
            self.__fh.flush()
        self.__fh.seek(0, os.SEEK_END)
        end = self.__fh.tell()
        return end // self.__record_size
```

Tato metoda hlásí, kolik záznamů se nachází v binárním souboru se záznamy. Tento údaj počítáme podělením koncové bajtové pozice (tj. počtu bajtů v souboru) velikostí záznamu.

Nyní jsme probrali veškerou základní funkčnost nabízenou třídou `BinaryRecordFile.BinaryRecordFile`. Je tu ještě jedna záležitost, kterou bychom měli uvážit: stlačení souboru pro eliminaci prázdných a vymazaných záznamů. To můžeme v podstatě provést dvěma způsoby. První spočívá v přepsání prázdných či vymazaných záznamů těmi, které mají vyšší indexovou pozici, čímž zaplníme všechny díry, a ve zkrácení souboru, jsou-li nějaké prázdné či vymazané záznamy na konci. Takto pracuje metoda `inplace_compact()`. Další možností je zkopírovat neprázdné a nevymazané záznamy do dočasného souboru a ten pak přejmenovat na soubor původní. Použití dočasného souboru je zvláště výhodné, pokud chceme rovněž provést zálohu. Tímto způsobem pracuje metoda `compact()`.

Začneme metodou `inplace_compact()`, kterou si rozdělíme na dvě části.

```
    def inplace_compact(self):
        index = 0
        length = len(self)
        while index < length:
            self.__seek_to_index(index)
            state = self.__fh.read(1)
            if state != _OKAY:
                for next in range(index + 1, length):
                    self.__seek_to_index(next)
                    state = self.__fh.read(1)
                    if state == _OKAY:
                        self[index] = self[next]
                        del self[next]
                        break
                else:
                    break
            index += 1
```

Procházíme všechny záznamy a postupně čteme stav každého z nich. Narazíme-li na prázdný či vymazaný záznam, vyhledáme v souboru následující nevymazaný záznam. Pokud jej nalezneme, nahradíme jím prázdný či vymazaný záznam a pak jej smažeme. Pokud jej nenalezneme, tak přerušíme cyklus, protože jsme již prošli všechny neprázdné a nevymazané záznamy.

```
        self.__seek_to_index(0)
        state = self.__fh.read(1)
        if state != _OKAY:
            self.__fh.truncate(0)
        else:
            limit = None
            for index in range(len(self) - 1, 0, -1):
                self.__seek_to_index(index)
                state = self.__fh.read(1)
                if state != _OKAY:
                    limit = index
                else:
                    break
            if limit is not None:
                self.__fh.truncate(limit * self.__record_size)
        self.__fh.flush()
```

Je-li první záznam prázdný či vymazaný, pak jsou nutně všechny záznamy prázdné či vymazané, protože předchozí kód přesunul všechny neprázdné a nevymazané záznamy na začátek souboru a prázdné a vymazané na konec. V takovém případě můžeme jednoduše zkrátit soubor na 0 bajtů.

Pokud existuje alespoň jeden neprázdný a nevymazaný záznam, pak začneme procházet záznamy od posledního zpět k prvnímu, protože víme, že prázdné a vymazané záznamy jsou přesunuty na konec. Proměnná `limit` je nastavena na poslední prázdný či vymazaný záznam (nebo ponechána s hodnotou `None`, pokud žádné prázdní či vymazané záznamy neexistují) a podle ní pak soubor zkrátíme.

Kromě stlačení souboru na místě je možné stlačit i jeho kopii v jiném souboru, což je užitečné, pokud chceme mít k dispozici zálohu. Tento způsob implementujeme v metodě `compact()`.

```
    def compact(self, keep_backup=False):
        compactfile = self.__fh.name + ".$$$"
        backupfile = self.__fh.name + ".bak"
        self.__fh.flush()
        self.__fh.seek(0)
        fh = open(compactfile, "wb")
        while True:
            data = self.__fh.read(self.__record_size)
            if not data:
                break
            if data[:1] == _OKAY:
                fh.write(data)
        fh.close()
```

```
        self.__fh.close()

        os.rename(self.__fh.name, backupfile)
        os.rename(compactfile, self.__fh.name)
        if not keep_backup:
            os.remove(backupfile)
        self.__fh = open(self.__fh.name, "r+b")
```

Tato metoda vytváří dva soubory, stlačenou verzi a záložní kopii původního souboru. Jméno stlačeného souboru začíná stejně jako jméno původního souboru, na jeho konec jsme ale připojili příponu `.$$$`. Podobně je tomu i s názvem záložního souboru, k němuž jsme připojili příponu `.bak`. Stávající soubor čteme záznam po záznamu, přičemž záznamy, které jsou neprázdné a nevymazané, zapisujeme do stlačeného souboru. (Všimněte si, že zapisujeme skutečné záznamy, což znamená, že vždy zapisujeme stavový bajt plus uživatelský záznam.)

Panel „Datové typy bytes a bytearray" ➤ 286

Řádek `if data[:1] == _OKAY:` je docela zákeřný. Oba objekty `data` a `_OKAY` jsou typu `bytes`, přičemž my potřebujeme porovnat první bajt objektu `data` s (1bajtovým) objektem `_OKAY`. Pokud vezmeme řez objektu `bytes`, obdržíme objekt typu `bytes`. Pokud ale vezmeme jediný bajt (např. `data[0]`), pak obdržíme objekt typu `int` obsahující hodnotu daného bajtu. Proto zde porovnáváme 1bajtový řez objektu `data` (jeho první bajt, což je stavový bajt) s 1bajtovým objektem `_OKAY`. (Další možností by bylo napsat podmínku `if data[0] == _OKAY[0]:`, která by porovnala dva objekty typu `int`.)

Na konci přejmenujeme původní soubor na záložní a stlačený soubor na původní. Poté záložní soubor odstraníme, má-li parametr `keep_backup` hodnotu `False` (což je výchozí hodnota). Na závěr stlačený soubor (který má nyní jméno původního souboru) otevřeme, aby byl připraven pro čtení či zápis.

Třída `BinaryRecordFile.BinaryRecordFile` sice pracuje na docela nízké úrovni, může však sloužit jako základ třídy pracující na vyšší úrovni, která potřebuje náhodný přístup do souborů se záznamy fixní délky. Příklad takovéto třídy si ukážeme v následujícím oddílu.

Příklad: Třídy modulu BikeStock

Modul `BikeStock` poskytuje pomocí třídy `BinaryRecordFile.BinaryRecordFile` třídu pro jednoduché řízení skaldu. Skladovými položkami jsou jízdní kola, každé reprezentované instancí třídy `BikeStock.Bike`, přičemž celý sklad kol je uchováván v instanci třídy `BikeStock.BikeStock`. Třída `BikeStock.BikeStock` agreguje slovník, jehož klíči jsou identifikační kódy kol a indexové pozice záznamů v souboru reprezentovaném instancí třídy `BinaryRecordFile.BinaryRecordFile`. Zde je krátký příklad, který demonstruje způsob použití těchto tříd:

```
bicycles = BikeStock.BikeStock(bike_file)
value = 0.0
for bike in bicycles:
    value += bike.value
bicycles.increase_stock("GEKKO", 2)
for bike in bicycles:
    if bike.identity.startswith("B4U"):
        if not bicycles.increase_stock(bike.identity, 1):
            print("neúspěšný pohyb skladových zásob pro", bike.identity)
```

V tomto úryvku otevíráme soubor s údaji o skladových zásobách kol a průchodem všech v něm obsažených záznamů o jízdních kolech hledáme celkovou hodnotu (součet cen × množství) uskladněných kol. Poté zvyšujeme počet kol „GEKKO“ na skladě o dvě a zvyšujeme skladové zásoby všech kol, jejichž identifikační kód začíná „B4U“. Všechny tyto akce se odehrávají na disku, takže jakýkoliv další proces, který čte soubor s údaji o skladových zásobách kol, obdrží vždy ta nejaktuálnější data.

Přestože třída `BinaryRecordFile.BinaryRecordFile` funguje na principu indexů, třída `BikeStock.BikeStock` pracuje na principu identifikačních kódu kol. K tomuto účelu má instance třídy `BikeStock.BikeStock` slovník, který páruje identifikační kódy kol s indexy.

Začneme pohledem na řádek s příkazem `class` a inicializační metodu třídy `BikeStock.Bike`, pak se podíváme na několik vybraných metod třídy `BikeStock.BikeStock` a nakonec si ukážeme kód, který poskytuje most mezi objekty typu `BikeStock.Bike` a binárními záznamy určenými pro jejich reprezentaci v instanci třídy `BinaryRecordFile.BinaryRecordFile`. (Veškerý kód naleznete v souboru `BikeStock.py`.)

```
class Bike:

    def __init__(self, identity, name, quantity, price):
        assert len(identity) > 3, ("neplatná identita kola '{0}'"
                                   .format(identity))
        self.__identity = identity
        self.name = name
        self.quantity = quantity
        self.price = price
```

Všechny atributy kola jsou k dispozici jako vlastnosti – identifikační kód kola (`self.__identity`) jako vlastnost `Bike.identity` určená pouze pro čtení a ostatní jako vlastnosti pro čtení i zápis s příkazy `assert` pro validaci. Kromě toho vlastnost `Bike.value` určená pouze pro čtení vrací množství vynásobené cenou. (Implementaci vlastností si zde neukážeme, protože jsme již dříve viděli podobný kód.)

Třída `BikeStock.BikeStock` poskytuje své vlastní metody pro manipulaci s objekty kol a ty zase používají zapisovatelné vlastnosti těchto objektů.

```
class BikeStock:

    def __init__(self, filename):
        self.__file = BinaryRecordFile.BinaryRecordFile(filename,
                                                _BIKE_STRUCT.size)
        self.__index_from_identity = {}
        for index in range(len(self.__file)):
            record = self.__file[index]
            if record is not None:
                bike = _bike_from_record(record)
                self.__index_from_identity[bike.identity] = index
```

Třída `BikeStock.BikeStock` je třídou implementující naši vlastní kolekci, která agreguje binární soubor se záznamy (`self.__file`) a slovník (`self.__index_from_identity`), jehož klíči jsou identifikační kódy kol a hodnotami indexové pozice záznamů.

Jakmile máme soubor otevřený (a vytvořený, pokud zatím neexistoval), procházíme jeho obsah (má-li nějaký). Načteme záznam o každém kole ve formě objektu typu `bytes`, který soukromou metodou `_bike_from_record()` převedeme na objekt typu `BikeStock.Bike`. Potom přidáme identitu a index kola do slovníku `self.__index_from_identity`.

```
def append(self, bike):
    index = len(self.__file)
    self.__file[index] = _record_from_bike(bike)
    self.__index_from_identity[bike.identity] = index
```

Přidání nového kola je záležitostí vyhledání vhodné indexové pozice a nastavení záznamu na této pozici na binární reprezentaci kola. Dále se postaráme o aktualizaci slovníku `self.__index_from_identity`.

```
def __delitem__(self, identity):
    del self.__file[self.__index_from_identity[identity]]
```

Vymazání záznamu o kolu je snadné. Stačí jen z jeho identity vyhledat indexovou pozici jeho záznamu a záznam na této indexové pozici vymazat. V případě třídy `BikeStock.BikeStock` nevyužíváme schopnosti třídy `BinaryRecordFile.BinaryRecordFile` zrušit vymazání záznamu.

```
def __getitem__(self, identity):
    record = self.__file[self.__index_from_identity[identity]]
    return None if record is None else _bike_from_record(record)
```

Záznamy o kolech se získávají podle identifikačního kódu kola. Pokud se zadaný identifikační kód ve slovníku `self.__index_from_identity` nenachází, vyvolá se výjimka `KeyError`, a pokud je záznam prázdný či vymazaný, vrátí třída `BinaryRecordFile.BinaryRecordFile` hodnotu `None`. Pokud ale záznam obdržíme, vrátíme jej jako objekt typu `BikeStock.Bike`.

```
def __change_stock(self, identity, amount):
    index = self.__index_from_identity[identity]
    record = self.__file[index]
    if record is None:
        return False
    bike = _bike_from_record(record)
    bike.quantity += amount
    self.__file[index] = _record_from_bike(bike)
    return True
increase_stock = (lambda self, identity, amount:
                          self.__change_stock(identity, amount))
decrease_stock = (lambda self, identity, amount:
                          self.__change_stock(identity, -amount))
```

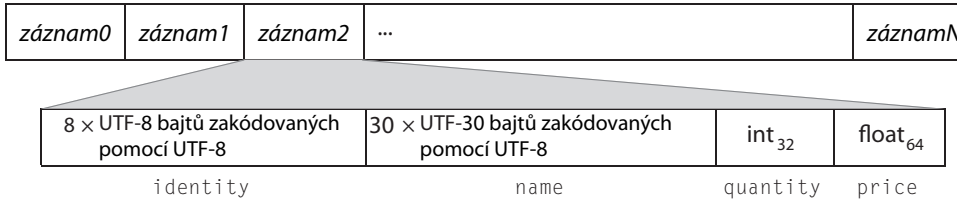
Soukromá metoda `__change_stock()` poskytuje implementaci pro metody `increase_stock()` a `decrease_stock()`. Vyhledáme indexovou pozici kola a vezmeme holý binární záznam. Pak tato data převedeme na objekt typu `BikeStock.Bike`, aplikujeme na něj požadovanou změnu a poté záznam v souboru přepíšeme binární reprezentací aktualizovaného objektu. (Dále je zde metoda `__change_bike()`,

která poskytuje implementaci pro metody `change_name()` a `change_price()`. Žádnou z nich si zde ale ukazovat nebudeme, protože jsou velmi podobné tomu, co jsme již viděli.)

```
    def __iter__(self):
        for index in range(len(self.__file)):
            record = self.__file[index]
            if record is not None:
                yield _bike_from_record(record)
```

Tato metoda zajišťuje, aby bylo možné objekty typu `BikeStock.BikeStock` procházet stejně jako seznam s tím, že se v každé iteraci vrátí objekt typu `BikeStock.Bike`, přičemž prázdné a vymazané záznamy se přeskočí.

Soukromé funkce `_bike_from_record()` a `_record_from_bike()` izolují binární reprezentaci třídy `BikeStock.Bike` od třídy `BikeStock.BikeStock`, která uchovává kolekci objektů typu `BikeStock.Bike`. Logická struktura souboru se záznamy o kole je znázorněna na obrázku 7.4. Fyzická struktura je malinko odlišná, protože před každým záznamem je umístěn stavový bajt.



Obrázek 7.4: Logická struktura souboru se záznamy o kolech

```
_BIKE_STRUCT = struct.Struct("<8s30sid")

def _bike_from_record(record):
    ID, NAME, QUANTITY, PRICE = range(4)
    parts = list(_BIKE_STRUCT.unpack(record))
    parts[ID] = parts[ID].decode("utf8").rstrip("\x00")
    parts[NAME] = parts[NAME].decode("utf8").rstrip("\x00")
    return Bike(*parts)

def _record_from_bike(bike):
    return _BIKE_STRUCT.pack(bike.identity.encode("utf8"),
                             bike.name.encode("utf8"),
                             bike.quantity, bike.price)
```

Při převodu binárního záznamu na objekt typu `BikeStock.Bike` nejdříve převedeme n-tici vrácenou funkcí `unpack()` na seznam. Díky tomu můžeme upravit jednotlivé prvky, což v tomto případě znamená převést bajty zakódované pomocí kódování UTF-8 na řetězce bez výplňových bajtů `0x00`. Poté použijeme operátor rozbalení posloupnosti (*) pro předání údajů inicializační metodě třídy `BikeStock.Bike`. Zabalení dat je mnohem jednodušší. Stačí jen zajistit, aby se řetězce zakódovaly jako bajty s kódováním UTF-8.

Rozbalení posloupnosti ➢ 113

U moderních desktopových systémů klesá potřeba aplikačních programů využívajících binární data s náhodným přístupem s růstem pamětí RAM a rychlosti disků. Je-li přece jen tato funkčnost zapotřebí, pak je často snazší sáhnout po souboru DBM nebo databázi SQL. Nicméně existují systémy, v nichž může být výše uvedená funkčnost užitečná. Patří mezi ně například vložené (embedded) systémy a další systémy s omezenými prostředky.

Shrnutí

V této lekci jsme si ukázali nejčastěji používané techniky pro ukládání kolekcí dat do souborů a pro jejich načítání ze souborů. Viděli jsme, jak snadno se používají naložené objekty a jak můžeme pracovat s komprimovanými i nekomprimovanými soubory bez toho, abychom dopředu věděli, zda je daný soubor komprimován.

Ukázali jsme si, že zapisování a čtení binárních dat vyžaduje zvláštní péči, a dále jsme viděli, že kód může být docela dlouhý, pokud potřebujeme zpracovávat řetězce proměnlivé délky. Kromě toho jsme se dozvěděli, že používání binárních souborů obvykle vede k nejmenší možné velikosti souborů a k nejrychlejším operacím zápisu a čtení dat. Dále jsme se dozvěděli, že je důležité pro identifikování našeho typu souboru používat nějaké magické číslo a že použití čísla verze je praktické pro budoucí změny formátu.

V této lekci jsme viděli, že holý text představuje formát, který se uživatelům nejsnadněji čte, a že pokud jsou data dobře strukturovaná, pak může být docela přímočaré vytvářet další nástroje pro jejich zpracování. Analyzování textových dat však může být složité. Ukázali jsme si, jak číst textová data ručně i pomocí regulárních výrazů.

XML je velice populární formát pro výměnu dat, přičemž je obecně užitečné mít možnost jej alespoň importovat a exportovat, a to i v případě, kdy je běžný formát binární či textový. Ukázali jsme si, jak zapisovat kód jazyka XML ručně (včetně správného zakódování hodnot atributů a textových dat) a jak jej zapisovat s použitím stromu elementů a modelu DOM. Naučili jsme také, jak analyzovat XML pomocí analyzátorů, které poskytuje standardní knihovna Pythonu v rámci stromu elementů, modelu DOM a rozhraní SAX.

V poslední části této lekce jsme viděli, jak vytvořit generickou třídu pro práci s binárními soubory s náhodným přístupem, které uchovávají záznamy fixní velikosti, a potom jsme si ukázali, jak tuto generickou třídu použít v určitém kontextu.

Touto lekcí jsme se dostali na konec tématu o základech programování v jazyku Python. Je možné nyní přestat číst a na základě všeho, co jste se dosud naučili, začít v Pythonu psát skvělé programy. Bylo by ale škoda zastavit se právě nyní, protože Python nabízí mnohem více, počínaje parádními technikami, které dokážou zkrátit a zjednodušit kód, a konče šokujícími pokročilými prostředky, o kterých je přinejmenším dobré vědět, i kdyby nebyly tak často potřeba. V následující lekci budeme pokračovat ve výkladu procedurálního a objektově orientovaného programování a vyzkoušíme si též funkcionální programování. Pak se v dalších lekcích zaměříme na širší programovací techniky, mezi něž patří práce s více vlákny, práce v síti, databázové programování, regulární výrazy a programování rozhraní GUI (Graphical User Interface – grafické uživatelské rozhraní).

Cvičení

První cvičení spočívá ve vytvoření modulu pro binární soubory se záznamy, který bude jednodušší, než ten, který jsme si ukázali v této lekci – velikost jeho záznamů bude naprosto stejná jako velikost zadaná uživatelem. Ve druhém cvičení upravíte modul `BikeStock` tak, aby používal náš nový modul pro binární soubory se záznamy. Ve třetím cvičení vytvoříte program úplně od začátku. Práce se soubory je docela přímočará, avšak určité aspekty formátování výstupu jsou docela náročné.

1. Vytvořte novou, jednodušší verzi modulu `BinaryRecordFile` – takovou, která nepoužívá stavový bajt. V této verzi je velikost záznamu zadaná uživatelem stejná jako skutečně používaná velikost záznamu. Nové záznamy se musejí přidávat pomocí nové metody `append()`, která jednoduše přesune ukazatel v souboru na konec a zapíše zadaný záznam. Metoda `__setitem__()` by měla umožnit pouze nahrazení stávajících záznamů, což lze snadno provést například s využitím metody `__seek_to_index()`. Bez stavového bajtu je metoda `__getitem__()` zredukována na pouhé tři řádky. Metodu `__delitem__()` bude nutné zcela přepsat, protože musí pro zaplnění mezery přesunout všechny záznamy. Metodu `undelete()` je nutné odstranit, protože není podporována. Odstranit se musí také metody `compact()` a `inplace_compact()`, protože již nejsou potřeba.

   Změny zahrnují všehovšudy 20 nových či změněných řádků a nejméně 60 vymazaných řádků ve srovnání s původním programem, nepočítáme-li dokumentační testy. Řešení je k dispozici v souboru `BinaryRecordFile_ans.py`.

2. Jakmile si budete jisti, že vaše jednodušší třída `BinaryRecordFile` funguje, zkopírujte si soubor `BikeStock.py` a upravte jej tak, aby pracovat s vaší třídou `BinaryRecordFile`. To zahrnuje změnu jen hrstky řádků. Řešení je k dispozici v souboru `BikeStock_ans.py`.

3. Ladění binárních formátů může být obtížné. Při této činnosti nám může pomoci nástroj, který dokáže vypsat obsah binárního souboru v šestnáctkovém formátu. Vytvořte program, který do konzoly vypíše následující text nápovědy:

```
Usage: xdump.py [volby] soubor1 [soubor2 [... souborN]]

Options:
  -h, --help            show this help message and exit
  -b BLOCKSIZE, --blocksize=BLOCKSIZE
                        velikost bloku (8..80) [výchozí: 16]
  -d, --decimal         desítková čísla v bloku [výchozí: šestnáctková]
  -e ENCODING, --encoding=ENCODING
                        kódování (ASCII..UTF-32) [výchozí: UTF-8]
```

   Máme-li třídu `BinaryRecordFile`, která ukládá záznamy se strukturou `"<i10s"` (formát bajtů Little Endian, 4bajtové celé číslo se znaménkem, bajtový řetězec délky 10 bajtů), pak můžeme při použití tohoto programu s blokem nastaveným na velikost odpovídající jednomu záznamu (15 bajtů včetně stavového bajtu) získat jasnou představu o tom, co se v souboru nachází. Například:

```
xdump.py -b15 test.dat
Blok     Bajty                           Znaky v UTF-8
-------- ------------------------------- ----------------
```

```
00000000 02000000 00416C70 68610000 000000 .....Alpha.....
00000001 01140000 00427261 766F0000 000000 .....Bravo.....
00000002 02280000 00436861 726C6965 000000 .(...Charlie...
00000003 023C0000 0044656C 74610000 000000 .<...Delta.....
```

Každý bajt je reprezentován dvojciferným šestnáctkovým číslem. Mezery mezi každou skupinou čtyř bajtů (tj. mezi každou skupinou osmi šestnáctkových číslic) slouží čistě pro zlepšení čitelnosti. Zde můžeme vidět, že druhý záznam („Bravo“) byl vymazán, protože jeho stavový bajt je `0x01`, a ne `0x02`, což je hodnota, kterou používáme pro označení neprázdných a nevymazaných záznamů.

Pro zpracování voleb příkazového řádku použijte modul `optparse`. (Specifikací volby „type“ můžeme zařídit, aby se modul `optparse` postaral u velikosti bloku o převod řetězce na číslo.) Může být docela obtížně zajistit, aby se záhlaví správně srovnalo pro poslední blok, nezapomeňte tedy testovat s různými velikostmi bloku (např. 8, 9, 10, ..., 40). Dále nezapomeňte, že v souborech proměnné délky může být poslední blok krátký. Jak je patrné z ukázky, místo netisknutelných znaků použijte tečku.

Program lze napsat na méně než 70 řádků rozdělených do dvou funkcí. Řešení je k dispozici v souboru `xdump.py`.

LEKCE 8

Pokročilé techniky programování

V této lekci:

- Další techniky procedurálního programování
- Další techniky objektově orientovaného programování
- Funkcionální styl programování

V této lekci se podíváme na pestrou škálu různých programovacích technik a seznámíme se s mnoha dalšími, často pokročilejšími syntaxemi jazyka Python. Některá témata probíraná v této lekci jsou docela obtížná, mějte však na paměti, že ty nejpokročilejší techniky jsou jen zřídka potřebné a že stačí, když si napoprvé text jen letmo projdete, abyste získali představu, co vše lze provádět, a pak se k němu vrátíte až v momentě, kdy to bude potřeba.

V první části této lekce se ponoříme hlouběji do procedurálních prvků jazyka Python. Na začátku si ukážeme, jak dříve osvojené znalosti použít novým způsobem, a poté se vrátíme k tématu generátorů, kterého jsme se jen letmo dotkli v lekci 6. Dále se seznámíme s dynamickým programováním, což představuje načítání modulů podle jména za běhu programu a provádění libovolného kódu za běhu programu. Vrátíme se též k tématu lokálních (vnořených) funkcí a kromě něj se budeme věnovat také klíčovému slovu `nonlocal` a rekurzivním funkcím. Dříve jsme viděli, jak používat předdefinované dekorátory Pythonu a v této části se naučíme, jak vytvářet své vlastní dekorátory. První část uzavřeme výkladem o anotacích funkcí.

Ve druhé části se budeme věnovat jen novým věcem souvisejícím s objektově orientovaným programováním. Začneme seznámením s mechanizmem `__slots__`, který slouží k minimalizaci paměti používané každým objektem. Pak si ukážeme, jak přistupovat k atributům bez použití vlastností. Představíme si také funktory (objekty, které lze volat jako funkce) a správce kontextu, které se používají společně s klíčovým slovem `with` a v řadě případů (např. práce se soubory) lze pomocí nich nahradit konstrukty `try ... except ... finally` jednoduššími konstrukty `try ... except`. V této části si též ukážeme, jak vytvářet vlastní správce kontextu, a podíváme se na pokročilé prvky objektové orientace, mezi něž patří dekorátory tříd, abstraktní bázové třídy, vícenásobná dědičnost a metatřídy.

Ve třetí části se seznámíme se základními principy funkcionálního programování a představíme si několik užitečných funkcí z modulů `functools`, `itertools` a `operator`. V této části si též ukážeme, jak lze pomocí částečné aplikace funkce zjednodušit kód a jak vytvářet a používat korutiny.

Všechny předchozí lekce dohromady nás vybavily jistou „standardní sadou nástrojů pro jazyk Python". V této lekci vezmeme vše, co jsme se dosud naučili, a uděláme z toho „luxusní sadu nástrojů pro jazyk Python", obsahující všechny původní nástroje (techniky a syntaxe) plus řadu nových, díky nimž budeme moci snadněji psát kratší a efektivnější kód. Některé z těchto nástrojů lze nahradit jinými. Kupříkladu některé činnosti lze provést buď pomocí dekorátoru třídy, nebo pomocí metatřídy. Jiné nástroje, jako jsou například deskriptory, je zase možné použít více způsoby pro dosažení různých výsledků. Některé z nástrojů, které zde budeme probírat (například správce kontextu), budeme používat neustále a jiné zůstanou po ruce připraveny pro ty situace, v nichž představují perfektní řešení.

Další techniky procedurálního programování

V této části se sice budeme povětšinou věnovat dalších prvkům souvisejícím s procedurálním programováním a s funkcemi, avšak první oddíl je odlišný, protože prezentuje užitečné programovací techniky založené na tom, co již známe, takže se v něm s žádnou novou syntaxi neseznámíme.

Větvení pomocí slovníků

Jak jsme si řekli již dříve, funkce jsou v Pythonu objekty jako cokoliv jiného a název funkce je odkazem na objekt, který ukazuje na tuto funkci. Pokud napíšeme název funkce bez závorek, pak Python ví, že chceme odkaz na objekt, který můžeme předávat dál jako jakýkoliv jiný odkaz na objekt. Této skutečnosti můžeme využít k nahrazení příkazů `if`, které mají spoustu klauzulí `elif`, k čemuž nám stačí zavolání jediné funkce.

V lekci 12 budeme probírat interaktivní konzolový program s názvem `dvds-dbm.py`, který má následující nabídku:

```
(P)řidat  (U)pravit  (V)ypsat  (O)dstranit  (I)mportovat  (E)xportovat  (K)onec
```

Tento program má funkci, která přečte volbu od uživatele a vrátí pouze platnou volbu, což je v tomto případě jedno z písmen „p“, „u“, „v“, „o“, „i“, „e“ a „k“. Zde jsou dva ekvivalentní úryvky kódu pro zavolání relevantní funkce na základě volby uživatele:

```
if action == "a":
    add_dvd(db)
elif action == "e":
    edit_dvd(db)
elif action == "l":
    list_dvds(db)
elif action == "r":
    remove_dvd(db)
elif action == "i":
    import_(db)
elif action == "x":
    export(db)
elif action == "q":
    quit(db)
```

```
functions = dict(a=add_dvd,
    e=edit_dvd, l=list_dvds,
    r=remove_dvd, i=import_,
    x=export, q=quit)
functions[action](db)
```

Volba je uchovávána jako jednoznakový řetězec v proměnné `action` a databáze, která se má použít, je v proměnné `db`. Funkce `import_()` má na konci podtržítko, aby byl její název odlišný od vestavěného příkazu import.

V úryvku kódu na pravé straně vytváříme slovník, jehož klíči jsou platné volby nabídky a hodnotami odkazy na funkce. Ve druhém příkazu bereme odkaz funkci odpovídající zadané akci a pomocí operátoru volání `()` zavoláme příslušnou funkci, přičemž jí v našem příkladu předáme argument `db`. Kód na pravé straně je ve srovnání s kódem vlevo nejen kratší, ale lze jej také škálovat (přidávat mnohem více prvků do slovníku) bez vlivu na výkon, což nelze říci o kódu na levé straně, jehož rychlost závisí na tom, kolik podmínek je potřeba otestovat před `elif` nalezením příslušného volání funkce.

Program `convert-incidents.py` z předchozí lekce používá tuto techniku ve své metodě `import_()`, což je patrné na níže uvedeném úryvku z této metody:

```
call = {(".aix", "dom"): self.import_xml_dom,
        (".aix", "etree"): self.import_xml_etree,
        (".aix", "sax"): self.import_xml_sax,
```

```
                (".ait", "manual"): self.import_text_manual,
                (".ait", "regex"): self.import_text_regex,
                (".aib", None): self.import_binary,
                (".aip", None): self.import_pickle}
        result = call[extension, reader](filename)
```

Celá metoda má 13 řádků. Parametr `extension` se počítá v metodě a parametr `reader` metoda obdrží při svém volání. Klíči slovníku jsou n-tice se dvěma prvky a hodnotami metody. Pokud bychom použili příkazy `if`, byl by kód dlouhý 22 řádků a nebylo by možné jej dobře škálovat.

Generátorové výrazy a funkce

Generátorové funkce ➤ 273

V lekci 6 jsme se seznámili s generátorovými funkcemi a metodami. Vytvářet můžeme také generátorové výrazy, které jsou ze syntaktického hlediska téměř totožné se seznamovými komprehenzemi s tím rozdílem, že nejsou uzavřené do hranatých, ale do kulatých závorek. Zde jsou jejich syntaxe:

```
(výraz for prvek in iterovatelný_objekt)
(výraz for prvek in iterovatelný_objekt if podmínka)
```

V předchozí lekci jsme pomocí výrazů s příkazem `yield` vytvořili několik iterátorových metod. Zde jsou dva ekvivalentní úryvky kódu, na nichž je patrné, jak lze jednoduchý cyklus `for ... in` obsahující výraz `yield` zapsat jako generátor:

```
def items_in_key_order(d):
    for key in sorted(d):
        yield key, d[key]
```

```
def items_in_key_order(d):
    return ((key, d[key])
            for key in sorted(d))
```

Obě funkce vracejí generátor, který pro zadaný slovník vytváří seznam prvků klíč-hodnota. Pokud potřebujeme všechny prvky naráz, pak můžeme generátor vrácený těmito funkcemi předat funkci `list()` nebo `tuple()`, jinak jej můžeme procházet a získávat prvky postupně, dle aktuální potřeby.

Generátory poskytují prostředky k provádění líného vyhodnocování, což znamená, že počítají pouze hodnoty, které jsou skutečně potřeba. To může být mnohem efektivnější než třeba počítání velmi rozsáhlého seznamu najednou. Některé generátory vytvářejí tolik hodnot, o kolik si řekneme, aniž by mělY nějaký horní limit. Například:

```
def quarters(next_quarter=0.0):
    while True:
        yield next_quarter
        next_quarter += 0.25
```

Tato funkce bude vracet `0.0`, `0.25`, `0.5` a tak pořád dál až donekonečna. Tento generátor bychom mohli využít například takto:

```
result = []
for x in quarters():
    result.append(x)
    if x >= 1.0:
        break
```

Příkaz `break` je naprosto zásadní, protože bez něj by cyklus `for ... in` nikdy neskončil. Seznam `result` na konci obsahuje `[0.0, 0.25, 0.5, 0.75, 1.0]`.

Při každém zavolání funkce `quarters()` obdržíme generátor, který začíná na hodnotě `0.0` a postupně přičítá `0.25`. Jak ale aktuální hodnotu generátoru resetovat? Generátoru je možné předat hodnotu, jak ukazuje tato nová verze generátorové funkce:

```
def quarters(next_quarter=0.0):
    while True:
        received = (yield next_quarter)
        if received is None:
            next_quarter += 0.25
        else:
            next_quarter = received
```

Výraz `yield` postupně vrací každou hodnotu volajícímu. Pokud volající zavolá metodu generátoru s názvem `send()`, obdrží odeslanou hodnotu generátorová funkce jako výsledek výrazu `yield`. Zde je způsob použití naší nové generátorové funkce:

```
result = []
generator = quarters()
while len(result) < 5:
    x = next(generator)
    if abs(x - 0.5) < sys.float_info.epsilon:
        x = generator.send(1.0)
    result.append(x)
```

Vytváříme proměnnou, která ukazuje na generátor, a voláme vestavěnou funkci `next()`, která získá následující prvek ze zadaného generátoru. (Stejného efektu lze dosáhnout zavoláním speciální metody `__next__()`, v tomto případě tedy `x = generator.__next__()`.) Pokud se hodnota rovná `0.5`, odešleme do generátoru hodnotu `1.0` (který okamžitě vynese tuto hodnotu zpět). Tentokrát seznam `result` obsahuje `[0.0, 0.25, 1.0, 1.25, 1.5]`.

V následujícím oddílu se podíváme na program `magic-numbers.py`, který zpracovává soubory zadané na příkazovém řádku. Shell systému Windows (`cmd.exe`) ale naneštěstí neposkytuje expanzi zástupných znaků (označovaná též jako *globbing souborů*), takže pokud program spustíme ve Windows s argumentem `*.*`, pak v seznamu `sys.argv` nebudeme mít všechny soubory v aktuální adresáři, ale jen textový literál „*.*“. Tento problém vyřešíme vytvořením dvou odlišných funkcí `get_files()`, z nichž jedna bude pro Windows a druhá pro Unix, a v obou použijeme generátory. Zde je kód:

```
if sys.platform.startswith("win"):
    def get_files(names):
        for name in names:
            if os.path.isfile(name):
                yield name
            else:
                for file in glob.iglob(name):
                    if not os.path.isfile(file):
```

```
                        continue
                    yield file
else:
    def get_files(names):
        return (file for file in names if os.path.isfile(file))
```

V obou případech se očekává, že se funkce zavolá se seznamem názvů souborů, takže jako argument můžeme použít například `sys.argv[1:]`.

V systému Windows prochází tato funkce všechny uvedené názvy. U každého názvu souboru vynese jeho název, pokud ale nejde o soubor (ale například o adresář), pak pomocí funkce `glob.iglob()` z modulu `glob` vrátí iterátor s názvy souborů, které daný název reprezentují po expanzi zástupných znaků. Pro běžný název jako `autoexec.bat` vrátí iterátor produkující jeden prvek (tento název) a pro název, který používá zástupné znaky jako `*.txt` vrátí iterátor produkující všechny odpovídající soubory (v tomto případě s příponou `.txt`). (K dispozici je také funkce `glob.glob()`, která nevrací iterátor, ale seznam.)

V systému Unix za nás provede expanzi zástupných znaků shell, a proto stačí, když vrátíme generátor pro všechny zadané soubory.*

Generátorové funkce lze použít též jako *korutiny*, mají-li správou strukturu. Korutiny jsou funkce, které lze uprostřed provádění pozastavit (na výrazu `yield`), počkat na výsledek příkazu `yield`, s níž se bude dále pracovat, a po jeho přijetí pokračovat ve zpracování. Jak uvidíme v pozdější části této lekce v oddílu věnovaném korutinám (strana 385), korutiny lze použít k rozdělení práce a k vytvoření zpracovávacích kolon.

Dynamické provádění kódu a dynamické importy

Existují určité situace, kdy je snazší napsat kód, který generuje potřebný kód, než napsat potřebný kód přímo. A v některých situacích je užitečné nechat uživatele, aby zadali kód (např. funkci v kalkulační tabulce), a nechat Python, aby pro nás zadaný kód provedl, než psát analyzátor a zpracovávat si jej sami – i když provádění libovolného kódu tímto způsobem samozřejmě představuje určité bezpečnostní riziko. Dalším použitím pro provádění dynamického kódu je poskytnutí zásuvných modulů, které rozšiřují funkčnost programu. Použití zásuvných modulů má jednu nevýhodu, která spočívá v tom, že veškerá nezbytná funkčnost není zabudována do programu (což může ztížit nasazení programu a dále je zde riziko, že se zásuvné moduly ztratí), ale také výhodu, protože zásuvné moduly lze modernizovat nezávisle na sobě a lze je nabízet samostatně, třeba jako vylepšení, která nebyla původně plánována.

Dynamické provádění kódu

Nejsnazší způsob, jak provést nějaký výraz, je použít vestavěnou funkci `eval()`, s níž jsme se poprvé setkali v lekci 6. Například:

```
x = eval("(2 ** 31) - 1")     # x == 2147483647
```

* Funkce `glob.glob()` nejsou tak výkonné jako například unixový shell Bash, protože i když podporují syntaxe `*`, `?` a `[]`, nepodporují syntaxi `{}`.

To je samozřejmě skvělé pro výrazy zadávané uživateli, ale co když potřebujeme dynamicky vytvořit funkci? K tomuto účelu můžeme použít vestavěnou funkci `exec()`. Uživatel například může zadat vzorec, například $4\pi r^2$, a název „plocha koule“, který chce převést na funkci. Za předpokladu, že symbol π nahradíme hodnotou `math.pi`, můžeme požadovanou funkci vytvořit takto:

```
import math
code = '''
def area_of_sphere(r):
    return 4 * math.pi * r ** 2
'''
context = {}
context["math"] = math
exec(code, context)
```

Musíme použít správné odsazení – koneckonců kód v uvozovkách je standardní kód jazyka Python. (I když v tomto případě bychom jej mohli napsat na jediný řádek, protože sadu tvoří jen jeden řádek.)

Pokud zavoláme funkci `exec()` s nějakým kódem jako jediným argumentem, pak nemůžeme žádným způsobem přistupovat k funkcím či proměnným vytvořeným v důsledku provedení tohoto kódu. Funkce `exec()` kromě toho nemůže přistupovat k žádnému importovanému modulu ani k žádné proměnné, funkci nebo k jiným objektům, které se v místě volání nacházejí v oboru platnosti. Oba tyto problémy lze vyřešit předáním slovníku jako druhého argumentu. Tento slovník poskytuje místo, kde lze po dokončení volání funkce `exec()` uchovávat odkazy na objekty, které tak budou přístupné. Například při použití slovníku `context` bude po zavolání funkce `exec()` obsahovat odkaz na objekt ukazující na funkci `area_of_sphere()`, kterou vytvořila funkce `exec()`. V tomto příkladu jsme potřebovali, aby funkce `exec()` byla schopná přistupovat k modulu `math`, a proto jsme vložili prvek do slovníku `context`, jehož klíč je názvem modulu a hodnota odkaz na objekt ukazující na odpovídající objekt modulu. Díky tomu bude uvnitř volání funkce `exec()` přístupná konstanta `math.pi`.

V některých případech je vhodné poskytnout funkci `exec()` celý globální kontext. To lze provést předáním slovníku vráceného funkcí `globals()`. Nevýhodou tohoto přístupu je, že jakýkoliv objekt vytvořený voláním funkce `exec()` se přidá do globálního slovníku. Řešením je zkopírovat globální kontext do jiného slovníku, například `context = globals().copy()`. Tímto způsobem bude mít funkce `exec()` i nadále přístup k importovaným modulům, k proměnným a dalším objektům v oboru platnosti, ale díky kopírování jakékoli změny tohoto kontextu provedené uvnitř volání funkce `exec()` zůstanou ve slovníku `context` a nebudou rozšířeny do globálního prostředí. (Mohlo by vypadat, že bezpečnější je `copy.deepcopy()`, ale pokud nám jde o bezpečnost, pak bychom neměli funkci `exec()` vůbec používat.) Předat můžeme i lokální kontext, například předáním `locals()` jako třetího argumentu, čímž zpřístupníme objekty v lokálním oboru platnosti kódu prováděného funkcí `exece()`.

Po zavolání funkce `exec()` bude slovník `context` obsahovat klíč s názvem `"area_of_sphere"`, jehož hodnotou bude funkce `area_of_sphere()`. Zde je způsob, jak k této funkci přistoupit a jak ji zavolat:

```
area_of_sphere = context["area_of_sphere"]
area = area_of_sphere(5)     # area == 314.15926535897933
```

Objekt `area_of_sphere` je odkazem na objekt ukazující na funkci, kterou jsme vytvořili dynamicky. Tento objekt lze použít stejně jako kteroukoli jinou funkci. Voláním funkce `exec()` jsme sice vytvořili jen jedinou funkci, avšak na rozdíl od funkce `eval()`, která umí pracovat jen na jednom výrazu, dokáže funkce `exec()` zpracovat libovolné množství příkazů jazyka Python, včetně celých modulů, jak uvidíme v následujícím pododdílu.

Dynamické importování modulů

Python nabízí tři přímočaré mechanizmy, které lze použít pro vytváření zásuvných modulů. Každý z nich zahrnuje importování modulů podle jména za běhu programu. Jakmile máme dynamicky importované dodatečné moduly, můžeme využít introspektivní funkce Pythonu pro ověření dostupnosti požadované funkčnosti a dle potřeby k ní přistupovat.

V tomto pododdílu prostudujeme program `magic-numbers.py`. Tento program přečte prvních 1000 bajtů z každého souboru zadaného na příkazovém řádku a pro každý z nich vypíše typ souboru (nebo text „Neznámý“) a název souboru. Zde je příklad použití programu na příkazovém řádku a část jeho výstupu:

```
C:\Python31\python.exe magic-numbers.py c:\windows\*.*
...
Neznámý.....................c:\windows\win.ini
XML.........................c:\windows\WindowsShell.Manifest
Neznámý.....................c:\windows\WindowsUpdate.log
Spustitelný soubor Windows....c:\windows\winhlp32.exe
Neznámý.....................c:\windows\WMSysPr9.prx
...
```

Program se pokusí načít libovolný modul, který se nachází ve stejném adresáři a jehož název obsahuje text „magic“. U těchto modulů se očekává, že budou poskytovat jedinou veřejnou funkci `get_file_type()`. V rámci příkladů ke knize jsou dostupné dva velice jednoduché ukázkové moduly `StandardMagicNumbers.py` a `WindowsMagicNumbers.py`, z nichž každý má funkci `get_file_type()`.

Na funkci `main()` tohoto programu se podíváme ve dvou krocích.

```
def main():
    modules = load_modules()
    get_file_type_functions = []
    for module in modules:
        get_file_type = get_function(module, "get_file_type")
        if get_file_type is not None:
            get_file_type_functions.append(get_file_type)
```

Za okamžik se podíváme na tři různé implementace funkce `load_modules()`, které vracejí (případně prázdný) seznam objektů představujících moduly, a poté se podíváme na funkci `get_funcion()`. U každého nalezeného modulu se pokusíme získat funkci `get_file_type()`, přičemž všechny takto nalezené funkce `get_file_type()` přidáváme do seznamu.

```
    for file in get_files(sys.argv[1:]):
```

```
        fh = None
        try:
            fh = open(file, "rb")
            magic = fh.read(1000)
            for get_file_type in get_file_type_functions:
                filetype = get_file_type(magic,
                                         os.path.splitext(file)[1])
                if filetype is not None:
                    print("{0:.<30}{1}".format(filetype, file))
                    break
            else:
                print("{0:.<30}{1}".format("Neznámý", file))
        except EnvironmentError as err:
            print(err)
        finally:
            if fh is not None:
                fh.close()
```

V tomto cyklu procházíme každý soubor zadaný na příkazovém řádku a z každého přečteme prvních 1000 bajtů. Potom postupně u každé funkce `get_file_type()` vyzkoušíme, zda dokáže stanovit typ aktuálního souboru. Je-li typ souboru stanoven, vypíšeme podrobnosti a ukončíme vnitřní cyklus, přičemž zpracování pokračuje dalším souborem. Pokud žádná z funkcí nedokáže určit typ souboru – nebo pokud nebyly nalezeny žádné funkce `get_file_type()` – vypíšeme „Neznámý“.

Nyní se podíváme na tři odlišné (ale ekvivalentní) způsoby dynamického importu modulů. Začneme nejdelším a nejkomplikovanějším přístupem, protože explicitně ukazuje každý krok.

```
def load_modules():
    modules = []
    for name in os.listdir(os.path.dirname(__file__) or "."):
        if name.endswith(".py") and "magic" in name.lower():
            filename = name
            name = os.path.splitext(name)[0]
            if name.isidentifier() and name not in sys.modules:
                fh = None
                try:
                    fh = open(filename, "r", encoding="utf8")
                    code = fh.read()
                    module = type(sys)(name)
                    sys.modules[name] = module
                    exec(code, module.__dict__)
                    modules.append(module)
                except (EnvironmentError, SyntaxError) as err:
                    sys.modules.pop(name, None)
                    print(err)
                finally:
```

```
                    if fh is not None:
                        fh.close()
    return modules
```

Začínáme průchodem přes všechny soubory v adresáři programu. Pokud se jedná o aktuální adresář, pak volání `os.path.dirname(__file__)` vrátí prázdný řetězec, což by způsobilo, že by funkce `os.listdir()` vyvolala výjimku, a proto musíme v takovém případě použít `"."`. Pro každý kandidátní soubor (končí příponou `.py` a jeho název obsahuje text „magic“) získáme název modulu odseknutím přípony souboru. Je-li tento název platným identifikátorem, pak se jedná o použitelný název modulu, a pokud dosud není v globálním seznamu modulů udržovaném ve slovníku `sys.modules`, tak se jej pokusíme importovat.

Text souboru přečteme do řetězce `code`. Další řádek (`module = type(sys)(name)`) je docela komplikovaný. Při zavolání funkce `type()` obdržíme objekt představující typ zadaného objektu. Když tedy například zavoláme `type(1)`, tak obdržíme `int`. Pokud objekt představující typ vypíšeme, obdržíme něco, co bude čitelné pro člověka, například „int“. Pokud ale objekt představující typ zavoláme jako funkci, pak získáme zpět objekt tohoto typu. Můžeme tak například získat číslo `5` v proměnné `x` zapsáním `x = 5` nebo `x = int(5)` nebo `x = type(0)(5)` nebo `int_type = type(0)` a `x = int_type(5)`. V tomto případě jsme použili `type(sys)`, kde `sys` je modul, takže obdržíme objekt představující typ, který reprezentuje modul (což je v podstatě totéž jako objekt představující třídu), který můžeme použít pro vytvoření nového modulu se zadaným názvem. Stejně jako u příkladu s typem `int`, kde nezáleželo na tom, jaké číslo jsme použili pro získání objektu představujícího typ `int`, tak ani zde nezáleží na tom, jaký modul použijeme (dokud se jedná o modul, který existuje, což je modul, který byl importován) pro získání objektu představujícího typ, který reprezentuje modul.

Jakmile máme nový (prázdný) modul, přidáme jej do globálního seznamu modulů, abychom jej omylem neimportovali podruhé. To provádíme před voláním funkce `exec()`, abychom těsněji napodobili chování příkazu `import`. Potom se voláním funkce `exec()` provede kód, který jsme načetli, přičemž jako kontext kódu použijeme slovník modulu. Na konci přidáme modul do seznamu modulů, který pak vrátíme volajícímu. A pokud dojde k nějakému problému, tak vymažeme modul z globálního slovníku modulů, pokud do něj byl přidán. Došlo-li k nějaké chybě, tak do seznamu modulů přidán nebude. Všimněte si, že funkce `exec()` umí zpracovat libovolné množství kódu (kdežto funkce `eval()` vyhodnocuje jediný výraz – viz tabulka 8.1) a v případě syntaktické chyby vyvolá výjimku `SyntaxError`.

Tabulka 8.1: Funkce pro dynamické programování a introspekci

Syntaxe	Popis
__import__(...)	Importuje modul podle jména (viz níže uvedený text).
compile(*zdroj*, *soubor*, *režim*)	Vrátí objekt představující kód, který je výsledkem zkompilování zdrojového textu. Soubor by měl obsahovat název souboru nebo "*řetězec*". Režim musí být „single“, „eval“ nebo „exec“.
delattr(*obj*, název)	Vymaže atribut se zadaným názvem z objektu *obj*.
dir(*obj*)	Vrátí seznam názvů v lokálním oboru platnosti nebo názvů objektu, je-li zadán objekt (např. jeho atributů a metod).

Syntaxe	Popis
eval(*zdroj*, *glob*, *lok*)	Vrátí výsledek vyhodnocení jediného výrazu ve zdroji. Jsou-li zadány slovníky *glob* a *lok*, pak *glob* je globální kontext a *lok* je lokální kontext.
exec(*obj*, *glob*, *lok*)	Vyhodnotí objekt *obj*, což může být řetězec nebo objekt představující kód z funkce `compile()`, a vrátí hodnotu `None`. Jsou-li zadány slovníky *glob* a *lok*, pak *glob* je globální kontext a *lok* je lokální kontext.
getattr(*obj*, *název*, *hodnota*)	Vrátí hodnotu atributu se zadaným názvem z objektu *obj* nebo hodnotu, je-li zadána a příslušný atribut neexistuje.
globals()	Vrátí slovník aktuálního globálního kontextu.
hasattr(*obj*, *název*)	Vrátí hodnotu `True`, má-li objekt *obj* atribut se zadaným názvem.
locals()	Vrátí slovník aktuálního lokálního kontextu.
setattr(*obj*, *název*, *hodnota*)	Nastaví atribut se zadaným názvem, který v případě potřeby vytvoří, na zadanou hodnotu pro objekt *obj*.
type(*obj*)	Vrátí objekt představující typ, který reprezentuje objekt *obj*.
vars(*obj*)	Vrátí kontext objektu *obj* jako slovník nebo lokální kontext, není-li objekt *obj* zadán.

Zde je druhý způsob dynamického načtení modulu za běhu programu. Níže uvedený kód nahrazuje blok `try ... except` v první verzi:

```
try:
    exec("import " + name)
    modules.append(sys.modules[name])
except SyntaxError as err:
    print(err)
```

Teoretickým problémem tohoto přístupu je to, že je potenciálně nedostatečně zabezpečený. Proměnná `name` může začínat `sys` a pokračovat nějakým destruktivním kódem.

A zde je třetí přístup, u něhož si opět ukážeme pouze část nahrazující blok `try ... except` v první verzi:

```
try:
    module = __import__(name)
    modules.append(module)
except (ImportError, SyntaxError) as err:
    print(err)
```

Jedná se o nejsnazší způsob dynamického importu modulů a ve srovnání s použitím funkce `exec()` je malinko bezpečnější, i když stejně jako jakýkoliv jiný dynamický import není v žádném případě naprosto bezpečný, protože nevíme, co se při importu modulu provádí.

Žádná z technik, které jsme si zde ukázali, si neporadí s balíčky či moduly umístěnými na různých cestách, není ale nijak obtížné uvedený kód odpovídajícím způsobem rozšířit. Potřebujete-li sofistikovanější přístup, pak si rozhodně přečtěte webovou dokumentace, zvláště pro funkci `__import__()`.

Po importu modulu chceme přistupovat k funkčnosti, kterou nabízí. Toho můžeme docílit pomocí vestavěných introspektivních funkcí Pythonu `getattr()` a `hasattr()`. Zde je jejich použití při implementaci funkce `get_function()`.

```
def get_function(module, function_name):
    function = get_function.cache.get((module, function_name), None)
    if function is None:
        try:
            function = getattr(module, function_name)
            if not hasattr(function, "__call__"):
                raise AttributeError()
            get_function.cache[module, function_name] = function
        except AttributeError:
            function = None
    return function
get_function.cache = {}
```

`collections.Callable` ➤ 379

Když dáme na okamžik stranou kód související s mezipaměti, pak funkce nedělá nic jiného, než že volá funkci `getattr()` na objektu modulu s názvem požadované funkce. Pokud žádný takovýto atribut neexistuje, vyvolá se výjimka `AttributeError`. Pokud ale atribut existuje, pak pomocí funkce `hasattr()` zjistíme, zda obsahuje atribut `__call__`, který mají všechny volatelné objekty (funkce a metody). (Později uvidíme hezčí způsob kontroly, zda je atribut volatelný.) Pokud atribut existuje a je volatelný, můžeme jej vrátit volajícímu, jinak vrátíme hodnotu `None`, která signalizuje, že funkce není dostupná.

Pokud by se zpracovávaly stovky souborů (např. kvůli použití `*.*` v adresáři `C:\windows`), pak by nebylo dobré procházet pro každý soubor vyhledávacím procesem pro každý modul. Proto ihned po definování funkce `get_function()` do ní přidáme slovník `cache` jako atribut. (Python nám obecně umožňuje přidávat libovolné atributy do libovolných objektů.) Při prvním zavolání funkce `get_function()` je slovník `cache` prázdný, takže volání `dict.get()` vrátí hodnotu `None`. Avšak pokaždé, když je nalezena vhodná funkce, tak ji umístíme do slovníku s n-ticí (modul, název funkce) jako klíčem a samotnou funkcí jako hodnotou. To znamená, že při druhém a každém dalším požadavku na určitou funkci je tato funkce okamžitě vrácena z mezipaměti, takže k žádnému hledání atributu nedojde.*

Tato technika použitá pro ukládání návratové hodnoty funkce `get_function()` do mezipaměti na základě zadané skupiny argumentů se nazývá *memoizace* (memoizing). Můžeme ji použít pro libovolnou funkci, která nemá žádné vedlejší efekty (nemění žádnou globální proměnnou) a která vždy pro tytéž (neměnitelné) argumenty vrací tentýž výsledek. Kód nezbytný pro vytvoření a správu mezipaměti pro každou memoizovanou funkci je stejný, a proto jde o ideálního kandidáta na dekorátor funkce (několik receptů na dekorátor `@memoize` najdete v kuchařce Pythonu na adrese *http://code.activestate.com/recipes/langs/python/*). Nicméně objekty představující modul jsou měnitelné, takže

* Malinko sofistikovanější funkce `get_function()`, která lépe pracuje s moduly bez požadované funkčnosti, se nachází v programu `magic-numbers.py` vedle výše uvedené verze.

některé standardní memoizační dekorátory by s naší funkcí `get_function()` v její nynější podobě nefungovaly. Snadným řešením by bylo použít místo samotného modulu řetězec `__name__` každého modulu pro první část klíče n-tice.

Upozornění: Provádět dynamické importování modulů je snadné a totéž platí i o provádění libovolného kódu jazyka Python pomocí funkce `exec()`. Může to být velice užitečné, protože tak například můžeme uložit kód v databázi. Nemáme ovšem žádnou kontrolu nad tím, co importovaný kód nebo kód provedený funkcí `exec()` udělá. Vzpomeňte si, že kromě proměnných, funkcí a tříd mohou moduly obsahovat kód, který se spustí při jejich importu. Pokud takový kód pochází z nedůvěryhodného zdroje, mohl by provést něco nepříjemného. Způsob řešení tohoto problému záleží na konkrétních okolnostech, i když v některých prostředích nebo v rámci osobních projektů by vůbec nemuselo jít o problém.

Lokální a rekurzivní funkce

Často je užitečné mít jednu či více malých pomocných funkcí uvnitř jiné funkce. V jazyku Python je toto možné bez zbytečných formalit. Stačí jen uvnitř definice stávající funkce definovat funkce, které potřebujeme. Tyto funkce se často označují jako *vnořené funkce* nebo *lokální funkce*. S příklady těchto funkcí jsme se již setkali v lekci 7.

Lokální funkce se často používají v situaci, kdy chceme použít rekurzi. V takovýchto případech obklopující funkce po svém zavolání vše připraví a pak provede první volání lokální rekurzivní funkce. Rekurzivní funkce (či metody) jsou takové funkce, které volají samy sebe. Z hlediska struktury lze ve všech přímo rekurzivních funkcích spatřovat dva případy: *základní případ* (base case) a *rekurzivní případ* (recursive case). Základní případ se používá k zastavení rekurze.

Rekurzivní funkce mohou být výpočetně náročné, protože se pro každé rekurzivní volání musí použít další zásobníkový blok. Nicméně některé algoritmy lze nejpřirozeněji vyjádřit právě pomocí rekurze. Většina implementací jazyka Python má pevně stanovený limit pro maximální počet rekurzivních volání. Tento limit obdržíme z funkce `sys.getrecursionlimit()` a můžeme jej změnit funkcí `sys.setrecursionlimit()`, ačkoliv jeho zvětšení je často známkou nevhodně zvoleného algoritmu nebo chybné implementace.

Klasický příklad rekurzivní funkce je funkce počítající faktoriál.* Například `factorial(5)` počítá 5! a vrátí 120, což je $1 \times 2 \times 3 \times 4 \times 5$:

```
def factorial(x):
    if x <= 1:
        return 1
    return x * factorial(x - 1)
```

To sice moc efektivní řešení není, ukazuje však dvě základní možnosti rekurzivních funkcí. Je-li zadané číslo `x` 1 nebo menší, vrátí se 1 a k žádné rekurzi nedojde – jedná se o základní případ. Pokud je ale `x` větší než 1, vrátí se `x * factorial(x - 1)`, což je rekurzivní případ, protože zde volá funkce `factorial()` sebe samu. Je zaručeno, že funkce skončí, protože pokud je počáteční hodnota `x` menší nebo rovna 1, použije se základní případ a funkce skončí ihned, a pokud je `x` větší než 1, bude se každé další rekurzivní volání provádět na čísle o jedničku menším, takže se nakonec provede i na čísle 1.

* Modul `math` Pythonu nabízí mnohem efektivnější funkci `math.factorial()`.

Lokální funkce a rekurzivní funkce ve smysluplném kontextu si ukážeme na funkci `indented_list_sort()` z modulu `IndentedList.py`. Tato funkce přijímá seznam řetězců, které používají odsazení pro vytvoření hierarchie, a řetězec obsahující jednu úroveň odsazení a vrací seznam se stejnými řetězci, které jsou však seřazené podle abecedy bez ohledu na velikost písmen s tím, že odsazené prvky jsou rekurzivně seřazeny pod svým rodičovským prvkem, což probíhá rekurzivně. Zde je příklad seznamů před (seznam `before`) a po (seznam `after`) tomto seřazení:

```
before = ["Nekovy",
"    Vodík",
"    Uhlík",
"    Dusík",
"    Kyslík",
"Přechodné prvky",
"    Lanthanoidy",
"        Cerium",
"        Europium",
"    Aktinoidy",
"        Uran",
"        Curium",
"        Plutonium",
"Alkalické kovy",
"    Lithium",
"    Sodík",
"    Draslík"]
```

```
after = ["Alkalické kovy",
"    Draslík",
"    Lithium",
"    Sodík",
"Nekovy",
"    Dusík",
"    Kyslík",
"    Uhlík",
"    Vodík",
"Přechodné prvky",
"    Aktinoidy",
"        Curium",
"        Plutonium",
"        Uran",
"    Lanthanoidy",
"        Cerium",
"        Europium"]
```

Máme-li seznam `before`, pak seznam `after` obdržíme voláním `after = IndentedList.indented_list_sort(before)`. Výchozí hodnotou odsazení jsou čtyři mezery, což je stejné odsazení, jaké se používá v seznamu `before`, a proto jej nemusíme explicitně uvádět.

Začneme pohledem na funkci `indented_list_sort()` jako celek a poté se podíváme na její dvě lokální funkce.

```
def indented_list_sort(indented_list, indent=" "):
    KEY, ITEM, CHILDREN = range(3)

    def add_entry(level, key, item, children):
    ...

    def update_indented_list(entry):
    ...

    entries = []
    for item in indented_list:
        level = 0
        i = 0
        while item.startswith(indent, i):
            i += len(indent)
```

```
            level += 1
        key = item.strip().lower()
        add_entry(level, key, item, entries)

    indented_list = []
    for entry in sorted(entries):
        update_indented_list(entry)
    return indented_list
```

Kód začíná tvorbou tří konstant, které slouží jako jména indexových pozic používaných lokálními funkcemi. Potom definujeme dvě lokální funkce, kterým se budeme věnovat za okamžik. Řadicí algoritmus pracuje ve dvou fázích. V první fázi vytváříme seznam záznamů, z nichž každý je n-tice se třemi prvky, které tvoří „klíč“, jenž se použije pro řazení, původní řetězec a seznam podřízených záznamů tohoto řetězce. Klíčem je pouhá kopie řetězce převedená na malá písmena bez bílého místa na obou koncích. Proměnná `level` uchovává úroveň odsazení, kde 0 znamená prvky na nejvyšší úrovni, 1 potomci prvků na nejvyšší úrovni a tak dále. Ve druhé fázi vytváříme nový seznam s odsazením a přidáváme do něj každý řetězec ze seznamu seřazených záznamů, dále všechny jejich podřízené řetězec – a tak dále, až k seřazenému seznamu s odsazením.

```
    def add_entry(level, key, item, children):
        if level == 0:
            children.append((key, item, []))
        else:
            add_entry(level - 1, key, item, children[-1][CHILDREN])
```

Tato funkce se volá pro každý řetězec v seznamu. Argument `children` je seznam, do kterého mají být přidány nové záznamy. Při zavolání z vnější funkce (`indented_list_sort()`) jde o seznam `entries`. Tímto způsobem se ze seznamu řetězců stane seznam záznamů, z nichž každý má řetězec na nejvyšší úrovni (neodsazený) a (třeba prázdný) seznam podřízených záznamů.

Jedná-li se o úroveň 0 (nejvyšší úroveň), přidáme do seznamu `entries` novou n-tici se třemi prvky, která obsahuje klíč (pro řazení), původní prvek (který půjde do výsledného seřazeného seznamu) a prázdný seznam potomků. Jedná se o základní případ, protože nedochází k žádné rekurzi. Je-li úroveň větší než 0, pak je parametr `item` potomkem (nebo následníkem) posledního prvku v seznamu potomků. V tomto případě rekurzivně voláme opět funkci `add_entry()`, přičemž úroveň snížíme o 1 a jako seznam, do kterého se mají přidat prvky, předáme seznam potomků posledního prvku v seznamu potomků. Je-li úroveň 2 nebo více, dochází k dalším rekurzivním voláním, až nakonec úroveň klesne na 0 a seznam potomků je připraven na přidání záznamu.

Například při dosažení řetězce „Přechodné prvky“ zavolá vnější funkce funkci `add_entry()` a předá jí úroveň 0, klíč „přechodné prvky“, prvek „Přechodné prvky“ a seznam `entries` jako seznam potomků. Vzhledem k tomu, že úroveň je 0, připojí se nový prvek do seznamu potomků (`entries`) se zadaným klíčem, prvkem a prázdným seznamem potomků. Další m řetězcem je „Lanthanoidy“, který je odsazený, a proto je potomkem řetězce „Přechodné prvky“. Toto volání funkce `add_entry()` má úroveň 1, klíč „lanthanoidy“, prvek „Lanthanoidy“ a seznam `entries` jako seznam potomků. Úroveň je 1, a proto funkce `add_entriy()` rekurzivně zavolá sebe samu, tentokrát s úrovní 0 (1 - 1), stejným

klíčem a prvkem, ale se seznamem potomků, který je seznamem potomků posledního prvku, což je seznam potomků prvku „Přechodné prvky“.

Zde je podoba seznamu `entries` po přidání všech řetězců, avšak před seřazením:

```
[('nekovy',
  'Nekovy',
  [('vodík', '    Vodík', []),
   ('uhlík', '    Uhlík', []),
   ('dusík', '    Dusík', []),
   ('kyslík', '    Kyslík', [])]),
   ('přechodné prvky',
    'Přechodné prvky',
    [('lanthanoidy',
      '    Lanthanoidy',
      [('cerium', '        Cerium', []),
       ('europium', '        Europium', [])]),
     ('aktinoidy',
      '    Aktinoidy',
      [('uran', '        Uran', []),
       ('curium', '        Curium', []),
       ('plutonium', '        Plutonium', [])])]),
   ('alkalické kovy',
    'Alkalické kovy',
    [('lithium', '    Lithium', []),
     ('sodík', '    Sodík', []),
     ('draslík', '    Draslík', [])])]
```

Tento výstup byl vytvořen pomocí funkce `pprint.pprint()` modulu `pprint` („pretty print“ – pěkný tisk). Všimněte si, že seznam `entries` má pouze tři prvky (všechny jsou n-tice se třemi prvky) a že posledním prvkem každé n-tice se třemi prvky je seznam n-tic se třemi prvky potomků (nebo prázdný seznam).

Funkce `add_entry()` je lokální i rekurzivní. Jako každá rekurzivní funkce má základní případ (parametr `level` má hodnotu 0), který ukončí rekurzi, a rekurzivní případ.

Tuto funkci bychom mohli napsat trošku odlišným způsobem:

```
    def add_entry(key, item, children):
        nonlocal level
        if level == 0:
            children.append((key, item, []))
        else:
            level -= 1
            add_entry(key, item, children[-1][CHILDREN])
```

Zde místo předávání úrovně jako parametru používáme pro přístup k proměnné ve vnějším, obklopujícím oboru platnosti příkaz `nonlocal`. Pokud bychom uvnitř funkce hodnotu proměnné `level` neměnili, nepotřebovali bychom příkaz `nonlocal`. V takové situaci by ji totiž Python v lokálním oboru platnosti (tj. ve vnitřní funkci) nenašel, podíval by se do obklopujícího oboru platnosti a tam by ji nalezl. V této verzi funkce `add_entry()` potřebujeme změnit hodnotu úrovně, takže podobně jako když potřebujeme pomocí příkazu `global` oznámit Pythonu, že chceme změnit globální proměnnou (aby se místo aktualizace globální proměnné nevytvořila nová lokální proměnná), potřebujeme také v případě proměnných, které chceme změnit, ale které patří do vnějšího oboru platnosti, použít příkaz `nonlocal`. Příkaz `global` je nejlepší pokud možno vůbec nepoužívat, což platí i o příkazu `nonlocal`, s nímž byste měli pracovat se zvláštní opatrností.

```
def update_indented_list(entry):
    indented_list.append(entry[ITEM])
    for subentry in sorted(entry[CHILDREN]):
        update_indented_list(subentry)
```

V první fázi algoritmu sestavujeme seznam záznamů, každý ve formě n-tice se třemi prvky (klíč, prvek, potomci), ve stejném pořadí, ve kterém se vyskytují v původním seznamu. Ve druhé fázi algoritmu začneme s novým prázdným seznamem s odsazením, projdeme všechny seřazené záznamy a pro každý zavoláme funkci `update_indented_list()`, čímž sestavíme nový seznam s odsazením. Funkce `update_indented_list()` je rekurzivní. Pro každý záznam na nejvyšší úrovni přidá prvek do seznamu `indented_list` a poté pro každý záznam potomka tohoto prvku zavolá sebe samu. Do seznamu `indented_list` přidá každého potomka, pro jehož potomky pak zavolá samu sebe – a tak pořád dál. Základní případ (když rekurze zastaví) nastane tehdy, když prvek nebo potomek nebo potomek potomka nebo některý z dalších potomků již nemá žádné vlastní potomky.

Python hledá seznam `indented_list` v lokální oboru platnosti (tj. v oboru platnosti vnitřní funkce), kde jej nenalezne, a proto se podívá do obklopujícího oboru platnosti, kde jej nalezne. Všimněte si ale, že uvnitř funkce jsme do seznamu `indented_list` přidali prvky, aniž bychom museli použít příkaz `nonlocal`. To je dáno tím, že příkaz `nonlocal` (a `global`) se týká odkazů na objekty a ne objektů, na které ukazují. Ve druhé verzi funkce `add_entry()` jsme museli použít příkaz `nonlocal` pro proměnnou `level`, protože operátor += aplikovaný na číslo opětovně svazuje odkaz na objekt na nový objekt, takže ve skutečnosti se provádí příkaz `level = level + 1`, takže proměnná `level` je nastavena tak, aby odkazovala na nový objekt typu `int`. Na druhou stranu při zavolání metody `list.append()` na objektu `indented_list` tak nedojde k opětovnému svázání, ale k modifikaci samotného seznamu, a proto není příkaz `nonlocal` nutný. (Ze stejného důvodu můžeme do slovníku, seznamu nebo jiné globální kolekce přidávat nebo z nich odebírat prvky, aniž bychom museli použít příkaz `global`.)

Dekorátory funkcí a metod

Dekorátor je funkce, která jako svůj jediný argument přijímá jinou funkci či metodu a vrací novou funkci či metodu, která do dekorované funkce či metody včleňuje dodatečně přidanou funkčnost. Některé z předdefinovaných dekorátorů jsme již používali, například `@property` a `@classmethod`. V tomto oddílu se naučíme vytvářet své vlastní dekorátory funkcí a v části této lekce si ukážeme, jak se vytvářejí dekorátory tříd.

Dekorátory tříd ➤ 365

U našeho prvního příkladu dekorátoru budeme předpokládat, že máme spoustu funkcí, které provádějí výpočty, a že některé z nich musejí vždy dávat kladný výsledek. Do každé z nich bychom mohli přidat příkaz `assert`, avšak použití dekorátoru je snazší a čistší. Zde je funkce dekorovaná dekorátorem `@positive_result`, který za okamžik vytvoříme:

```
@positive_result
def discriminant(a, b, c):
    return (b ** 2) - (4 * a * c)
```

Díky dekorátoru se vyvolá výjimka `AssertionError`, je-li výsledek menší než 0, a program se ukončí. Tento dekorátor můžeme samozřejmě použít na libovolné množství funkcí. Zde je implementace dekorátoru:

```
def positive_result(function):
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() má výsledek < 0"
        return result
    wrapper.__name__ = function.__name__
    wrapper.__doc__ = function.__doc__
    return wrapper
```

Dekorátory definují novou lokální funkci, která volá původní funkci. Zde je lokální funkcí funkce `wrapper()`. Ta volá původní funkci a ukládá výsledek, který pak pomocí příkazu `assert` otestuje, zda je kladný (v opačném případě program skončí). Po vytvoření funkce `wrapper()` nastavíme její název a dokumentační řetězec na hodnoty původní funkce. To pomůže introspekci, protože chceme, aby se chybové zprávy odkazovaly na název původní funkce. Nakonec vrátíme nově vytvořenou funkci, která se pak bude používat namísto funkce původní.

```
def positive_result(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() result isn't >= 0"
        return result
    return wrapper
```

Zde je malinko čistší verze dekorátoru `@positive_result`. Samotná funkce `wrapper()` je obalena pomocí dekorátoru `@functools.wraps` z modulu `functools`, který zajistí, aby funkce `wrapper()` měla stejný název a dokumentační řetězec jako původní funkce.

V některých případech by bylo užitečné mít možnost dekorátor parametrizovat, což ale na první pohled vypadá nemožně, protože dekorátor přijímá pouze jediný argument, funkci či metodu. Jedno elegantní řešení by tu ale bylo. Můžeme zavolat nějakou funkci s požadovanými parametry, která na jejich základě vrátí dekorátor, který pak dekoruje funkci, která následuje za ní. Například:

```
@bounded(0, 100)
def percent(amount, total):
    return (amount / total) * 100
```

Funkci `bounded()` zde voláme se dvěma argumenty. Tato funkce vrátí dekorátor, který se použije k dekorování funkce `percent()`. Účelem dekorátoru je v tomto případě zajistit, aby vrácené číslo bylo vždy mezi 0 a 100. Zde je implementace funkce `bounded()`:

```
def bounded(minimum, maximum):
    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            result = function(*args, **kwargs)
            if result < minimum:
                return minimum
            elif result > maximum:
                return maximum
            return result
        return wrapper
    return decorator
```

Tato funkce vytváří dekorátorovou funkci, která sama vytváří obalovou funkci (`wrapper()`). Obalová funkce provede výpočet a vrátí výsledek, který je v rámci stanoveného rozsahu. Funkce `decorator()` vrátí funkci `wrapper()` a funkce `bounded()` vrátí dekorátor.

Ještě je třeba poznamenat, že při každém vytvoření obalové funkce uvnitř funkce `bounded()` bude funkce `wrapper()` používat hodnoty minimum a maximum předané funkci `bounded()`.

Poslední dekorátor, který v tomto oddílu vytvoříme, je trošku složitější. Jedná se o funkci provádějící protokolování, které spočívá v záznamu názvu, argumentů a výsledku libovolné funkce, pro kterou je použita jako dekorátor. Například:

```
@logged
def discounted_price(price, percentage, make_integer=False):
    result = price * ((100 - percentage) / 100)
    if not (0 < result <= price):
        raise ValueError("neplatná cena")
    return result if not make_integer else int(round(result))
```

Pokud Python běží v ladicím režimu (což je běžný režim), pak při každém zavolání funkce `discounted_price()` se do souboru protokolu v místním dočasném adresáři počítače přidá záznam. Tyto záznamy mohou vypadat například takto:

```
volání: discounted_price(100, 10) -> 90.0
volání: discounted_price(210, 5) -> 199.5
volání: discounted_price(210, 5, make_integer=True) -> 200
volání: discounted_price(210, 14, True) -> 181
volání: discounted_price(210, -8) <type 'ValueError'>: neplatná cena
```

Pokud Python běží v optimalizovaném režimu (při použití volby příkazového řádku `-O` nebo při nastavení proměnné prostředí `PYTHONOPTIMIZE` na hodnotu `-O`), pak k žádnému protokolování nedochází. Zde je kód pro přípravu protokolování i pro samotný dekorátor:

```
if __debug__:
    logger = logging.getLogger("Logger")
    logger.setLevel(logging.DEBUG)
    handler = logging.FileHandler(
                os.path.join(tempfile.gettempdir(), "logged.log"))
    logger.addHandler(handler)

    def logged(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            log = "volání: " + function.__name__ + "("
            log += ", ".join(["{0!r}".format(a) for a in args] +
                             ["{0!s}={1!r}".format(k, v)
                              for k, v in kwargs.items()])
            result = exception = None
            try:
                result = function(*args, **kwargs)
                return result
            except Exception as err:
                exception = err
            finally:
                log += (") -> " + str(result)) if exception is None
                        else ") {0}: {1}".format(type(exception),
                                                 exception))
                logger.debug(log)
                if exception is not None:
                    raise exception
        return wrapper
else:
    def logged(function):
        return function
```

V ladicím režimu má globální proměnná `__debug__` hodnotu `True`. V tomto případě připravíme protokolování pomocí modulu `logging` a poté vytvoříme dekorátor `@logged`. Modul `logging` je velice výkonný a flexibilní. Dokáže protokolovat do souborů, rotovaných souborů, e-mailů, síťových připojení, serverů HTTP a do dalších prvků. My jsme využili pouze základní možnosti a vytvořili jsme objekt provádějící protokolování, nastavili jeho úroveň protokolování (podporováno je několik úrovní) a pro výstup jsme se rozhodli použít soubor.

Slovníkové komprehenze ➢ 134

Kód obalové funkce začíná přípravou zaznamenávaného řetězce s názvem funkce a argumenty. Poté se pokusíme zavolat funkci a uložit její výsledek. Pokud dojde k jakékoli výjimce, uložíme ji také. V každém případě se provede blok `finally`, v němž přidáme návratovou hodnotu (nebo výjimku)

do zaznamenávaného řetězce, který zapíšeme do protokolu. Pokud k žádné výjimce nedošlo, vrátíme výsledek. V opačném případě zachycenou výjimku opět vyvoláme, abychom správně napodobili chování původní funkce.

Pokud Python běží v optimalizovaném režimu, má proměnná `__debug__` hodnotu `False`. V tomto případě definujeme funkci `logged()` tak, aby jen vrátila zadanou funkci, takže kromě nepatrné režie při prvním vytvoření funkce nepředstavuje tento přístup za běhu programu žádnou zátěž navíc.

Je třeba poznamenat, že moduly `trace` a `cProfile` standardní knihovny dokážou spouštět a analyzovat programy a moduly a vytvářet nejrůznější trasovací a profilovací hlášení. Oba moduly využívají introspekci, takže na rozdíl od námi použitého dekorátoru `@logged` nevyžadují žádné zásahy do zdrojového kódu.

Anotace funkcí

Funkce a metody lze definovat s anotacemi, což jsou výrazy, které lze použít v signatuře funkce. Zde je obecná syntaxe:

```
def názevFunkce(par1 : výraz1, par2 : výraz2, ..., parN : výrazN)
  -> návratový_výraz:
    sada
```

Každá část s výrazem za dvojtečkou (`: výrazX`) je volitelnou anotací, stejně jako část s šipkou a návratovým výrazem. Poslední (nebo jediný) poziční parametr (je-li přítomný) může mít tvar `*args` s anotací nebo bez ní. Podobně poslední (nebo jediný) klíčový parametr (je-li přítomný) může mít tvar`**kwargs` a opět s anotací nebo bez ní.

Jsou-li anotace uvedeny, přidají se do slovníku funkce s názvem `__annotations__`. Pokud uvedeny nejsou, slovník zůstává prázdný. Klíči tohoto slovníku jsou názvy parametrů a hodnotami odpovídající výrazy. Syntaxe nám umožňuje anotovat všechny, některé nebo žádné parametry a anotovat či neanotovat návratovou hodnotu. Anotace nemají pro Python žádný zvláštní význam. Jediné, co Python v souvislosti s anotacemi provádí, je to, že je umisťuje do slovníku `__annotations__`. Jakákoli další akce je na nás. Zde je příklad anotace funkce, která se nachází v modulu `Util`.

```
def is_unicode_punctuation(s : str) -> bool:
    for c in s:
        if unicodedata.category(c)[0] != "P":
            return False
    return True
```

Každý znak ze znakové sady Unicode patří do určité kategorie a každá kategorie je identifikována pomocí dvouznakového identifikátoru. Všechny kategorie, které začínají „P“, obsahují interpunkční znaky.

Zde jsme použili datové typy Pythonu jako anotační výrazy. Pro Python však žádný konkrétní význam nemají, což je patrné z následujících volání:

```
Util.is_unicode_punctuation("zebr\a")    # vrátí: False
Util.is_unicode_punctuation(s="!@#?")    # vrátí: True
Util.is_unicode_punctuation(("!", "@"))  # vrátí: True
```

V prvním volání používáme poziční argument a ve druhém klíčovaný argument, abychom si ukázali, že oba druhy fungují dle očekávání. V posledním volání předáváme místo řetězce n-tici, která je přesto přijata, protože Python s anotacemi neudělá nic jiného, než že je zaznamená do slovníku `__annotations__`.

Pokud chceme dát anotacím nějaký význam, jako je například poskytnutí typové kontroly, pak můžeme funkce, na které chceme tento význam aplikovat, dekorovat vhodným dekorátorem. Zde je velmi základní dekorátor pro typovou kontrolu:

```
def strictly_typed(function):
    annotations = function.__annotations__
    arg_spec = inspect.getfullargspec(function)

    assert "return" in annotations, "chybějící typ pro návratovou hodnotu"
    for arg in arg_spec.args + arg_spec.kwonlyargs:
        assert arg in annotations, ("chybějící typ pro parametr '" +
                                    arg + "'")
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        for name, arg in (list(zip(arg_spec.args, args)) +
                          list(kwargs.items())):
            assert isinstance(arg, annotations[name]), (
                    "argument '{0}' měl být typu {1}, je však typu {2}"
                    .format(name, annotations[name], type(arg)))
        result = function(*args, **kwargs)
        assert isinstance(result, annotations["return"]),
                 ("návratová hodnota měla být typu {0}, je však typu {1}"
                  .format(annotations["return"], type(result)))
        return result
    return wrapper
```

Tento dekorátor vyžaduje, aby každý argument a každá návratová hodnota byly anotovány očekávaným typem. Při vytváření předávané funkce kontroluje, zda jsou argumenty funkce a návratový typ anotovány nějakým typem, a za běhu programu ověřuje, zda typy skutečných argumentů odpovídající těmto očekávaným typům.

Modul `inspect` nabízí výkonné introspektivní služby pro objekty. My jsme zde využili pouze malou část obdrženého objektu se specifikacemi argumentů pro získání názvů každého pozičního a klíčovaného argumentu (ve správném pořadí v případě pozičních argumentů). Tyto názvy jsme pak použili ve spojení s anotačním slovníkem pro kontrolu, že anotován je každý parametr i návratová hodnota.

Obalová funkce vytvářená uvnitř dekorátoru začíná průchodem přes všechny dvojice název-argument zadaných pozičních a klíčovaných argumentů. Funkce `zip()` vrací iterátor a metoda `dictionary.items()` slovníkový pohled, takže je nemůžeme spojit dohromady, a proto nejdříve oba výsledky převedeme na seznam. Pokud má některý ze skutečných argumentů jiný typ, než jaký je uveden v odpovídající anotaci, příkaz `assert` selže. V opačném případě zavoláme skutečnou funkci a zkontrolujeme její návratovou hodnotu. Má-li správný typ, vrátíme jej. Na konci funkce

`strictly_typed()` vrátíme jako obvykle tuto obalovou funkci. Všimněte si, že kontrolu provádíme pouze v ladicím režimu (což je výchozí režim Pythonu, ovládaný volbou příkazového řádku `-O` a proměnnou prostředí `PYTHONOPTIMIZE`).

Pokud dekorujeme funkci `is_unicode_punctuation()` dekorátorem `@strictly_typed` a vyzkoušíme stejné příklady jako výše s použitím dekorované verze, začnou tyto anotace pracovat:

```
is_unicode_punctuation("zebr\a")    # vrátí: False
is_unicode_punctuation(s="!@#?")    # vrátí: True
is_unicode_punctuation(("!", "@"))  # vyvolá AssertionError
```

Nyní jsou typy argumentů kontrolovány, takže se v posledním případě vyvolá výjimka `AssertionError`, protože n-tice není řetězec ani podtřída třídy `str`.

Nyní se podíváme na zcela jiné použití anotace. Zde je malá funkce, která má stejnou funkčnost jako vestavěná funkce `range()`, jenže vždy vrací čísla typu `float`:

```
def range_of_floats(*args) -> "author=Reginald Perrin":
    return (float(x) for x in range(*args))
```

Samotná funkce tuto anotaci nijak nepoužívá, snadno ale napíšeme nástroj, který importuje všechny moduly projektu a vytvoří seznam názvů funkcí a jmen jejich autorů, přičemž název funkce extrahujeme z jejího atributu `__name__` a jméno autora z hodnoty prvku `"return"` ve slovníku `__annotations__`.

Anotace jsou úplně novým prvkem Pythonu a vzhledem k tomu, že jim Python nepřisuzuje žádný předem definovaný význam, zůstává jejich použití omezeno pouze naší představivostí. Další nápady pro možná využití a několik užitečných odkazů najdete v dokumentu PEP 3107 s názvem „Function Annotations“ (viz *www.python.org/dev/peps/pep-3107*).

Další objektově orientované programování

V této části se podíváme podrobněji na podporu objektové orientace v jazyku Python a naučíme se řadu technik, které dokážou snížit množství potřebného kódu a které dále rozšíří sílu a možnosti programovacích prvků, jež jsou nám dostupné. Začneme ale jedním velice malým a jednoduchým novým prvkem. Zde je začátek definice třídy Point, která se chová naprosto stejně jako její verze z lekce 6:

```
class Point:

    __slots__ = ("x", "y")

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

Funkce pro přístup k atributům ➤ 338

Když vytvoříme třídu bez atributu `__slots__`, vytvoří Python v pozadí pro každou instanci soukromý slovník s názvem `__dict__`, který bude uchovávat datové atributy instance. Díky tomuto slovníku můžeme přidávat a odstraňovat atributy z objektů. (V dřívější části této lekce jsme například do funkce `get_function()` přidali atribut `cache`.)

Pokud potřebujeme pouze objekty, u nichž přistupujeme k původním atributům a u nichž nepotřebujeme atributy přidávat ani odebírat, pak můžeme vytvářet třídy, které slovník `__dict__` nemají. Toho docílíme jednoduše tak, že definujeme atribut třídy s názvem `__slots__`, jehož hodnotou je n-tice názvů atributů. Každý objekt této třídy bude mít atributy s uvedenými názvy a nebude mít slovník `__dict__`. Dále u nich nepůjdou přidávat ani odebírat atributy. Tyto objekty spotřebují méně paměti a ve srovnání s tradičními objekty jsou rychlejší, což ale nebude příliš znatelné, pokud nevytváříme obrovské množství objektů. Pokud odvodíme třídu od třídy, která používá n-tici `__slots__`, pak musíme deklarovat sloty v této podtřídě, a to i tehdy, je-li prázdná (`__slots__ = ()`), jinak by paměťové a rychlostní úspory byly ztraceny.

Řízení přístupu k atributům

Někdy je pohodlné mít třídu, v níž nejsou hodnoty atributů uloženy, ale počítají se až za běhu. Zde je kompletní implementace takové třídy:

```
class Ord:

    def __getattr__(self, char):
        return ord(char)
```

Se třídou `Ord` můžeme vytvořit instanci, například `ord = Ord()`, která nám poskytuje alternativu k vestavěné funkci `ord()`, jež funguje pro každý znak, který je platným identifikátorem. Například `ord.a` vrátí 97, `ord.Z` vrátí 90 a `ord.Í` vrátí 229. (Avšak `ord.!` a podobné výrazy jsou syntaktické chyby.)

Všimněte si, že pokud třídu `Ord` napíšeme do editoru IDLE, pak příkaz `ord = Ord()` nebude fungovat. To je dáno tím, že tato instance má stejný název jako vestavěná funkce `ord()`, kterou používá třída `Ord`, takže volání `ord()` se ve skutečnosti stane voláním instance `ord()`, což vede k výjimce `TypeError`. K tomuto problému by nedošlo, pokud bychom importovali modul obsahující třídu `Ord`, protože interaktivně vytvořený objekt ord a vestavěná funkce `ord()` používaná třídou `Ord` by se nacházely v samostatných modulech, takže by jeden nevytlačoval druhého. Pokud opravdu potřebujeme vytvořit třídu interaktivně a opětovně použít název nějakého vestavěného prvku, pak musíme zajistit, aby taková třída zavolala onen vestavěný prvek – v tomto případě musíme importovat modul `builtins`, který poskytuje jednoznačný přístup ke všem vestavěným funkcím, a místo holé funkce `ord()` pak zavolat funkce `builtins.ord()`.

Zde je další malá, přesto však kompletní třída. Díky této třídě můžeme vytvářet „konstanty“. Není sice obtížné měnit jejich hodnoty za zády této třídy, může nám ale pomoci předejít nechtěným chybám.

```
class Const:

    def __setattr__(self, name, value):
```

```
        if name in self.__dict__:
            raise ValueError("nemohu změnit konstantní atribut")
        self.__dict__[name] = value

    def __delattr__(self, name):
        if name in self.__dict__:
            raise ValueError("nemohu vymazat konstantní atribut")
        raise AttributeError("'{0}' object has no attribute '{1}'"
                             .format(self.__class__.__name__, name))
```

Díky této třídě můžeme vytvořit konstantní objekt, třeba `const = Const()`, a nastavit na něm libovolný atribut, například `const.limit = 591`. Jakmile ale nějaký atribut nastavíme, způsobí jakýkoliv pokus o jeho změnu či vymazání vyvolání výjimky `ValueError`, ačkoliv číst jej můžeme dle libosti. Metodu `__getattr__()` jsme reimplementovat nemuseli, protože metoda `object.__getattr__()` bázové třídy dělá přesně to, co chceme – vrací hodnotu zadaného atributu nebo vyvolá výjimku `AttributeError`, pokud zadaný atribut neexistuje. V metodě `__delattr__()` napodobujeme v případě neexistujícího atributu chybovou zprávu metody `__getattr__()`, k čemuž musíme získat název třídy, ve které se nachází, a také název neexistujícího atributu. Třída funguje, protože používáme slovník `__dict__`, který používají i metody `__getattr__()`, `__setattr__()` a `__delattr__()` bázové třídy, ačkoliv zde jsme použili pouze metodu `__getattr__()` bázové třídy. Všechny speciální metody použité pro přístup k atributům jsou uvedeny v tabulce 8.2.

Tabulka 8.2: Speciální metody pro přístup k atributům

Speciální metoda	Použití	Popis
__delattr__(self, *název*)	del x.n	Vymaže z objektu x atribut s názvem n.
__dir__(self)	dir(x)	Vrátí seznam názvů atributů objektu x.
__getattr__(self, *název*)	v = x.n	Vrátí hodnotu atributu objektu x s názvem n, není-li nalezena přímo.
__getattribute__(self, *název*)	v = x.n	Vrátí hodnotu atributu objektu x s názvem n (viz níže uvedený text).
__setattr__(self, *název*, *hodnota*)	x.n = v	Nastaví atribut objektu x s názvem n na hodnotu v.

Konstanty můžeme získat i jiným způsobem: pomocí pojmenovaných n-tic. Zde je několik příkladů:

```
Const = collections.namedtuple("_", "min max")(191, 591)
Const.min, Const.max                        # vrátí: (191, 591)
Offset = collections.namedtuple("_", "id name description")(*range(3))
Offset.id, Offset.name, Offset.description # vrátí: (0, 1, 2)
```

V obou případech jsme pro název pojmenované n-tice použili nedůležité jméno, protože nám stačí jen jedna instance každé pojmenované n-tice, a ne podtřída třídy `tuple` pro vytváření instancí pojmenované n-tice. Ačkoliv Python výčtový datový typ nepodporuje, můžeme podobného efektu dosáhnout právě pomocí pojmenovaných n-tic použitých výše uvedeným způsobem.

Image.py ➤ 256

Pro náš poslední pohled na speciální metody pro přístup k atributům se vrátíme k příkladu, s nímž jsme se poprvé setkali v lekci 6. V této lekci jsme vytvořili třídu `Image`, jejíž šířka, výška a barva pozadí jsou při vytvoření její instance pevně dány (i když při načtení obrázku se změní). Přístup k těmto údajům je tedy realizován pomocí vlastností určených pouze pro čtení. Měli jsme tak například následující vlastnost:

```
@property
def width(self):
    return self.__width
```

Tento kód je sice jednoduchý, jeho tvorba může být ale pracná, máme-li spoustu vlastností určených pouze pro čtení. Zde je jiné řešení, v němž se postaráme o všechny vlastnosti třídy Image určené pouze pro čtení v jediné metodě:

```
def __getattr__(self, name):
    if name == "colors":
        return set(self.__colors)
    classname = self.__class__.__name__
    if name in frozenset({"background", "width", "height"}):
        return self.__dict__["_{classname}__{name}".format(**locals())]
    raise AttributeError("'{classname}' object has no "
                         "attribute '{name}'".format(**locals()))
```

Pokud se pokusíme přistoupit k atributu objektu, který neexistuje, tak Python zavolá metodu `__getattr__()` (tedy za předpokladu, že je implementována a že není reimplementována metoda `__getattribute__()`), které předá jako argument název požadovaného atributu. Implementace metody `__getattr__()` musí vyvolat výjimku `AttributeError`, pokud zadaný atribut není schopná obstarat.

Máme-li například příkaz `image.colors`, začne Python hledat atribut `colors`, nenajde jej, a proto zavolá metodu `Image.__getattr__(image, "colors")`. V tomto případě metoda `__getattr__()` zpracuje název atributu `"colors"` a vrátí kopii množiny barev používaných v daném obrázku.

Další atributy jsou neměnitelné, a proto je můžeme bez obav vracet volajícímu přímo. Pro každý z nich bychom mohli tímto způsobem napsat samostatné příkazy `elif`:

```
elif name == "background":
    return self.__background
```

My jsme se ale rozhodli, že se vydáme o něco kompaktnější cestou. Vzhledem k tomu, že víme, že všechny nespeciální atributy jsou na pozadí uchovávány ve slovníku `self.__dict__`, tak k nim můžeme přistupovat přímo. Název soukromých atributů (tj. takové, jejichž název začíná dvěma podtržítky) je zkomolen do tvaru *`_názevTřídy__názevAtributu`*, s čímž musíme při získávání hodnoty atributu ze soukromého slovníku objektu počítat.

Kvůli komolení názvů (name mangling) u soukromých atributů a standardnímu chybovému textu výjimky `AttributeError` musíme znát název třídy, v níž se nacházíme. (Nemusí jít o třídu Image, protože daný objekt může být instancí nějaké její podtřídy.) Každý objekt má speciální atribut

`__class__`, takže uvnitř metod je vždy k dispozici atribut `self.__class__`, k němuž můžeme v metodě `__getattr__()` přistupovat bez riskování nechtěné rekurze.

Všimněte si drobného rozdílu v tom, že při použití metody `__getattr__()` nebo atributu `self.__class__` přistupujeme k atributu ve třídě dané instance (nebo v nějaké její podtřídě), ale při přímém přístupu k atributu se používá třída, v níž je tento atribut definován.

Upozornění: Existuje ještě jedna speciální metoda, kterou jsme zde neprobírali a která nese název `__getattribute__()`. Zatímco metoda `__getattr__()` se při hledání (nespeciálního) atributu volá jako poslední, metoda `__getattribute__()` se při přístupu ke každému atributu volá jako první. V některých případech sice můžeme být užitečná, nebo dokonce nezbytná, její reimplementace však může být složitá. Při její reimplementaci si totiž musíme dávat dobrý pozor, abychom ji nezavolali rekurzivně, protože se v těchto případech často používá volání `super().__getattribute__()` nebo `object.__getattribute__()`. Její reimplementace může dále vést k degradaci výkonu ve srovnání s přímým přístupem k atributům či s vlastnostmi, protože se volá pro každý přístup k atributu. V žádné ze tříd prezentovaných v této knize nebudeme metodu `__getattribute__()` reimplementovat.

Funktory

V Pythonu se termínem *funkční objekt* označuje odkaz na objekt ukazující na libovolný volatelný objekt, jako je funkce, lambda funkce nebo metoda. Definice zahrnuje také třídy, poněvadž odkaz na objekt ukazující na třídu je volatelný objekt, který při své zavolání vrátí instanci dané třídy (např. `x = int(5)`). V informatice se jako *funktor* označuje objekt, který lze volat, jako by šlo o funkci, takže v Pythonu termín funktor značí jen další druh funkčního objektu. Libovolná třída, která má speciální metodu `__call__()`, je funktor. Klíčovou výhodou nabízenou funktory je to, že dovedou udržovat stavové informace. Mohli bychom tak například vytvořit funktor, který vždy ořeže základní interpunkční znaménka z konce řetězce. Tento funktor bychom mohli použít třeba takto:

```
strip_punctuation = Strip(",;:.!?")
strip_punctuation("Nazdar lidi!")     # vrátí: 'Nazdar lidi'
```

Zde vytváříme instanci funktoru `Strip` a inicializujeme ji hodnotou `",;:.!?"`. Při každém zavolání pak tato instance vrátí zadaný řetězec bez interpunkčních znaků. Kompletní implementace třídy `Strip` vypadá takto:

```
class Strip:

    def __init__(self, characters):
        self.characters = characters

    def __call__(self, string):
        return string.strip(self.characters)
```

Stejného výsledku bychom dosáhli i pomocí obyčejné funkce nebo lambda funkce, pokud ale potřebujeme uložit nějaké stavové informace nebo provést komplexní zpracování, bývá funktor často tím pravým řešením.

Schopnost funktoru zachytit pomocí třídy stav je velice všestranná a silná, někdy však poskytuje více, než skutečně potřebujeme. Další možnost pro zachycení stavu spočívá v použití *uzávěru*. Uzávěr je funkce nebo metoda, která zachytí nějaký externí stav. Například:

```
def make_strip_function(characters):
    def strip_function(string):
        return string.strip(characters)
    return strip_function

strip_punctuation = make_strip_function(",;:.!?")
strip_punctuation("Nazdar lidi!")     # vrátí: 'Nazdar lidi'
```

Funkce `make_strip_function()` přijímá jako svůj jediný argument znaky, které mají být ořezány, a vrací funkci (`strip_function()`), která přijímá řetězcový argument, z něhož ořeže znaky, které byly zadány v okamžiku vytvoření tohoto uzávěru. To tedy znamená, že stejně jako můžeme vytvářet libovolné množství instancí třídy `Strip`, kde každá má své vlastní znaky k ořezání, můžeme vytvářet libovolné množství ořezávacích funkcí s jejich vlastními znaky.

Klasickou oblastí pro nasazení funktorů je poskytování klíčových funkcí řadicí rutiny. Zde je generická faktorová třída `SortKey` (ze souboru `SortKey.py`):

```
class SortKey:

    def __init__(self, *attribute_names):
        self.attribute_names = attribute_names

    def __call__(self, instance):
        values = []
        for attribute_name in self.attribute_names:
            values.append(getattr(instance, attribute_name))
        return values
```

Objekt typu `SortKey` po své vytvoření uchovává n-tici s názvy atributů, se kterými byl inicializován. Jakmile dojde k zavolání tohoto objektu, vytvoří pro zadanou instanci seznam hodnot atributů, a to v pořadí, ve kterém byly uvedeny při jeho inicializaci. Představte si kupříkladu, že máme následující třídu `Person`:

```
class Person:

    def __init__(self, forename, surname, email):
        self.forename = forename
        self.surname = surname
        self.email = email
```

Předpokládejme, že máme seznam objektů typu `Person` v seznamu `person`. Tento seznam můžeme seřadit podle příjmení takto: `people.sort(key=SortKey("surname"))`. Je-li v seznamu mnoho lidí, pak je pravděpodobné, že se objeví i stejná příjmení, a proto můžeme následujícím způsobem seznam seřadit podle příjmení a pak v rámci příjmení podle křestního jména: `people.sort(key=SortKey`

`("surname", "forename"))`. A pokud bychom v seznamu měli lidi se stejným jménem i příjmením, pak bychom mohli přidat také atribut `email`. Změnou pořadí názvů atributů zadávaných funktoru `SortKey` bychom samozřejmě mohli seznam seřadit podle jména a pak podle příjmení.

Další možnost, jak dosáhnout stejného výsledku, ovšem bez nutnosti vytvářet funktor, tkví v použití funkce `operator.attrgetter()` modulu `operator`. Například pro seřazení podle příjmení můžeme napsat: `people.sort(key=operator.attrgetter("surname"))`. A podobně pro seřazení podle příjmení a jména napíšeme: `people.sort(key=operator.attrgetter("surname", "forename"))`. Funkce `operator.attrgetter()` vrací funkci (uzávěr), která při zavolání na nějakém objektu vrátí ty z jeho atributů, které byly uvedeny při vytváření uzávěru.

Funktory se v Pythonu používají spíše méně často než v jiných jazycích s jejich podporou, protože Python má i jiné prostředky pro dosažení stejných cílů – například uzávěry nebo rutiny pro čtení prvků a atributů.

Správce kontextu

Správce kontextu nám umožňují zjednodušit kód, protože zajišťují, že před a po provedení jistého bloku kódu provedou určité operace. Tohoto chování je dosaženo prostřednictvím dvou speciálních metod `__enter__()` a `__exit__()`, definovaných správci kontextu, s nimiž Python pracuje speciálně v rámci příkazu `with`. Při vytvoření správce kontextu v příkazu `with` se automaticky zavolá jeho metoda __enter__(), a jakmile správce kontextu za svým příkazem `with` obor platnosti opouští, automaticky se zavolá jeho metoda `__exit__()`.

Můžeme vytvořit svého vlastního správce kontextu nebo použít některý z předdefinovaných, jako jsou například objekty souboru vrácené vestavěnou funkcí `open()`, o čemž se přesvědčíme v pozdější části tohoto oddílu. Syntaxe pro použití správce kontextu vypadá takto:

```
with výraz as proměnná:
    sada
```

Uvedený *`výraz`* musí být objektem správce kontextu nebo jej musí vytvářet. Je-li uvedená volitelná část `as` *`proměnná`*, pak tato *`proměnná`* odkazuje na objekt vrácený metodou `__enter__()` správce kontextu (což často bývá samotný správce kontextu). Správce kontextu zaručuje, že provede svůj „ukončovací“ kód (a to i v případě výjimek), a proto jej můžeme v řadě situací použít pro odstranění bloku `finally`.

Některé typy Pythonu jsou správce kontextu, například všechny objekty souboru vrácené funkcí open(), takže při práci se soubory můžeme eliminovat blok `finally`, jak ukazují tyto dva ekvivalentní úryvky kódu (předpokládáme, že `process()` je na jiném místě definovaná funkce):

```
fh = None
try:
    fh = open(filename)
    for line in fh:
        process(line)
except EnvironmentError as err:
    print(err)
finally:
    if fh is not None:
        fh.close()
```

```
try:
    with open(filename) as fh:
        for line in fh:
            process(line)
except EnvironmentError as err:
    print(err)
```

Objekt souboru je správce kontextu, jehož ukončovací kód vždy uzavře soubor, pokud byl otevřen. Ukončovací kód se provede bez ohledu na to, zda došlo či nedošlo k nějaké výjimce. Pokud ale k nějaké výjimce skutečně dojde, je tato výjimka propagována dále, což zajišťuje, že se soubor uzavře a my i tak dostaneme šanci zpracovat případné chyby, třeba jako v tomto případě vypsáním chybové zprávy pro uživatele.

Správce kontextu ve skutečnosti výjimky propagovat nemusejí, tím by ale v posledku jakoukoli výjimku skryly, což by téměř jistě znamenalo chybu při programování. Všechny vestavěné správce kontextu a správce kontextu standardní knihovny výjimky propagují. Někdy potřebujeme použít více než jeden správce kontextu najednou. Například:

```
try:
    with open(source) as fin:
        with open(target, "w") as fout:
            for line in fin:
                fout.write(process(line))
except EnvironmentError as err:
    print(err)
```

Zde čteme řádky ze zdrojového souboru a jejich zpracované verze zapisujeme do cílového souboru.

Použití vnořených příkazů `with` může rychle vést ke spoustě odsazení. Naštěstí je zde modul standardní knihovny `contextlib`, jež nabízí dodatečnou podporu pro správce kontextu včetně funkce `contextlib.nested()`, která umožňuje zpracování dvou či více správců kontextu v jednom příkazu `with`. Zde je druhá, identická verze výše uvedeného kódu, ovšem s výraznou úsporou řádků:

```
try:
    with contextlib.nested(open(source), open(target, "w")) as (fin, fout):
        for line in fin:
```

3.1

Funkce `contextlib.nested()` je nezbytná pouze pro Python 3.0. Počínaje Pythonem 3.1 je tato funkci již zastaralá, protože Python 3.1 dokáže pracovat s více správci kontextu v jediném příkazu `with`. Zde je stejný příklad, ovšem tentokrát pro Python 3.1:

```
try:
    with open(source) as fin, open(target, "w") as fout:
        for line in fin:
```

S použitím této syntaxe jsou správce kontextu a s nimi spojené proměnné pohromadě, díky čemuž je příkaz `with` čitelnější, než při jejich vnořování nebo použití funkce `contextlib.nested()`.

Práce s vlákny ➤ 424

Správce kontextu nejsou jen objekty souboru. Správcem kontextu je například také několik tříd pro práci s vlákny, které se používají k zamykání. Správce kontextu lze též použít s čísly typu `decimal.Decimal`, což je užitečné, pokud chceme provádět výpočty s určitými nastaveními (jako je např. přesnost).

Pokud chceme vytvořit vlastního správce kontextu, potom musíme vytvořit třídu, která nabízí dvě metody: `__enter__()` a `__exit__()`. Kdykoliv se pak na instanci takové třídy použije příkaz `with`, zavolá se metoda `__enter__()` a její návratová hodnota se použije pro proměnnou v klauzuli `as` (nebo se zahodí, pokud klauzule `as` chybí). Jakmile program opustí obor platnosti příkazu `with`, zavolá se metoda `__exit__()` (s údaji o výjimce předanými jako argumenty, pokud k nějaké výjimce došlo).

Představte si, že chceme provést několik operací na seznamu atomickým způsobem, což znamená, že chceme, aby se provedly buď všechny operace, nebo žádná z nich, takže výsledný seznam bude stále ve známém stavu. Máme-li například seznam celých čísel a chceme připojit další, jiné vymazat a několik změnit a to vše jako jedinou operaci, pak bychom mohli napsat následující kód:

```
try:
    with AtomicList(items) as atomic:
        atomic.append(58289)
        del atomic[3]
        atomic[8] = 81738
        atomic[index] = 38172
except (AttributeError, IndexError, ValueError) as err:
    print("nebyly provedeny žádné změny:", err)
```

Pokud nedojde k žádné výjimce, aplikují se všechny tyto operace na původní seznam (`items`). Pokud ale nastane nějaká výjimka, neprovedou se vůbec žádné změny. Zde je kód pro správce kontextu `AtomicList`:

```
class AtomicList:

    def __init__(self, alist, shallow_copy=True):
        self.original = alist
        self.shallow_copy = shallow_copy

def __enter__(self):
    self.modified = (self.original[:] if self.shallow_copy
                     else copy.deepcopy(self.original))
    return self.modified

def __exit__(self, exc_type, exc_val, exc_tb):
    if exc_type is None:
        self.original[:] = self.modified
```

Mělké a hloubkové kopírování ➤ 146

Při tvorbě objektu typu `AtomicList` ukládáme odkaz na původní seznam a poznámku, zda se má použít mělké kopírování. (Pro seznamy čísel nebo řetězců je mělké kopírování dostatečné, ale pro seznamy obsahující další seznamy nebo jiné kolekce, by již dostatečné nebylo.)

Jakmile je objekt správce kontextu `AtomicList` použit v příkazu `with`, zavolá se jeho metoda `__enter__()`. V tomto okamžiku zkopírujeme původní seznam a vrátíme kopii, takže se všechny změny budou provádět na této kopii.

Jakmile dorazíme na konec oboru platnosti příkazu `with`, zavolá se metoda `__exit__()`. Nedojde-li k žádné výjimce, bude mít parametr `exc_type` (typ výjimky) hodnotu `None`, a my tak víme, že můžeme bezpečně nahradit prvky původního seznamu prvky z upraveného seznamu. (Příkaz `self.original = self.modified` provést nemůžeme, protože tím bychom jen nahradili jeden odkaz na objekt druhým a původní seznam by zůstal beze změny.) Pokud ale došlo k nějaké výjimce, neprovedeme v původním seznamu žádné změny a modifikovaný seznam zahodíme.

Návratová hodnota metody `__exit__()` se použije pro signalizaci, zda se má jakákoli případná výjimka propagovat. Hodnota `True` znamená, že jsme výjimku ošetřili, a proto již není její další propagace nutná. Normálně vracíme vždy hodnotu `False` nebo cokoliv, co se v logickém kontextu vyhodnotí na `False`, aby se případná výjimka propagovala dále. Tím, že explicitní návratovou hodnotu neuvedeme, vrátí naše metoda `__exit__()` hodnotu `None`, která se vyhodnotí na `False` a správně způsobí, že se jakákoli výjimka propaguje.

Vlastní správce kontextu budeme používat v lekci 11 pro zajištění, aby se uzavřela soketová připojení a soubory komprimované pomocí gzip. Dále v lekci 10 použijeme některé správce kontextu z modulu `threading` pro zajištění, aby se odemknuly vzájemně výlučné zámky. Ve cvičeních této lekce pak dostanete šanci vytvořit více generického správce kontextu pro atomické operace.

Deskriptory

Deskriptory jsou třídy, které poskytují řízení přístupu pro atributy jiných tříd. Termínem deskriptor označujeme libovolnou třídu, která implementuje jednu či více speciálních metod pro deskriptory s názvem `__get__()`, `__set__()` a `__delete__()`.

Vestavěné funkce `property()` a `classmethod()` jsou implementovány pomocí deskriptorů. Klíč k pochopení deskriptorů spočívá v tom, že ačkoliv instanci deskriptoru vytváříme ve třídě jako atribut třídy, Python přistupuje k deskriptoru prostřednictvím instancí této třídy.

Abychom si v tom udělali jasno, tak si můžeme představit, že máme třídu, jejíž instance uchovávají nějaké řetězce. K těmto řetězcům chceme přistupovat běžným způsobem, například jako k vlastnostem, kromě toho ale chceme mít kdykoli přístup k verzi řetězců zakódované pro XML. Jednoduchým řešením by bylo při každém nastavení řetězce ihned vytvořit kopii zakódovanou pro XML. Máme-li však tisíce řetězců a verzi pro XML čteme jen pro pár z nich, pak bychom naprosto zbytečně vyplýtvaly velké množství výkonu a paměti. Proto vytvoříme deskriptor, který bude na vyžádání poskytovat řetězce zakódované pro XML, aniž by je ukládal. Začneme úvodní částí definice klientské třídy (vlastníka), což je třída, která deskriptor používá:

```
class Product:
```

```
    __slots__ = ("__name", "__description", "__price")

    name_as_xml = XmlShadow("name")
    description_as_xml = XmlShadow("description")

    def __init__(self, name, description, price):
        self.__name = name
        self.description = description
        self.price = price
```

Jediný kód, který jsme si zde neukázali, jsou vlastnosti. Ty jsme vytvořili obvyklým způsobem, přičemž název má podobu vlastnosti `name` určené pouze pro čtení a popis a cena jsou vlastnosti `description` a `price`, které je možné číst i zapisovat. (Veškerý kód najdete v souboru `XmlShadow.py`.) Proměnnou `__slots__` používáme pro zajištění, aby třída neměla atribut `__dict__`, díky čemuž bude schopná uchovávat pouze tři námi stanovené soukromé atributy, což ale nijak nesouvisí s použitím našich deskriptorů. Atributy třídy `name_as_xml` a `description_as_xml` nastavujeme jako instance deskriptoru `XmlShadow`. Přestože žádný objekt typu `Product` nemá atribut `name_as_xml` ani `description_as_xml`, můžeme díky deskriptoru napsat následující kód (který jsme převzali z dokumentačních testů modulu):

```
>>> product = Product("Dláto <3cm>", "Dláto & násada", 45.25)
>>> product.name, product.name_as_xml, product.description_as_xml
('Dláto <3cm>', 'Dláto &lt;3cm&gt;', 'Dláto &amp; násada')
```

Tento kód funguje díky tomu, že při pokusu o přístup třeba k atributu `name_as_xml` Python zjistí, že třída `Product` má deskriptor tohoto jména, a proto jej použije pro získání hodnoty tohoto atributu. Zde je kompletní kód pro deskriptorovou třídu `XmlShadow`:

```
class XmlShadow:

    def __init__(self, attribute_name):
        self.attribute_name = attribute_name


    def __get__(self, instance, owner=None):
        return xml.sax.saxutils.escape(
                              getattr(instance, self.attribute_name))
```

Při vytváření objektů `name_as_xml` a `description_as_xml` předáváme inicializační metodě třídy `XmlShadow` název odpovídajícího atributu třídy `Product`, aby tak deskriptor věděl, se kterými atributy má pracovat. Python pak při hledání atributu `name_as_xml` nebo `description_as_xml` zavolá metodu deskriptoru s názvem `__get__()`. Argument `self` je instance deskriptoru, argument `instance` je instance třídy `Product` (tj. objekt `self` daného produktu) a argument `owner` je vlastnící třída (v tomto případě `Product`). Pro získání příslušného atributu z produktu (v tomto případě příslušné vlastnosti) používáme funkci `getattr()`, načež vrátíme jeho verzi zakódovanou pro XML.

Pokud se nacházíme v situaci, kdy se k řetězcům pro XML přistupuje jen u velmi malé části produktů, avšak tyto řetězce jsou dlouhé a často se přistupuje ke tstejným, pak můžeme použít mezipaměť. Například:

```
class CachedXmlShadow:

    def __init__(self, attribute_name):
        self.attribute_name = attribute_name
        self.cache = {}


    def __get__(self, instance, owner=None):
        xml_text = self.cache.get(id(instance))
        if xml_text is not None:
            return xml_text
        return self.cache.setdefault(id(instance),
                xml.sax.saxutils.escape(
                            getattr(instance, self.attribute_name)))
```

Jedinečnou identitu instance neuchováváme jako samotnou instanci, ale jako klíč, protože klíče slovníku musejí být hashovatelné (což identifikační kódy jsou), a to je požadavek, který si na třídách používajících deskriptor `CachedXmlShadow` nechceme vynucovat. Klíč je nezbytný, protože deskriptory se nevytvářejí pro každou instanci, ale pro třídu jako celek. (Metoda `dict.setdefault()` obyčejně vrací hodnotu zadaného klíče nebo vytvoří nový prvek se zadaným klíčem a hodnotou, pokud prvek s tímto klíčem dosud neexistuje, přičemž vrátí tuto hodnotu.)

Viděli jsme deskriptory použité pro generování dat bez nutnosti jejich ukládání, a proto se nyní podíváme na deskriptor, který lze použít pro uložení dat všech atributů objektu, přičemž objekt sám nemusí uchovávat vůbec nic. V tomto příkladu použijeme pouze slovník, avšak v realističtějším kontextu bychom mohli data ukládat do souboru nebo databáze. Zde je začátek upravené verze třídy `Point`, která využívá náš nový deskriptor (ze souboru `ExternalStorage.py`):

```
class Point:

    __slots__ = ()
    x = ExternalStorage("x")
    y = ExternalStorage("y")

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

Nastavením atributu třídy `__slots__` na prázdnou n-tici zajistíme, že třída nebude moci uchovávat vůbec žádné datové atributy. Při přiřazení atributu `self.x` Python zjistí, že existuje deskriptor s názvem „x“, a proto použije jeho metodu `__set__()`. Zbývající část třídy si neukážeme, protože je stejná jako původní třída `Point` z lekce 6. Zde je kompletní deskriptorová třída `ExternalStorage`:

```
class ExternalStorage:

    __slots__ = ("attribute_name",)
    __storage = {}


    def __init__(self, attribute_name):
        self.attribute_name = attribute_name


    def __set__(self, instance, value):
        self.__storage[id(instance), self.attribute_name] = value


    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return self.__storage[id(instance), self.attribute_name]
```

Každý objekt typu `ExternalStorage` má jediný datový atribut `attribute_name`, který uchovává název datového atributu vlastnické třídy. Kdykoliv je nějaký atribut nastaven, uloží se jeho hodnota do soukromého slovníku `__storage`. Podobně se při každém čtení hodnoty atributu získá jeho hodnota ze slovníku `__storage`.

Stejně jako u všech deskriptorových metod je i zde argument `self` instancí objektu deskriptoru a argument `instance` objekt `self` toho objektu, který obsahuje daný deskriptor, takže zde je argument `self` objektem typu `ExternalStorage` a instance objektem typu `Point`.

Přestože `__storage` je atributem třídy, můžeme k němu přistupovat pomocí `self.__storage` (stejně jako můžeme volat metody třídy pomocí `self.method()`), protože Python jej nejdříve začne hledat jako atribut instance, nenalezne jej, a tak jej začne hledat jako atribut třídy. Jedinou (teoretickou) nevýhodou tohoto přístupu je, že pokud bychom měli atribut třídy a atribut instance téhož jména, pak by jeden zastínil ten druhý. (Pokud by to opravdu představovalo problém, mohli bychom se vždy na atribut třídy odkázat pomocí třídy, to znamená `ExternalStorage.__storage`. Přímé uvádění názvu třídy v kódu si z obecného hlediska sice moc nerozumí s vytvářením podtříd, ale u soukromých atributů na tom ve skutečnosti nezáleží, protože v jejich případě Python název třídy stejně zkomolí.)

Implementace speciální metody `__get__()` je ve srovnání s předchozím příkladem malinko sofistikovanější, protože musíme implementovat určitý způsob přístupu k samotné instanci třídy `ExternalStorage`. Máme-li například `p = Point(3, 4)`, pak můžeme k souřadnici `x` přistupovat pomocí příkazu `p.x`, přičemž k objektu typu `ExternalStorage`, který uchovává všechny souřadnice `x`, můžeme přistupovat pomocí příkazu `Point.x`.

K dokončení našeho výkladu deskriptorů vytvoříme deskriptor `Property`, který napodobuje chování vestavěné funkce `property()`, tedy alespoň co se týče zápisu a čtení vlastností. Kód najdete v souboru `Property.py`. Zde je kompletní třída `NameAndExtension`, která jej využívá.

```
class NameAndExtension:

    def __init__(self, name, extension):
        self.__name = name
        self.extension = extension

    @Property               # Používá vlastní deskriptor Property
    def name(self):
        return self.__name

    @Property               # Používá vlastní deskriptor Property
    def extension(self):
        return self.__extension

    @extension.setter       # Používá vlastní deskriptor Property
    def extension(self, extension):
        self.__extension = extension
```

Z hlediska použití je deskriptor `Property` stejný jako vestavěný dekorátor `@property` a dekorátor `@názevVlasntosti.setter`. Níže je uveden začátek implementace deskriptoru `Property`:

```
class Property:

    def __init__(self, getter, setter=None):
        self.__getter = getter
        self.__setter = setter
        self.__name__ = getter.__name__
```

Inicializační metoda třídy přijímá jako své argumenty jednu nebo dvě funkce. Je-li použita jako dekorátor, pak obdrží pouze dekorovanou funkci, která se stane funkcí pro čtení vlastnosti, zatímco funkce pro zápis vlastnosti se nastaví na hodnotu `None`. Pro název vlastnosti používáme název funkce pro čtení této vlastnosti. Pro každou vlastnost tedy máme funkci pro čtení, případně funkci pro zápis a název.

```
    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return self.__getter(instance)
```

Při přístupu k vlastnosti vracíme výsledek volání funkce pro její čtení, které předáváme instanci jako první argument. Na první pohled vypadá příkaz `self.__getter()` jako volání metody, ve skutečnosti tomu tak ale není. Výraz `self.__getter` je totiž atribut, který uchovává odkaz na objekt ukazující na metodu, která byla dříve zadána. Stane se tedy to, že nejdříve vezmeme atribut (`self.__getter`), který poté zavoláme jako funkci `()`. A protože jej zavoláme jako funkci, a ne jako metodu, musíme jí sami explicitně předat příslušný objekt `self`. V případě deskriptoru se tento objekt `self` (ze třídy, která používá deskriptor) nazývá `instance` (protože `self` je objekt deskriptoru). Totéž se vztahuje i na metodu `__set__()`.

```
    def __set__(self, instance, value):
        if self.__setter is None:
            raise AttributeError("'{0}' je pouze pro čtení".format(
                                 self.__name__))
        return self.__setter(instance, value)
```

Pokud nebyla zadána žádná funkce pro zápis vlastnosti, vyvoláme výjimku `AttributeError`. V opačném případě zavoláme funkci pro zápis vlastnosti s argumentem `instance` a novou hodnotou.

```
    def setter(self, setter):
        self.__setter = setter
        return self.__setter
```

Tato metoda se volá v okamžiku, kdy interpret narazí například na `@extension.setter`, přičemž jako argument `setter` obdrží dekorovanou funkci. Tuto funkci uloží (a od tohoto okamžiku ji může použít v metodě `__set__()`) a vrátí, poněvadž dekorátory by měly vracet funkci či metodu, kterou dekorují.

Viděli jsme tři docela odlišná použití deskriptorů. Deskriptory jsou velmi mocné a flexibilní nástroje, pomocí nichž lze na pozadí provádět množství práce, zatímco ve své klientské (vlastnické) třídě vypadají jen jako obyčejné atributy.

Dekorátory tříd

Dekorátory můžeme vytvářet nejen pro funkce a metody, ale také pro celé třídy. Dekorátory tříd přijímají objekt představující třídu (výsledek příkazu `class`) a měly by vracet třídu – obvykle upravenou verzi třídy, kterou dekorují. V tomto oddílu prostudujeme dva dekorátory třídy, abychom si ukázali, jak se implementují.

V lekci 6 jsme vytvořili vlastní třídu `SortedList` představující kolekci, která agregovala holý seznam jako soukromý atribut `self.__list`. Osm metod třídy `SortedList` jen předává práci tomuto soukromému atributu. Zde je kupříkladu implementace metod `SortedList.clear()` a `SortedList.pop()`:

SortedList ➤ 262

```
    def clear(self):
        self.__list = []


    def pop(self, index=-1):
        return self.__list.pop(index)
```

S metodou `clear()` nemůžeme udělat nic, protože neexistuje odpovídající metoda pro typ `list`, ale v případě metody `pop()` a dalších šesti metod delegovaných třídou `SortedList` můžeme jednoduše zavolat odpovídající metodu třídy `list`. To můžeme provést pomocí dekorátoru třídy `@delegate` z našeho modulu `Util`. Zde je začátek nové verze třídy `SortedList`:

```
@Util.delegate("__list", ("pop", "__delitem__", "__getitem__",
                          "__iter__", "__reversed__", "__len__", "__str__"))
class SortedList:
```

Prvním argumentem je název atributu, který se má použít pro delegování, a druhým argumentem je posloupnost jedné či více metod, které má dekorátor `delegate()` implementovat za nás, abychom tuto práci nemuseli provádět sami. Tento přístup používá třída `SortedList` v souboru `SortedListDelegate.py`, a proto nemá pro tyto metody žádný kód, přestože je plně podporuje. Zde je dekorátor třídy, který tyto metody implementuje:

```
def delegate(attribute_name, method_names):
    def decorator(cls):
        nonlocal attribute_name
        if attribute_name.startswith("__"):
            attribute_name = "_" + cls.__name__ + attribute_name
        for name in method_names:
            setattr(cls, name, eval("lambda self, *a, **kw: "
                                    "self.{0}.{1}(*a, **kw)".format(
                                    attribute_name, name)))
        return cls
    return decorator
```

Obyčejný dekorátor použít nemůžeme, protože potřebujeme dekorátoru předat argumenty, a proto jsme vytvořili funkci, která přijímá naše argumenty a vrací dekorátor třídy. Samotný dekorátor přijímá jediný argument ve formě třídy (tedy podobně jako dekorátor funkce, který přijímá jediný argument ve formě funkce či metody).

Musíme použít příkaz `nonlocal`, protože vnořená funkce používá proměnnou `attribute_name` z vnějšího oboru platnosti. S ohledem na komolení názvů soukromých atributů musíme být schopni v případě nutnosti název atributu opravit. Chování dekorátoru je docela jednoduché. Prochází přes všechny názvy metod předaných funkci `delegate()` a pro každou z nich vytváří novou metodu, kterou nastavuje se zadaným názvem metody jako atribut v dané třídě.

Pro vytvoření každé z delegovaných metod používáme funkce `eval()`, protože ji můžeme použít k provedení jediného příkazu a příkaz `lambda` vytváří metodu či funkci. Například kód provedený k vytvoření metody `pop()` vypadá takto:

```
lambda self, *a, **kw: self._SortedList__list.pop(*a, **kw)
```

Pro argumenty používáme `*` a `**`, aby bylo možné zpracovat libovolné argumenty, ačkoliv delegované metody mají specifický seznam argumentů. Kupříkladu metoda `list.pop()` přijímá jedinou indexovou pozici (nebo nic, což znamená, že se použije poslední prvek). To je v pořádku, protože pokud dojde k zadání nesprávného počtu nebo druhu argumentů, vyvolá volaná metoda seznamu odpovídající výjimku.

FuzzyBool ➤ 243

Druhou dekorátorovou třídu, na kterou se podíváme, jsme také použili v lekci 6. Při implementaci třídy `FuzzyBool` jsme si řekli, že jsme uvedli pouze speciální metody `__lt__()` a `__eq__()` (pro `<` a `==`) a nechali si všechny ostatní porovnávací metody automaticky vygenerovat. Neukázali jsme si však celý začátek definice třídy:

```
@Util.complete_comparisons
class FuzzyBool:
```

Další čtyři porovnávací operátory poskytl dekorátor třídy s názvem `complete_comparisons()`. Máme-li třídu, která definuje pouze operátor < (nebo < a ==), pak tento dekorátor vytvoří na základě následujících logických ekvivalencí chybějící operátory:

$x = y \Leftrightarrow \neg(x < y \vee y < x)$
$x \neq y \Leftrightarrow \neg(x = y)$
$x > y \Leftrightarrow y < x$
$x \leq y \Leftrightarrow \neg(y < x)$
$x \geq y \Leftrightarrow \neg(x < y)$

Má-li dekorovaná třída operátory < a ==, použije dekorátor oba dva. Je-li k dispozici pouze operátor <, vystačí si pouze s ním. (Ve skutečnosti je to samotný Python, který automaticky doplní operátor >, je-li k dispozici operátor <, operátor != z operátoru == a operátor >= z operátoru <=, takže stačí implementovat pouze volné operátory <, <= a == a nechat Python, aby odvodil ostatní. Nicméně díky tomuto dekorátoru třídy snižujeme potřebné minimum jen na operátor <. To je na jednak pohodlné a jednak máme tímto způsobem zajištěno, že všechny porovnávací operátory používají stejnou konzistentní logiku.)

```
def complete_comparisons(cls):
    assert cls.__lt__ is not object.__lt__, (
            "{0} musí definovat < a ideálně ==".format(cls.__name__))
    if cls.__eq__ is object.__eq__:
        cls.__eq__ = lambda self, other: (not
                (cls.__lt__(self, other) or cls.__lt__(other, self)))
    cls.__ne__ = lambda self, other: not cls.__eq__(self, other)
    cls.__gt__ = lambda self, other: cls.__lt__(other, self)
    cls.__le__ = lambda self, other: not cls.__lt__(other, self)
    cls.__ge__ = lambda self, other: not cls.__lt__(self, other)
    return cls
```

Jeden problém, kterému musí tento dekorátor čelit, spočívá v tom, že třída `object`, od které jsou vposledku odvozeny všechny ostatní třídy, definuje všech šest porovnávacích operátorů tak, že při svém použití vyvolají výjimku `TypeError`. Potřebujeme tedy vědět, zda byly operátory < a == reimplementovány (a zda jsou tedy použitelné). To lze snadno provést porovnáním příslušných speciálních metod v dekorované třídě s jejich protějšky ve třídě `object`.

Pokud dekorovaná třída nemá vlastní operátor <, příkaz `assert` selže, poněvadž se jedná o minimální požadavek dekorátoru. A pokud je k dispozici vlastní operátor ==, tak jej použijeme. V opačném případě vytvoříme nový. Potom vytvoříme všechny ostatní metody a vrátíme dekorovanou třídu, která nyní obsahuje všech šest porovnávacích metod.

Použití dekorátorů tříd představuje pravděpodobně ten nejjednodušší a nejpřímější způsob změny tříd. Další možností je sáhnout po metatřídách, což je téma, jemuž se budeme věnovat v pozdější části této lekce.

Metatřídy ➤ 377

Abstraktní bázové třídy

Abstraktní bázová třída je třída, kterou nelze použít pro vytvoření objektů. Účelem takovýchto tříd je totiž definovat rozhraní, což znamená uvést metody a vlastnosti, které musejí poskytovat třídy, které od dané abstraktní bázové třídy dědí. To je užitečné, protože můžeme použít abstraktní bázovou třídu jako jakýsi druh slibu – slibu, že každá odvozená třída bude poskytovat metody a vlastnosti specifikované abstraktní bázovou třídou.*

Abstraktní bázové třídy jsou takové třídy, které mají alespoň jednu abstraktní metodu či vlastnost. Abstraktní metody lze definovat bez implementace (tzn. jejich sada obsahuje příkaz `pass` nebo vyvolá výjimku `NotImplementedError()`, pokud si chceme vynutit reimplementaci v podtřídě) nebo se skutečnou (určitou) implementací, kterou lze vyvolat z podtříd, například v situaci, kdy existuje nějaký společný případ. Abstraktní bázové třídy mohou obsahovat i konkrétní (tj. neabstraktní) metody a vlastnosti.

Metatřídy ➢ 377

Třídy, které jsou odvozené od abstraktní bázové třídy, můžeme použít k tvorbě instancí pouze tehdy, když reimplementují všechny zděděné abstraktní metody a vlastnosti. U těch abstraktních metod, které mají konkrétní implementace (i kdyby šlo jen o příkaz `pass`), může odvozená třída k zavolání verze abstraktní bázové třídy jednoduše použít funkci `super()`. Jakékoli konkrétní metody a vlastnosti jsou jako obvykle dostupné prostřednictvím dědičnosti. Všechny abstraktní bázové třídy musejí mít metatřídy typu `abc.ABCMeta` (z modulu `abc`) nebo některého ze svých podtříd. Metatřídy budeme probírat o něco později.

Python nabízí dvě skupiny abstraktních bázových tříd, jednu v modulu `collections` a druhou v modulu `numbers`. Umožňují nám pokládat otázky o objektech. Máme-li například proměnnou `x`, pak můžeme pomocí příkazu `isinstance(x, collections.MutableSequence)` zjistit, zda jde o posloupnost, nebo pomocí příkazu `isinstance(x, numbers.Integral)`, zda jde o celé číslo. To je zvláště užitečné vzhledem k dynamickému typování jazyka Python, u něhož nemusíme nutně vědět (nebo nás to nemusí zajímat), o jaký typ objektu se jedná, ale chceme vědět, zda podporuje určité operace, které na něj potřebujeme aplikovat. Abstraktní bázové třídy modulů `numeric` a `collection` jsou uvedeny v tabulkách 8.3 a 8.4. Další důležitou abstraktní bázovou třídou je třída `io.IOBase`, od níž jsou odvozeny všechny třídy pro práci se soubory a datovými proudy.

Tabulka 8.3: Abstraktní bázové třídy modulu numbers

Abstraktní bázová třída	Odvozena od	Rozhraní API	Příklady
Number	object		`complex`, `decimal.Decimal`, `float`, `fractions.Fraction`, `int`
Complex	Number	`==`, `!=`, `+`, `-`, `*`, `/`, `abs()`, `bool()`, `complex()`, `conjugate()` a také vlastnosti `real` a `imag`	`complex`, `decimal.Decimal`, `float`, `fractions.Fraction`, `int`

* Abstraktní bázové třídy Pythonu jsou popsány v dokumentu PEP 3119 (*www.python.org/dev/peps/pep-3119*), který kromě toho obsahuje velice užitečný výklad a rozhodně stojí za přečtení.

Abstraktní bázová třída	Odvozena od	Rozhraní API	Příklady
Real	Complex	<, <=, ==, !=, >=, >, +, -, *, /, //, %, abs(), bool(), complex(), conjugate(), divmod(), float(), math.ceil(), math.floor(), round(), trunc() a také vlastnosti real a imag	decimal.Decimal, float, fractions.Fraction, int
Rational	Real	<, <=, ==, !=, >=, >, +, -, *, /, //, %, abs(), bool(), complex(), conjugate(), divmod(), float(), math.ceil(), math.floor(), round(), trunc()a také vlastnosti real, imag, numerator a denominator	fractions.Fraction, int
Integral	Rational	<, <=, ==, !=, >=, >, +, -, *, /, //, %, <<, >>, ~, &, ^, |, abs(), bool(), complex(), conjugate(), divmod(), float(), math.ceil(), math.floor(), pow(), round(), trunc()a také vlastnosti real, imag, numerator a denominator	int

Tabulka 8.4: Hlavní abstraktní bázové třídy modulu collections

Abstraktní bázová třída	Odvozena od	Rozhraní API	Příklady
Callable	object	()	Všechny běžné funkce, metody a lambda funkce
Container	object	in	bytearray, bytes, dict, frozenset, list, set, str, tuple
Hashable	object	hash()	bytes, frozenset, str, tuple
Iterable	object	iter()	bytearray, bytes, collections.deque, dict, frozenset, list, set, str, tuple
Iterator	Iterable	iter(), next()	

Abstraktní bázová třída	Odvozena od	Rozhraní API	Příklady
Sized	object	len()	bytearray, bytes, collections.deque, dict, frozenset, list, set, str, tuple
Mapping	Container, Iterable, Sized	==, !=, [], len(), iter(), in, get(), items(), keys(), values()	dict
MutableMapping	Mapping	==, !=, [], len(), iter(), in, clear(), get(), items(), keys(), values(), pop(), popitem(), setdefault(), update()	dict
Sequence	Container, Iterable, Sized	[], len(), iter(), reversed(), in, count(), index()	bytearray, bytes, list, str, tuple
MutableSequence	Container, Iterable, Sized	[], +=, del, len(), iter(), reversed(), in, append(), count(), extend(), index(), insert(), pop(), remove(), reverse()	bytearray, list
Set	Container, Iterable, Sized	<, <=, ==, !=, =>, >, &, |, ^, len(), iter(), in, isdisjoint()	frozenset, set
MutableSet	Set	<, <=, ==, !=, =>, >, &, |, ^, &=, |=, ^=, -=, len(), iter(), in, add(), clear(), discard(), isdisjoint(), pop(), remove()	set

K plné integraci svých vlastních tříd představujících čísla nebo kolekce musíme tyto třídy vytvářet tak, aby pasovaly k standardním abstraktním bázovým třídám. Například třída `SortedList` je posloupnost, ovšem s její současnou implementací vrátí volání `isinstance(L, collections.Sequence)` hodnotu `False`, je-li `L` objekt typu `SortedList`. Tento problém můžeme snadno vyřešit odvozením od příslušné abstraktní bázové třídy:

```
class SortedList(collections.Sequence):
```

Metatřídy ➢ 377

Jakmile tedy vezmeme jako bázovou třídu `collections.Sequence`, bude výše uvedené volání funkce `isinstance()` od nynějška vracet hodnotu `True`. Kromě toho se ale po nás žádá, abychom implementovali metody `__init__()` (nebo `__new__()`), `__getitem__()` a `__len__()` (což je splněno). Abstraktní bázová třída `collections.Sequence` dále poskytuje konkrétní (tj. neabstraktní) implementace pro metody `__contains__()`, `__iter__()`, `__reversed__()`, `count()` a `index()`. Ve tří-

dě `SortedList` je sice všechny reimplementujeme, pokud bychom ale chtěli, mohli bychom použít verze abstraktní bázové třídy, k čemuž by stačilo jen neprovést jejich reimplementaci. Přestože je seznam reprezentovaný třídou `SortedList` měnitelný, nemůžeme ji odvodit od třídy `collections.MutableSequence`, protože třída `SortedList` nemá všechny metody, které musí typ `collections.MutableSequence` poskytovat. Těmito chybějícími metodami jsou `__setitem__()` a `append()`. (Kód pro tuto třídu `SortedList` je v souboru `SortedListAbc.py`. Alternativní postup přetvoření třídy `SortedList` na typ `collections.Sequence` si ukážeme v oddílu „Metatřídy".)

Viděli jsme, jak vytvořit vlastní třídu napasovanou na standardní abstraktní bázové třídy, a proto se nyní obrátíme k dalšímu použití abstraktních bázových tříd, které spočívá v poskytování příslibu určitého rozhraní pro naše vlastní třídy. Podíváme se na tři odlišné příklady – na nichž si ukážeme různé stránky tvorby a použití abstraktních bázových tříd.

Začneme velice jednoduchým příkladem, který ukazuje, jak pracovat s vlastnostmi určenými pro čtení i zápis. Níže uvedená třída představuje domácí spotřebiče. Každý vytvořený spotřebič musí mít řetězec s názvem modelu (`model`) určený pouze pro čtení a cenu (`price`), kterou lze číst i zapisovat. Dále chceme mít jistotu, že se bude reimplementovat metoda `__init__()` naší abstraktní bázové třídy. Její kód je uveden níže (najdete jej také v souboru `Appliance.py`). Neukázali jsme si zde příkaz `import abc`, který je nezbytný pro funkce `abstractmethod()` a `abstractproperty()`, které lze použít jako dekorátory:

```
class Appliance(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def __init__(self, model, price):
        self.__model = model
        self.price = price

    def get_price(self):
        return self.__price

    def set_price(self, price):
        self.__price = price

    price = abc.abstractproperty(get_price, set_price)

    @property
    def model(self):
        return self.__model
```

Pro naši třídu jsme nastavili metatřídu na `abc.ABCMeta`, což je pro abstraktní bázové třídy nezbytné. Samozřejmě bychom mohli použít kteroukoli z tříd odvozených od třídy `abc.ABCMeta`. Z metody `__init__()` jsme udělali abstraktní metodu, kterou je třeba reimplementovat, a přitom jsme uvedli implementaci, kterou může (ale nemusí) odvozená třída zavolat. Pro vytvoření čitelné a zapisovatelné abstraktní vlastnosti nemůžeme použít syntaxi dekorátoru. Kromě toho jsme pro metody pro čtení a zapisování vlastnosti nepoužili soukromé názvy, což by bylo pro podtřídy nepohodlné.

Vlastnost `price` je abstraktní (a proto nemůžeme použít dekorátor `@property`) a lze ji číst i zapisovat. Postupujeme zde způsobem, který je běžný, máme-li soukromá data, která lze zapisovat i číst (např. `__price`), ve formě vlastnosti. Místo přímého zápisu do soukromých dat jsme totiž tuto vlastnost inicializovali v metodě `__init__()`, díky čemuž máme jistotu, že se zavolá metoda pro zápis této vlastnosti (která tak může provést případnou validaci nebo jinou činnost, i když v tomto příkladu nic takového nedělá).

Vlastnost `model` není abstraktní, podtřídy ji tedy nemusejí reimplementovat a my ji můžeme vytvořit pomocí dekorátoru `@property`. Postupujeme zde způsobem, který je obvyklý, máme-li soukromá data určená pouze pro čtení (např. `__model`) ve formě vlastnosti. Hodnotu atributu `__model` nastavíme jednou v metodě `__init__()` a vytvoříme k ní přístup prostřednictvím vlastnosti `model` určené pouze pro čtení.

Všimněte si, že objekty typu `Appliance` vytvářet nemůžeme, protože tato třída obsahuje abstraktní atributy. Zde je příklad podtřídy:

```
class Cooker(Appliance):

    def __init__(self, model, price, fuel):
        super().__init__(model, price)
        self.fuel = fuel

    price = property(lambda self: super().price,
                     lambda self, price: super().set_price(price))
```

Třída `Cooker` musí reimplementovat metodu `__init__()` a vlastnost `price`. V případě vlastnosti jsme prostě celou práci nechali na bázové třídě. Vlastnost `model` určená pouze pro čtení je zděděná. Mohli bychom takto vytvářet mnoho tříd založených na třídě `Appliance`, jako je například `Fridge`, `Toaster` a tak dále.

Další abstraktní bázová třída, na kterou se nyní podíváme, je dokonce ještě kratší. Jedná se o abstraktní bázovou třídu pro funktory filtrující text (v souboru `TextFilter.py`):

```
class TextFilter(metaclass=abc.ABCMeta):

    @abc.abstractproperty
    def is_transformer(self):
        raise NotImplementedError()

    @abc.abstractmethod
    def __call__(self):
        raise NotImplementedError()
```

Abstraktní bázová třída `TextFilter` neposkytuje vůbec žádnou funkčnost. Smyslem její existence je pouze definovat jisté rozhraní, v tomto případě vlastnost `is_transformer` určenou jen pro čtení a metodu `__call__()`, které musejí poskytovat všechny její podtřídy. Tato abstraktní vlastnost a metoda nemají žádné implementace, a proto nechceme, aby je podtřídy volaly. Z tohoto důvodu

místo neškodného příkazu `pass` vyvoláme při jejich použití (např. přes volání `super()`) výjimku `NotImplementedError`.

Zde je jedna jednoduchá podtřída:

```
class CharCounter(TextFilter):

    @property
    def is_transformer(self):
        return False

    def __call__(self, text, chars):
        count = 0
        for c in text:
            if c in chars:
                count += 1
        return count
```

Tento filtr textu není ve skutečnosti žádným transformátorem, protože místo transformování zadaného textu jednoduše vrátí počet uvedených znaků v textu. Zde je příklad použití této třídy:

```
vowel_counter = CharCounter()
vowel_counter("psi jedou a muchy ne", "aeiyou")    # vrátí: 8
```

K dispozici jsou také další dva filtry textu `RunLengthEncode` a `RunLengthDecode`, které tentokrát fungují jako transformátory. Jejich praktické použití vypadá takto:

```
rle_encoder = RunLengthEncode()
rle_text = rle_encoder(text)
...
rle_decoder = RunLengthDecode()
original_text = rle_decoder(rle_text)
```

Třída `RunLengthEncode` převádí řetězec do bajtů zakódovaných dle UTF-8, přičemž bajty `0x00` nahrazuje posloupností `0x00`, `0x01`, `0x00` a libovolnou posloupnost 3 až 255 opakujících se bajtů posloupností `0x00`, *`počet`*, *`bajt`*. Pokud zadaný řetězec obsahuje množství čtyř a více stejných, opakujících se a po sobě jdoucích znaků, pak lze tímto způsobem vytvořit ve srovnání s holým kódováním bajtů dle UTF-8 kratší bajtový řetězec. Třída `RunLengthDecode` přijímá takto zakódovaný bajtový řetězec a vrací původní řetězec. Zde je začátek třídy `RunLengthDecode`:

```
class RunLengthDecode(TextFilter):

    @property
    def is_transformer(self):
        return True

    def __call__(self, rle_bytes):
        ...
```

Vynechali jsme tělo metody `__call__()`, které najdete ve zdrojových kódech k této knize. Třída `RunLengthEncode` má naprosto stejnou strukturu.

Poslední abstraktní bázová třída, na kterou se podíváme, poskytuje rozhraní API a výchozí implementaci pro mechanizmus navrácení provedené činnosti zpět (undo). Zde je kompletní abstraktní bázová třída (ze souboru `Abstract.py`):

```
class Undo(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def __init__(self):
        self.__undos = []

    @abc.abstractproperty
    def can_undo(self):
        return bool(self.__undos)

    @abc.abstractmethod
    def undo(self):
        assert self.__undos, "nic dalšího už nelze vrátit zpět"
        self.__undos.pop()(self)

    def add_undo(self, undo):
        self.__undos.append(undo)

    def clear(self):        #   ve třídě Undo
        self.__undos = []
```

Metody `__init__()` a `undo()` musejí být reimplementovány, protože obě jsou abstraktní. Stejně tak musí být reimplementována také vlastnost `can_undo` určená pouze pro čtení. Podtřídy nemusejí reimplementovat metodu `add_undo()`, na druhou stranu jim v tom ale nic nebrání. Metoda `undo()` je malinko zákeřná. Seznam `self.__undos` by měl uchovávat odkazy na objekty ukazující na metody. Každá metoda musí při svém zavolání způsobit anulování odpovídající akce, což bude jasnější, až se za okamžik podíváme na podtřídu třídy `Undo`. K provedení anulování tedy vyjmeme ze seznamu `self.__undos` poslední metodu, kterou poté zavoláme jako funkci a předáme jí objekt `self` jako argument. (Objekt `self` předat musíme, protože v tomto případě voláme metodu jako funkci, ne jako metodu.)

Zde je začátek třídy `Stack`, která je odvozena od třídy `Undo`, takže jakékoli akce provedené na jejích objektech lze anulovat zavoláním metody `Stack.undo()` bez argumentů:

```
class Stack(Undo):

    def __init__(self):
        super().__init__()
        self.__stack = []
```

```
    @property
    def can_undo(self):
        return super().can_undo

    def undo(self):
        super().undo()

    def push(self, item):
        self.__stack.append(item)
        self.add_undo(lambda self: self.__stack.pop())

    def pop(self):
        item = self.__stack.pop()
        self.add_undo(lambda self: self.__stack.append(item))
        return item
```

Metody `Stack.top()` a `Stack.__str__()` jsme vynechali, protože v žádné z nich pro nás není nic nového a navíc žádná nepracuje s bázovou třídou `Undo`. V případě vlastnosti `can_undo` a metody `undo()` jednoduše předáme práci bázové třídě. Pokud bychom je nedefinovali jako abstraktní, nemuseli bychom je vůbec reimplementovat a dosáhli bychom stejného efektu. Avšak v tomto případě jsme chtěli přinutit podtřídy, aby je reimplementovaly a byly si tak vědomy případného anulování akcí. V metodách `push()` a `pop()` provádíme příslušné operace a navíc přidáme do seznamu anulačních funkcí funkci, která provede anulování právě provedené operace.

Abstraktní bázové třídy jsou nejužitečnější v rozsáhlých programech, knihovnách a aplikačních rámcích, kde mohou pomoci zajistit vzájemnou spolupráci tříd bez ohledu na implementační detaily nebo jejich autora, což je dáno tím, že poskytují rozhraní API, která stanoví jejich abstraktní bázové třídy.

Vícenásobná dědičnost

Vícenásobná dědičnost nastává tehdy, když jedna třída dědí od dvou či více jiných tříd. Třebaže Python (a např. C++) vícenásobnou dědičnost plně podporuje, některé jazyky (především Java) ji nepovolují. Problém totiž tkví v tom, že vícenásobná dědičnost může vést k tomu, že se od stejné třídy dědí více než jedenkrát (např. pokud jsou dvě z bázových tříd odvozeny od téže třídy), což znamená, že verze volané metody, není-li v podtřídě, ale ve dvou či více bázových třídách (nebo jejich bázových třídách atd.), závisí na pořadí vyhodnocování metod, čímž se třídy, které používají vícenásobnou dědičnost, mohou stát nestálé.

Vícenásobnou dědičnost můžeme obecně obejít prostřednictvím jednoduché dědičnosti (jedna bázová třída) a nastavení metatřídy, chceme-li podporovat další rozhraní API, protože jak uvidíme v následujícím oddílu, pomocí metatřídy můžeme nabídnout příslib rozhraní, aniž bychom skutečně zdědili nějaké metody či datové atributy. Další možností je použít vícenásobnou dědičnost s jednou konkrétní třídou a jednou či více abstraktními bázovými třídami pro dodatečná rozhraní API. A další možnost spočívá v použití jednoduché dědičnosti a agregování instancí dalších tříd.

Nicméně v některých případech může vícenásobná dědičnost nabízet velice pohodlné řešení. Představte si například, že chcete vytvořit novou verzi třídy `Stack` z předchozího oddílu, ale zároveň chcete, aby tato třída podporovala načítání a ukládání pomocí naložených objektů (pickle). Funkčnost pro načítání a ukládání by se mohla hodit i v dalších třídách, a proto ji implementujeme v samostatné třídě:

```
class LoadSave:

    def __init__(self, filename, *attribute_names):
        self.filename = filename
        self.__attribute_names = []
        for name in attribute_names:
            if name.startswith("__"):
                name = "_" + self.__class__.__name__ + name
            self.__attribute_names.append(name)

    def save(self):
        with open(self.filename, "wb") as fh:
            data = []
            for name in self.__attribute_names:
                data.append(getattr(self, name))
            pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)

    def load(self):
        with open(self.filename, "rb") as fh:
            data = pickle.load(fh)
            for name, value in zip(self.__attribute_names, data):
                setattr(self, name, value)
```

Tato třída má dva atributy: atribut `filename` je veřejný, takže jej lze kdykoliv změnit, a atribut `__attribute_names`, který je fixovaný, takže jej lze nastavit pouze při vytváření instance. Metoda `save()` prochází všechny názvy atributů a vytváří seznam s názvem data, který uchovává hodnoty každého z atributů, který se má uložit. Seznam `data` pak uloží do naloženého objektu. Příkaz `with` zajistí, že se soubor uzavře, pokud byl úspěšně otevřen, a že se jakékoli výjimky souboru či naloženého objektu předají volajícímu. Metoda `load()` prochází názvy atributů a odpovídající datové prvky, které byly načteny, a nastavuje každý atribut na jeho načtenou hodnotu.

Zde je začátek třídy `FileStack`, která používá vícenásobnou dědičnost. Tato třída je odvozena od třídy `Undo` z předchozího oddílu a od třídy `LoadSave` z tohoto oddílu:

```
class FileStack(Undo, LoadSave):

    def __init__(self, filename):
        Undo.__init__(self)
        LoadSave.__init__(self, filename, "__stack")
        self.__stack = []
```

```
    def load(self):
        super().load()
        self.clear()
```

Zbývající část této třídy je stejná jako třída `Stack`, a proto ji zde opakovat nebudeme. Místo funkce `super()` musíme v metodě `__init__()` uvést bázovou třídu, kterou inicializujeme, protože funkce `super()` nedokáže uhádnout naše záměry. Při inicializaci třídy `LoadSave` předáváme název souboru, který se má použít, a také názvy atributů, které chceme uložit. V tomto případě jde jen o jeden soukromý atribut `__stack`. (Seznam ukládat nechceme, což bychom ani v tomto případě nemohli, protože se jedná o seznam metod, který by tudíž nebylo možné z naloženého objektu vytáhnout.)

Třída `FileStack` má všechny metody třídy Undo a také metody `save()` a `load()` třídy `LoadSave`. Metodu `save()` jsme reimplementovat nemuseli, protože funguje správně, avšak v případě metody `load()` musíme po načtení vyčistit zásobník s anulačními metodami. To je nezbytné, protože můžeme provést uložení, pak několik změn a nato data načíst. Při načtení se zahodí předchozí data, takže jakékoli anulování již nemá smysl. Původní třída `Undo` neměla metodu `clear()`, takže jsme ji museli přidat:

```
    def clear(self): # ve třídě Undo
        self.__undos = []
```

V metodě `Stack.load()` jsme pro zavolání metody `LoadSave.load()` použili funkci `super()`, protože zde žádnou metodu `Undo.load()`, která by mohla způsobit nejednoznačnost, nemáme. Pokud by obě bázové třídy měly metodu `load()`, pak by se zavolala jedna z nich podle pořadí vyhodnocování metod v jazyku Python. Proto používáme funkci `super()` jen tehdy, když nehrozí žádná nejednoznačnost, přičemž v ostatních případech používáme název příslušné bázové třídy, takže se nikdy nespoléháme na pořadí vyhodnocování metod. Při volání metody `self.clear()` opět nehrozí žádná nejednoznačnost, protože metodu `clear()` má pouze třída `Undo`. Funkci `super()` ale použít nemusíme, protože (na rozdíl od metody `load()`) třída `FileStack` metodu `clear()` nemá.

Co by se stalo, kdybychom později přidali metodu `clear()` do třídy `FileStack`? Rozbilo by to metodu `load()`. Řešením by bylo zavolat uvnitř metody `load()` místo holé metody `self.clear()` metodu `super().clear()`. To by vedlo k tomu, že by se zavolala metoda `clear()` první nalezené nadtřídy. K ochraně proti těmto problémům bychom mohli zavést zásadu spočívající v používání explicitních názvů bázových tříd při použití vícenásobné dědičnosti (v tomto příkladu bychom tedy volali `Undo.clear(self)`). Nebo bychom se mohli zcela vyhnout vícenásobné dědičnosti a použít agregaci, kupříkladu odvozením od třídy `Undo` a vytvořením třídy `LoadSave` navržené pro agregaci.

Vícenásobná dědičnost nám zde dala směsici dvou spíše odlišných tříd s tím, že nemusíme sami implementovat žádné operace pro anulování nebo načítání a ukládání, přičemž se můžeme spolehnout na funkčnost poskytovanou bázovými třídami. To může být velice pohodlné a funguje to zvláště dobře v situaci, kdy se rozhraní API děděných třídy nijak nepřekrývají.

Metatřídy

Metatřída se má k třídě jako třída k instanci. To znamená, že metatřída se používá k vytváření tříd, stejně jako se třídy používají ke tvorbě instancí. A stejně jako se můžeme pomocí funkce `isinstance()`

zeptat, zda nějaká instance náleží určité třídě, můžeme se také pomocí funkce `issubclass()` zeptat, zda je nějaký objekt představující třídu (např. `dict`, `int` nebo `SortedList`) odvozen od jiné třídy.

Nejjednodušší použití metatříd spočívá v přizpůsobení vlastních tříd tak, aby zapadly do hierarchie abstraktních bázových tříd Pythonu. Například k tomu, aby byla třída `SortedList` zároveň typem `collections.Sequence`, můžeme třídu `SortedList` místo odvození od abstraktní bázové třídy (což jsme si ukázali dříve) zaregistrovat jako typ `collections.Sequence`:

```
class SortedList:
...
collections.Sequence.register(SortedList)
```

Třídu nejdříve definujeme běžným způsobem a poté ji zaregistrujeme u abstraktní bázové třídy `collections.Sequence`. Touto registrací se třída stává *virtuální podtřídou*.* Virtuální podtřída hlásí, že je podtřídou třídy nebo tříd, u nichž je zaregistrovaná (např. při zavolání funkce `isinstance()` nebo `issubclass()`), z těchto tříd však nedědí žádná data ani metody.

Tato registrace třídy poskytuje určitý příslib, že daná třída nabízí rozhraní API tříd, u nichž je registrována, neposkytuje však žádnou záruku, že svůj příslib bude ctít. Jedním z použití metatříd je poskytování příslibu i záruky ohledně rozhraní API určité třídy. Dalším použitím je modifikace třídy (podobně jako u dekorátorů tříd). Metatřídy lze samozřejmě použít také pro oba účely současně.

Představte si, že chcete vytvořit skupinu tříd, které poskytují metody `load()` a `save()`. Můžeme tedy vytvořit třídu, která při použití ve formě metatřídy zkontroluje, zda jsou tyto metody přítomny:

```
class LoadableSaveable(type):

    def __init__(cls, classname, bases, dictionary):
        super().__init__(classname, bases, dictionary)
        assert hasattr(cls, "load") and \
               isinstance(getattr(cls, "load"),
                          collections.Callable), ("třída '" +
               classname + "' musí poskytovat metodu load()")
        assert hasattr(cls, "save") and \
               isinstance(getattr(cls, "save"),
                          collections.Callable), ("třída '" +
               classname + "' musí poskytovat metodu save()")
```

Třídy, které mají sloužit jako metatřídy, musejí být odvozeny od nejzákladnější bázové třídy pro metatřídy s názvem `type` nebo od některé z jejích podtříd.

Všimněte si, že tato třída se volá v okamžiku, kdy dochází k tvorbě instancí *tříd*, které ji používají, což nejspíše nebývá příliš často, takže dopad na výkon za běhu programu je extrémně malý. Všimněte si také, že kontroly musíme provést až po vytvoření třídy (volání `super()`), protože jedině tak budou atributy třídy k dispozici v samotné třídě. (Atributy jsou ve slovníku, my ale raději při provádění kontrol pracujeme s aktuálně inicializovanou třídou.)

* V terminologii jazyka Python neznamená pojem *virtuální* to stejné jako v terminologii jazyka C++.

Mohli bychom pomocí funkce `hasattr()` zkontrolovat, zda jsou atributy `load` a `save` volatelné, abychom ověřili, mají-li atribut `__call__`, my ale raději zkontrolujeme, zda se jedná o instance třídy `collections.Callable`. Abstraktní bázová třída `collections.Callable` poskytuje příslib (ale ne záruku), že instance jejích podtříd (nebo virtuálních podtříd) jsou volatelné.

Abstraktní bázové třídy modulu `collections` ➤ 369

Jakmile je třída vytvořená (s použitím metody `type.__new__()` nebo reimplementací metody `__new__()`), inicializujeme metatřídu zavoláním její metody `__init__()`. Metodě `__init__()` předáváme argument `cls`, což je právě vytvořená třída, `classname`, což je název této třídy (dostupný také přes `cls.__name__`), `bases`, což je seznam jejích bázových tříd (bez třídy `object`, takže může být prázdný), a `dictionary`, který uchovává atributy, jež se při vytvoření třídy `cls` staly atributy třídy, tedy v případě, že jsme nezasáhli v reimplementaci metody `__new__()` metatřídy.

Zde je několik interaktivních příkladů, které ukazují, co se stane, když vytváříme třídy s použitím metatřídy `LoadableSaveable`:

```
>>> class Bad(metaclass=Meta.LoadableSaveable):
... def some_method(self): pass
Traceback (most recent call last):
...
AssertionError: třída 'Bad' musí poskytovat metodu load()
Tato metatřída stanoví, že třídy, které ji používají, musejí poskytovat určité
metody, a pokud tak jako v tomto případě nečiní, vyvolá se výjimka AssertionError.
>>> class Good(metaclass=Meta.LoadableSaveable):
... def load(self): pass
... def save(self): pass
>>> g = Good()
```

Třída `Good` ctí požadavky metatřídy na rozhraní API, i když nesplňuje naše neformální očekávání ohledně jejího chování.

Metatřídy můžeme použít také ke změně tříd, které je používají. Pokud se tato změna týká názvu, bázových tříd nebo slovníku vytvářené třídy (např. jejích slotů), pak musíme reimplementovat metodu `__new__()` metatřídy. V případě ostatních změn, jako je přidání metody či datového atributu, stačí reimplementovat jen metodu `__init__()`, ačkoliv operace v ní prováděné můžeme stejně dobře provést i metodě `__new__()`. Nyní se podíváme na metatřídu, která modifikuje třídy, které ji používají, pouze prostřednictvím své metody `__new__()`.

Jako alternativu k dekorátorům `@property` a `@name.setter` bychom mohli vytvořit třídy, které by pro identifikaci vlastnosti používaly jednoduchou konvenci pro pojmenovávání metod. Pokud má například nějaká třída metody ve tvaru `get_name()` a `set_name()`, pak můžeme očekávat, že tato třída má soukromou vlastnost `__name`, kterou můžeme číst i zapisovat pomocí výrazu *`instance`*`.name`. To vše lze realizovat pomocí metatřídy. Zde je příklad třídy, která používá tuto konvenci pro pojmenovávání metod:

```
class Product(metaclass=AutoSlotProperties):

    def __init__(self, barcode, description):
        self.__barcode = barcode
```

```
        self.description = description

    def get_barcode(self):
        return self.__barcode

    def get_description(self):
        return self.__description

    def set_description(self, description):
        if description is None or len(description) < 3:
            self.__description = "<Neplatný popis>"
        else:
            self.__description = description
```

V inicializační metodě musíme provést přiřazení do soukromé vlastnosti `__barcode`, protože nemáme žádnou metodu pro její čtení. Dalším důsledkem tohoto přístupu je, že vlastnost `barcode` je pouze pro čtení. Na druhou stranu vlastnost `description` lze číst i zapisovat. Zde je několik ukázek interaktivního použití této třídy:

```
>>> product = Product("101110110", "8mm Svorkovač")
>>> product.barcode, product.description
('101110110', '8mm Svorkovač')
>>> product.description = "8mm Svorkovač (dlouhý)"
>>> product.barcode, product.description
('101110110', '8mm Svorkovač (dlouhý)')
```

Pokud bychom se pokusili přiřadit hodnotu do vlastnosti `barcode`, došlo by k vyvolání výjimky `AttributeError` s textem „can't set attribute" (nemohu nastavit atribut).

Podíváme-li se na atributy třídy `Product` (např. pomocí funkce `dir()`), obdržíme pouze `barcode` a `description`. Metody `get_name()` a `set_name()` zde již nejsou, protože byly nahrazeny vlastností `name`. Dále proměnné uchovávající čárový kód a popis jsou soukromé (`__barcode` a `__description`) a byly přidány do slotů pro minimalizaci nároků na paměť této třídy. To vše má na svědomí metatřída `AutoSlotProperties`, která je implementována v jediné metodě:

```
class AutoSlotProperties(type):

    def __new__(mcl, classname, bases, dictionary):
        slots = list(dictionary.get("__slots__", []))
        for getter_name in [key for key in dictionary
                            if key.startswith("get_")]:
            if isinstance(dictionary[getter_name],
                          collections.Callable):
                name = getter_name[4:]
                slots.append("__" + name)
                getter = dictionary.pop(getter_name)
                setter_name = "set_" + name
```

```
            setter = dictionary.get(setter_name, None)
            if (setter is not None and
                isinstance(setter, collections.Callable)):
                del dictionary[setter_name]
            dictionary[name] = property(getter, setter)
        dictionary["__slots__"] = tuple(slots)
        return super().__new__(mcl, classname, bases, dictionary)
```

Metoda `__new__()` metatřídy obdrží jako parametry metatřídu, název třídy, bázové třídy a slovník třídy, která se má vytvořit. Místo metody `__init__()` musíme reimplementovat metodu `__new__()`, protože chceme změnit tento slovník ještě před vytvořením třídy.

Začínáme zkopírováním kolekce `__slots__` (vytvoříme prázdnou, není-li k dispozici) a zajištěním, abychom místo n-tice měli seznam, protože jej potřebujeme upravovat. Z každého atributu ve slovníku vybereme takové, které začínají na `"get_"` a jsou volatelné, což znamená všechny, které představují metody pro čtení vlastnosti. U každého takto nalezeného atributu přidáme do slotů soukromý název pro uložení odpovídajících dat. Máme-li například metodu `get_name()` pro čtení vlastnosti, pak do slotů přidáme `__name`. Potom vezmeme odkaz na metodu pro čtení vlastnosti a vymažeme ji ze slovníku pod jejím původním názvem (což provedeme najednou pomocí metody `dict.pop()`). Totéž provedeme i pro metody pro zápis vlastnosti, jsou-li k dispozici, a poté vytvoříme nový prvek slovníku s požadovaným názvem vlastnosti jako jeho klíčem. Máme-li například metodu `get_name()` pro čtení vlastnosti, pak se vlastnost bude jmenovat `name`. Hodnotu prvku nastavíme jako vlastnost s metodami pro její čtení a zápis (které mohou mít hodnotu `None`), které jsme našli a odstranili ze slovníku.

Na konci nahradíme původní sloty upraveným seznamem slotů, který obsahuje soukromý slot pro každou přidanou vlastnost, a zavoláme bázovou třídu, která vytvoří požadovanou třídu, ovšem s použitím našeho upraveného slovníku. Všimněte si, že v tomto případě musíme v souvislosti s použitím funkce `super()` explicitně předat metatřídu, což je nezbytné při každém volání metody `__new__()`, protože se nejedná o metodu instance, ale třídy.

U tohoto příkladu jsme metodu `__init__()` psát nemuseli, protože veškerou činnost provádíme v metodě `__new__()`. Nebyl by ale žádný problém reimplementovat obě metody `__new__()` a `__init__()` a rozdělit činnost mezi ně.

Pokud s ohledem na sílu a všestrannost přirovnáme agregaci a dědičnost k ručním vrtákům a dekorátory a deskriptory k elektrickým vrtačkám, pak můžeme metatřídy přirovnat k laserovým paprskům. Metatřídy jsou posledním nástrojem, po kterém bychom měli sáhnout, což neplatí snad jen pro vývojáře aplikačních rámců, kteří potřebují poskytovat svým uživatelům výkonné prostředky, pro jejichž pochopení nemusejí absolvovat žádná úmorná školení.

Funkcionální styl programování

Programování ve funkcionálním stylu je přístup k programování, u něhož se výpočty sestavují skládáním funkcí, které nemodifikují své argumenty, nepracují ani nemění stav programu a poskytují své výsledky jako návratové hodnoty. Silnou stránkou tohoto druhu programování je to, že je (teoreticky) snadnější vyvinout funkce v izolaci a odladit funkcionální program. Tomu napomáhá skuteč-

nost, že funkcionální programy nemají změny stavu, a proto lze o jejich funkcích uvažovat z matematického hlediska.

S funkcionálním programováním jsou spojeny tři principy: *mapování*, *filtrování* a *redukování*. Při mapování se vezme funkce a iterovatelný objekt a vytvoří se nový iterovatelný objekt (nebo seznam), v němž je každý prvek výsledkem volání funkce na odpovídající prvek v původním iterovatelném objektu. Tento princip podporuje vestavěná funkce `map()`:

```
list(map(lambda x: x ** 2, [1, 2, 3, 4]))     # vrátí: [1, 4, 9, 16]
```

Funkce `map()` přijímá jako své argumenty funkci a iterovatelný objekt a kvůli efektivitě vrací místo seznamu iterátor. Zde jsme si vynutili vytvoření seznamu, aby byl výsledek srozumitelnější.

```
[x ** 2 for x in [1, 2, 3, 4]]     # vrátí: [1, 4, 9, 16]
```

Místo funkce `map()` lze často použít generátor. Zde jsme sáhli po seznamové komprehenzi, abychom nemuseli používat funkci `list()`. Pro vytvoření generátoru pak už jen stačí změnit vnější hranaté závorky na kulaté.

Při filtrování se vezme funkce a iterovatelný objekt a vytvoří se nový iterovatelný objekt, v němž jsou prvky z původního iterovatelného objektu, pro které vrátila funkce aplikovaná na tento prvek hodnotu `True`. Tento princip podporuje vestavěná funkce `filter()`:

```
list(filter(lambda x: x > 0, [1, -2, 3, -4]))     # vrátí: [1, 3]
```

Funkce `filter()` přijímá jako své argumenty funkci a iterovatelný objekt a vrací iterátor.

```
[x for x in [1, -2, 3, -4] if x > 0]     # vrátí: [1, 3]
```

Funkci `filter()` můžeme vždy nahradit generátorovým výrazem nebo seznamovou komprehenzí.

Při redukování se vezme funkce a iterovatelný objekt a vytvoří se jediná výsledná hodnota. Funguje to tak, že funkce se zavolá na první dvě hodnoty iterovatelného objektu, pak na vypočtený výsledek a třetí hodnotu, pak na vypočtený výsledek a čtvrtou hodnotu – a tak pořád dál, dokud se nepoužijí všechny hodnoty. Tento princip podporuje funkce `functools.reduce()` z modulu `functools`. Zde jsou dva řádky kódu, které provádějí stejný výpočet:

```
functools.reduce(lambda x, y: x * y, [1, 2, 3, 4])  # vrátí: 24
functools.reduce(operator.mul, [1, 2, 3, 4])        # vrátí: 24
```

Modul `operator` obsahuje funkce pro všechny operátory jazyka Python k usnadnění programování ve funkcionálním stylu. Zde jsme na druhém řádku použili funkci `operator.mul()`, takže jsme oproti prvnímu řádku nemuseli vytvářet multiplikační funkci pomocí příkazu `lambda`.

Python dále nabízí několik vestavěných funkcí pro redukování. Funkce `all()` vrátí po zadání iterovatelného objektu hodnotu `True`, vrátí-li všechny jeho prvky po aplikaci funkce `bool()` hodnotu `True`. Funkce `any()` vrátí hodnotu `True`, vrátí-li kterýkoliv prvek iterovatelného objektu po aplikaci funkce `bool()` hodnotu `True`. Funkce `max()` vrátí největší prvek v iterovatelném objektu. Funkce `min()` vrátí nejmenší prvek v iterovatelném objektu. A Funkce `sum()` vrátí součet všech prvků iterovatelného objektu.

Klíčové principy jsme probrali, a proto se nyní můžeme podívat na pár dalších příkladů. Začneme několika způsoby pro získání celkové velikost všech souborů v seznamu `files`:

```
functools.reduce(operator.add, (os.path.getsize(x) for x in files))
functools.reduce(operator.add, map(os.path.getsize, files))
```

Použití funkce `map()` je často kratší než ekvivalentní seznamová komprehenze nebo generátorový výraz, pokud ovšem nepracujeme s podmínkou. Jako sčítací funkci jsme místo `lambda x, y: x + y` použili funkci `operator.add()`.

Pokud bychom chtěli jen spočítat velikosti souborů Pythonu, můžeme odfiltrovat soubory bez přípony `.py`. To můžeme provést třemi způsoby:

```
functools.reduce(operator.add, map(os.path.getsize,
                 filter(lambda x: x.endswith(".py"), files)))
functools.reduce(operator.add, map(os.path.getsize,
                 (x for x in files if x.endswith(".py"))))
functools.reduce(operator.add, (os.path.getsize(x)
                 for x in files if x.endswith(".py")))
```

Druhá a třetí verze jsou pravděpodobně lepší, protože nevyžadují vytváření lambda funkce, nicméně volba mezi generátorovým výrazem (nebo seznamovou komprehenzí) a funkcemi `map()` a `filter()` je povětšinou jen otázkou osobního stylu programování.

Funkce `map()`, `filter()` a `functools.reduce()` lze často použít k eliminování cyklů, jak jsme si ukázali na výše uvedených příkladech. Tyto funkce jsou užitečné při převodu kódu zapsaného ve funkcionálním jazyku, avšak v jazyku Python můžeme funkci map() obvykle nahradit seznamovou komprehenzí a funkci `filter()` seznamovou komprehenzí s podmínkou, přičemž funkci `functools.reduce()` lze častokrát nahradit některou z vestavěných funkcionálních funkcí Pythonu, jako je `all()`, `any()`, `max()`, `min()` nebo `sum()`. Například:

```
sum(os.path.getsize(x) for x in files if x.endswith(".py"))
```

Tímto způsobem docílíme stejného výsledku jako v předchozích třech příkladech, avšak s mnohem kompaktnějším kódem.

Modul `operator` nabízí kromě funkcí zastupujících operátory jazyka Python také funkce `operator.attrgetter()` a `operator.itemgetter()`. S první jsme se již stručně seznámili v dřívější části této lekce. Obě vracejí funkce, které lze poté zavolat pro extrahování zadaných atributů či prvků.

operator.attrgetter() ➤ 357

Zatímco řezání lze použít k extrakci posloupnosti z části seznamu a řezání s krokováním k extrakci posloupnosti určitých částí (např. `L[::3]` pro extrakci každého třetího prvku), funkci `operator.itemgetter()` můžeme použít pro extrahování posloupnosti libovolných částí, například `operator.itemgetter(4, 5, 6, 11, 18)(L)`. Funkci získanou při volání funkce `operator.itemgetter()` není nutné ihned zavolat a zahodit, jak jsme to provedli zde, ale můžeme si ji uložit a předat jako argument třeba funkci `map()`, `filter()` nebo `functools.reduce()` nebo ji můžeme použít v slovníkové, seznamové nebo množinové komprehenzi.

Při řazení prvků můžeme stanovit klíčovou funkci. Touto funkcí může být libovolná funkce, například lambda funkce, vestavěná funkce či metoda (např. `str.lower()`) nebo funkce vrácená funkcí `operator.attrgetter()`. Řekněme, že seznam *L* uchovává objekty s atributem `priority`. Tento seznam pak můžeme seřadit podle priority takto: `L.sort(key=operator.attrgetter("priority"))`.

Kromě již zmíněných modulů `functools` a `operator` může být pro programování ve funkcionálním stylu užitečný také modul `itertools`. Dva či více seznamů můžeme sice procházet tak, že je spojíme, další možnost však spočívá v použití funkce `itertools.chain()`:

```
for value in itertools.chain(data_list1, data_list2, data_list3):
    total += value
```

Funkce `itertools.chain()` vrací iterátor, který postupně vrací hodnoty z první zadané posloupnosti, pak z druhé posloupnosti a tak dál, dokud se nepoužijí všechny hodnoty ze všech posloupností. Modul `itertools` nabízí spoustu dalších funkcí. V jeho dokumentaci najdete řadu malých, přesto však užitečných příkladů, a proto rozhodně stojí za přečtení. (Je třeba poznamenat, že s příchodem Pythonu 3.1 bylo do modulu `itertools` přidáno několik nových funkcí.)

Částečná aplikace funkce

Částečná aplikace funkce znamená vytvoření funkce ze stávající funkce a určitých argumentů, čímž vznikne nová funkce, která dělá přesně to, co funkce původní, má však některé argumenty fixované, takže volající je nemusejí předávat. Zde je velice jednoduchý příklad:

```
enumerate1 = functools.partial(enumerate, start=1)
for lino, line in enumerate1(lines):
    process_line(i, line)
```

Na prvním řádku vytváříme novou funkci `enumerate1()`, která zabaluje zadanou funkci (`enumerate()`) a klíčovaný argument (`start=1`), takže se při zavolání funkce `enumerate1()` zavolá původní funkce s fixovaným argumentem a s libovolnými dalšími argumenty, které jsou zadány v okamžiku jejího zavolání (v tomto případě `lines`). Zde nám bude funkce `enumerate1()` poskytovat standardní počítání řádků začínající od řádku 1.

Díky částečné aplikaci funkce můžeme svůj kód zjednodušit, což platí zejména v případech, kdy chceme opětovně volat stejnou funkci se stejnými argumenty. Například místo toho, abychom při každém volání funkce `open()` pro zpracování textových souborů v kódování UTF-8 opakovaně uváděli argumenty pro režim a kódování, můžeme vytvořit několik funkcí, jež budou mít tyto argumenty fixované:

```
reader = functools.partial(open, mode="rt", encoding="utf8")
writer = functools.partial(open, mode="wt", encoding="utf8")
```

Nyní můžeme otevírat textové soubory pro čtení voláním `reader(`*`název_souboru`*`)` a pro zápis voláním `writer(`*`název_souboru`*`)`.

Částečná aplikace funkce se velice často používá při programování rozhraní GUI (Graphical User Interface – grafické uživatelské rozhraní, viz Lekce 15), kde se obvykle používá jedna funkce, která se volá pro kterékoliv ze skupiny tlačítek. Například:

```
loadButton = tkinter.Button(frame, text="Načíst",
                            command=functools.partial(doAction, "load"))
saveButton = tkinter.Button(frame, text="Uložit",
                            command=functools.partial(doAction, "save"))
```

V tomto příkladu používáme knihovnu GUI s názvem `tkinter`, která je standardní součástí Pythonu. Třída `tkinter.Button` se používá pro tlačítka – zde jsme vytvořili dvě, obě uvnitř stejného rámu a každé s textem označujícím jeho účel. Argument `command` každého tlačítka je nastaven na funkci (v tomto případě na funkci `doAction()`), kterou musí knihovna `tkinter` zavolat v okamžiku stisknutí daného tlačítka. Použili jsme částečnou aplikaci funkce pro zajištění, aby prvním argumentem předávaným funkci `doAction()` byl řetězec, který označuje, které tlačítko ji zavolalo, aby tak byla schopna rozhodnout, jakou akci má provést.

Korutiny

Korutiny jsou funkce, jejichž zpracování lze na určitých místech pozastavit a zase obnovit. Korutina se tedy obvykle provádí do určitého příkazu, kde se při čekání na nějaká data její provádění pozastaví. V tomto okamžiku mohou pokračovat ve svém provádění ostatní části programu (většinou další korutiny, které nejsou pozastaveny). Jakmile jsou požadovaná data k dispozici, pokračuje korutina od místa, kde byla pozastavena, provádí zpracování (patrně na základě získaných dat) a své výsledky třeba posílá jiné korutině. Říkáme, že korutiny mají více vstupních a výstupních bodů, poněvadž mohou obsahovat více než jedno místo, kde lze jejich zpracování pozastavit a obnovit.

Korutiny jsou užitečné tehdy, když potřebujeme aplikovat více funkcí na stejné části dat nebo když chceme vytvářet kolony pro zpracování dat nebo když chceme mít hlavní funkci a pomocné funkce. Korutiny lze použít také jako jednodušší a méně zatěžující alternativu pro práci s vlákny. V seznamu balíčků pro jazyk Python (viz *pypi.python.org/pypi*) je k dispozici několik balíčků na bázi korutin, které poskytují odlehčenou práci s vlákny.

V jazyku Python je korutina funkce, která přijímá vstup z výrazu `yield`. Své výsledky může též posílat do přijímající funkce (která sama musí být korutinou). Kdykoliv korutina narazí na výraz `yield`, pozastaví své provádění a čeká na data. Jakmile požadovaná data obdrží, obnoví od tohoto místa své provádění. Korutina může mít více než jeden výraz `yield`, třebaže ve všech příkladech korutin, které si zde ukážeme, bude přítomen pouze jeden.

příkaz `yield` ➤ 332

Provádění nezávislých akcí na datech

Chceme-li provést skupinu nezávislých operací na stejných datech, pak se nám nabízí tradiční přístup, který spočívá v postupné v aplikaci jednotlivých operací za sebou. Jeho nevýhodou však je, že pokud je některá z operací pomalá, pak celý program musí před přechodem na další čekat na její dokončení. Řešením je použít korutiny. Každou operaci můžeme implementovat jako korutinu a pak je rozjet všechny naráz. Je-li jedna z nich pomalá, nebude mít vliv na ostatní (tedy alespoň ne do doby, dokud jim nedojdou zpracovávaná data), protože všechny fungují nezávisle.

Tabulka 8.5 zachycuje použití korutin pro souběžné zpracování. V tabulce jsou tři korutiny (každá provádějící odlišnou činnost), které zpracovávají stejné dva datové prvky a které potřebují na svou činnost odlišný čas. Korutina `coroutine1()` pracuje docela rychle, korutina `coroutine2()` pracuje pomaleji a u korutiny `coroutine3()` se čas potřebný na zpracování mění. Jakmile všechny tři korutiny

obdrží svá počáteční data na zpracování, pak některá může klidně čekat (protože skončí jako první), přičemž ostatní pokračují v práci, čímž se minimalizuje doba nečinnosti procesoru. Po dokončení práce s korutinami zavoláme na každé z nich metodu `close()`, a tím se zastaví jejich čekání na další data – to znamená, že nebudou spotřebovávat žádný další procesorový čas.

Tabulka 8.5: Odeslání dvou prvků dat do tří korutin

Krok	Akce	coroutine1()	coroutine2()	coroutine3()
1	Vytvoření korutin	Čeká	Čeká	Čeká
2	coroutine1.send(„a“)	Zpracovává „a“	Čeká	Čeká
3	coroutine2.send(„a“)	Zpracovává „a“	Zpracovává „a“	Čeká
4	coroutine3.send(„a“)	Čeká	Zpracovává „a“	Zpracovává „a“
5	coroutine1.send(„b“)	Zpracovává „b“	Zpracovává „a“	Zpracovává „a“
6	coroutine2.send(„b“)	Zpracovává „b“	Zpracovává „a“ („b“ nachystané)	Zpracovává „a“
7	coroutine3.send(„b“)	Čeká	Zpracovává „a“ („b“ nachystané)	Zpracovává „b“
8		Čeká	Zpracovává „b“	Zpracovává „b“
9		Čeká	Zpracovává „b“	Čeká
10		Čeká	Zpracovává „b“	Čeká
11		Čeká	Čeká	Čeká
12	coroutineN.close()	Hotovo	Hotovo	Hotovo

K vytvoření korutiny v jazyku Python stačí jednoduše vytvořit funkci, která má alespoň jeden výraz `yield` (umístěný obvykle uvnitř nekonečného cyklu). Když program dorazí k výrazu `yield`, provádění korutiny se pozastaví a korutina čeká na data. Jakmile data obdrží, její provádění se opět obnoví (od výrazu `yield`) a v okamžiku, kdy dokončí zpracování, vrátí se v cyklu zpět na výraz `yield`, aby počkala na další data. Zatímco je jedna či více korutin pozastavených a čekají na data, další se mohou provádět. Tím lze ve srovnání s obyčejným prováděním funkcí jednu po druhé docílit vyšší propustnosti, a tím také výkonu.

Ukážeme si, jak provádění nezávislých operací funguje v praxi při aplikování několika regulárních výrazů na text ve skupině souborů HTML. Cílem je vypsat adresy URL každého souboru a záhlaví na první a druhé úrovni. Začneme pohledem na regulární výrazy, potom vytvoříme korutinu `matchers` a poté se podíváme na samotné korutiny a na způsob jejich použití.

```
URL_RE = re.compile(r"""href=(?P<quote>['"])(?P<url>[^\1]+?)"""
                    r"""(?P=quote)""", re.IGNORECASE)
flags = re.MULTILINE|re.IGNORECASE|re.DOTALL
H1_RE = re.compile(r"<h1>(?P<h1>.+?)</h1>", flags)
H2_RE = re.compile(r"<h2>(?P<h2>.+?)</h2>", flags)
```

Tyto regulární výrazy odpovídají adrese URL v atributu `href` jazyka HTML a textu obsaženému ve značkách záhlaví `<h1>` a `<h2>`. (Regulární výrazy budeme probírat v lekci 13. Jejich znalost není pro pochopení tohoto příkladu podstatná.)

```
receiver = reporter()
matchers = (regex_matcher(receiver, URL_RE),
            regex_matcher(receiver, H1_RE),
            regex_matcher(receiver, H2_RE))
```

Korutiny mají vždy výraz `yield`, a proto se jedná o generátory. Přestože zde tedy vytváříme n-tici korutin pro hledání shody s regulárním výrazem, v podstatě vytváříme n-tici generátorů. Každá funkce `regex_matcher()` je korutina, která přijímá funkci (také korutinu) a regulární výraz, pro který má hledat shody. Kdykoli pak tato korutina nalezne shodu, odešle ji příjemci (`receiver`).

Generátory ➤ 332

```
@coroutine
def regex_matcher(receiver, regex):
    while True:
        text = (yield)
        for match in regex.finditer(text):
            receiver.send(match)
```

Korutina `regex_matcher()` na začátku vstupuje do nekonečného cyklu, okamžitě pozastavuje provádění a čeká, až jí výraz `yield` vrátí text, na který má aplikovat regulární výraz (`regex`). Jakmile obdrží text, začne procházet všechny nalezené shody a každou z nich posílá příjemci (`receiver`). Po dokončení hledání shod se korutina vrátí v cyklu zpět k výrazu `yield`, kde opět pozastaví své provádění a čeká na další text.

S (nedekorovanou) korutinou `regex_matcher()` je spojen jeden malý problém. Při její prvním vytvoření by měla zahájit provádění a postoupit tak k výrazu `yield`, aby byla připravena přijmout svůj první text. To bychom mohli provést tak, že bychom na každé vytvořené korutině zavolali před odesláním jakýchkoli dat vestavěnou funkci `next()`. Pohodlnější ale bude, když si vytvoříme dekorátor `@coroutine`, který to provede za nás.

```
def coroutine(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        generator = function(*args, **kwargs)
        next(generator)
    return generator
return wrapper
```

Dekorátor `@coroutine` přijímá funkci představující korutinu, na níž zavolá vestavěnou funkci `next()`, což způsobí, že se provede její kód až do prvního výrazu `yield`, takže bude připravena přijímat data.

Dekorátory ➤ 345

Korutinu `regex_matcher()` již známe, a proto se nyní podíváme na to, jak se používá, a pak si ukážeme korutinu `reporter()`, která přijímá její výstup.

```
try:
    for file in sys.argv[1:]:
        print(file)
        html = open(file, encoding="utf8").read()
        for matcher in matchers:
            matcher.send(html)
finally:
    for matcher in matchers:
        matcher.close()
    receiver.close()
```

Tento program čte názvy souborů uvedené na příkazovém řádku, u každého vypíše jeho název a poté načte celý text souboru do proměnné `html` s použitím kódování UTF-8. Program pak prochází všechny korutiny pro hledání shody s regulárním výrazem (v tomto případě tři) a každé z nich pošle načtený text. Každá korutina zpracuje text nezávisle na ostatních a každou shodu pošle korutině `reporter()`. Na konci zavoláme na této korutině a na každé korutině `matcher` metodu `close()`, která je ukončí. Jinak by totiž neustále čekaly (pozastaveny) na další text (nebo shody v případě korutiny `reporter()`), neboť obsahují nekonečné cykly.

```
@coroutine
def reporter():
    ignore = frozenset({"style.css", "favicon.png", "index.html"})
    while True:
        match = (yield)
        if match is not None:
            groups = match.groupdict()
            if "url" in groups and groups["url"] not in ignore:
                print(" URL:", groups["url"])
            elif "h1" in groups:
                print(" H1: ", groups["h1"])
            elif "h2" in groups:
                print(" H2: ", groups["h2"])
```

Korutinu `reporter()` používáme pro výpis výsledků. Vytvořili jsme ji výše příkazem `receiver = reporter()` a předali jako argument `receiver` každé korutině `regex_matcher()`. Korutina `reporter()` čeká (je pozastavena) v nekonečném cyklu, dokud jí nebude zaslána nějaká shoda, jejíž detaily vypíše, a poté opět čeká, přičemž se zastaví až v okamžiku, kdy na ni zavoláme metodu `close()`.

Tento způsob může použití korutin vést k lepšímu výkonu, vyžaduje však přijetí poněkud odlišného způsobu uvažování o zpracování.

Skládání kolon

Někdy je užitečné vytvořit *kolony* (pipelines) pro zpracování dat. Kolona je prosté složení jedné či více funkcí, u nichž datový prvek prošel první funkci, která jej buď zahodí (odfiltruje), nebo předá další funkci (netknutý nebo nějakým způsobem transformovaný). Druhá funkce přijme prvek od první a celý proces zopakuje, takže prvek zahodí nebo předá (přičemž jej třeba nějakým způsobem

transformuje) další funkci a tak pořád dál. Prvky, které doputují na konec, jsou pak nějakým způsobem odeslány na výstup.

Kolony mají obvykle několik komponent. Jednu pro získávání dat, jednu či více pro filtrování či transformaci dat a jednu pro výstup výsledků. Jedná se tedy o přístup k programování ve zcela funkcionálním stylu, který jsme probírali již v dříve v této části při pohledu na skládání některých z vestavěných funkcí Pythonu, jako je `filter()` a `map()`.

Skládání funkcí ➢ 382

Výhoda při použití kolon spočívá v tom, že můžeme číst datové prvky inkrementálně, často jen po jednom, přičemž koloně musíme předat pouze dostatek datových prvků pro její naplnění (obvykle jeden nebo jen několik prvků pro každou z komponent). To může vést k výrazné úspoře paměti ve srovnání řekněme s načtením do paměti celé skupiny dat, která se pak zpracují najednou.

Tabulka 8.6 znázorňuje jednoduchou kolonu se třemi komponentami. První komponenta kolony (`get_data()`) získává datový prvek, který se má následně zpracovat. Druhá komponenta (`process()`) tato data zpracuje (nepotřebné datové prvky může zahodit) – a samozřejmě bychom zde mohli mít libovolný počet dalších komponent pro zpracování a filtrování. Poslední komponenta (`reporter()`) výsledky vypisuje. Podle tabulky 8.6 se zpracují a vypíší prvky `"a"`, `"b"`, `"c"`, `"e"` a `"f"`, zatímco prvek `"d"` se zahodí.

Tabulka 8.6: Tříkroková kolona korutin zpracovávající šest prvků dat

Krok	Akce	get_data()	process()	reporter()
1	pipeline = get_data(process(reporter()))	Čeká	Čeká	Čeká
2	pipeline.send(„a“)	Čte „a“	Čeká	Čeká
3	pipeline.send(„b“)	Čte „b“	Zpracovává „a“	Čeká
4	pipeline.send(„c“)	Čte „c“	Zpracovává „b“	Vypisuje „a“
5	pipeline.send(„d“)	Čte „d“	Zpracovává „c“	Vypisuje „b“
6	pipeline.send(„e“)	Čte „e“	Zahazuje „d“	Vypisuje „c“
7	pipeline.send(„f“)	Čte „f“	Zpracovává „e“	Čeká
8		Čeká	Zpracovává `"f"`	Vypisuje `"e"`
9		Čeká	Čeká	Vypisuje `"f"`
10		Čeká	Čeká	Čeká
11	Uzavření korutin	Hotovo	Hotovo	Hotovo

Kolona zachycená v tabulce 8.6 představuje filtr, protože každý datový prvek, který jí prochází beze změny, je buď zahozen, nebo vypsán ve své původní podobě. Koncové komponenty kolon mají sklon provádět stejné role: získávat datové prvky a vypisovat výsledky. Ovšem mezi nimi můžeme mít tolik komponent, kolik potřebujeme, přičemž každá může provádět filtrování, transformování

nebo obojí. A v některých případech může složení komponent v jiném pořadí vytvořit kolonu, která provádí něco jiného.

Začneme pohledem na teoretický příklad, abychom získali lepší představu o tom, jak kolony na bázi korutin pracují, a poté se podíváme na jeden praktický příklad.

Předpokládejme, že máme posloupnost čísel s pohyblivou řádovou čárkou a chceme je zpracovat ve vícekomponentové koloně tak, aby se každé z nich transformovalo na celé číslo (zaokrouhlením) a přitom se zahodila jakákoli čísla, která jsou mimo rozsah (< 0 nebo >= 10). Máme-li čtyři komponenty ve formě korutin `acquire()` (vezme číslo), `to_int()` (transformuje jej zaokrouhlením a převodem na celé číslo), `check()` (předá číslo, které je v platném rozsahu, ostatní zahodí) a `output()` (vypíše číslo), pak můžeme vytvořit kolonu následujícím způsobem:

```
pipe = acquire(to_int(check(output())))
```

Do této kolony můžeme nyní posílat čísla voláním `pipe.send()`. Podíváme se podrobně na čísla 4,3 a 9,6 při jejich postupu kolonou a nepoužijeme při tom tabulku s jednotlivými kroky jako dříve, ale jinou vizualizační pomůcku:

```
pipe.send(4.3) -> acquire(4.3) -> to_int(4.3) -> check(4) -> output(4)
pipe.send(9.6) -> acquire(9.6) -> to_int(9.6) -> check(10)
```

Všimněte si, že u čísla 9,6 nemáme žádný výstup. To je dáno tím, že korutina `check()` přijala číslo 10, které je mimo rozsah (>= 10), a proto jej odfiltrovala.

Podívejme se na to, co by se stalo, kdybychom ze stejných komponent vytvořili odlišnou kolonu:

```
pipe = acquire(check(to_int(output())))
```

Tato kolona provede filtrování (`check()`) před transformováním (`to_int()`). Zde je její způsob práce s čísly 4,3 a 9,6:

```
pipe.send(4.3) -> acquire(4.3) -> check(4.3) -> to_int(4.3) -> output(4)
pipe.send(9.6) -> acquire(9.6) -> check(9.6) -> to_int(9.6) -> output(10)
```

Zde jsme nesprávně vypsali číslo 10, přestože se jedná o číslo mimo povolený rozsah. Důvodem je to, že jsme nejdříve aplikovat komponentu `check()`, která přijala hodnotu `9.6`, která je v platném rozsahu, a proto ji pustila dál. Avšak komponenta `to_int()` obdržená čísla zaokrouhluje.

Nyní prostudujeme konkrétní příklad, který čte všechny názvy souborů zadané na příkazovém řádku (včetně rekurzivního načítání těch, které jsou umístěné v adresářích zadaných na příkazovém řádku) a vypisuje absolutní cesty pro ty soubory, které splňují určitá kritéria.

Začneme pohledem na způsob skládání kolon a poté se podíváme na korutiny, které poskytují kolonám komponenty. Zde je nejjednodušší kolona:

```
pipeline = get_files(receiver)
```

Tato kolona vypíše každý zadaný soubor (nebo rekurzivně všechny soubory v zadaném adresáři). Funkce `get_files()` je korutina, která předává názvy souborů, a `receiver` je korutina `reporter()` (vytvořená příkazem `receiver = reporter()`), která jen vypíše každý obdržený název souboru. Tato kolona neprovádí nic moc navíc než funkci `os.walk()` (ve skutečnosti tuto funkci používá), můžeme však využít její komponenty pro skládání sofistikovanějších kolon.

os.walk() ➤ 219

```
pipeline = get_files(suffix_matcher(receiver, (".htm", ".html")))
```

Tuto kolonu jsme vytvořili složením korutiny `get_files()` společně s korutinou `suffix_matcher()`, díky níž se vypisují pouze soubory HTML.

Korutiny složené tímto způsobem se mohou rychle stát nečitelné, nic nám ale nebrání skládat kolonu postupně. Jednotlivé komponenty však musíme vytvářet od poslední po první.

```
pipeline = size_matcher(receiver, minimum=1024 ** 2)
pipeline = suffix_matcher(pipeline, (".png", ".jpg", ".jpeg"))
pipeline = get_files(pipeline)
```

Tato kolona vyhledá pouze soubory, které jsou nejméně jeden megabajt velké a mají příponu, která signalizuje, že se jedná o obrázek.

Jak se tyto kolony používají? Stačí je nakrmit názvy souborů či cestami a o zbytek už se postarají samy.

```
for arg in sys.argv[1:]:
    pipeline.send(arg)
```

Všimněte si, že nezáleží na tom, kterou kolonu používáme. Může jít o kolonu, která vypisuje všechny soubory, nebo o kolonu, který vypisuje soubory HTML, nebo o kolonu, která vypisuje obrázky – všechny fungují stejným způsobem. V tomto případě navíc všechny tři kolony představují filtry, protože všechny názvy souborů, které obdrží, buď předají beze změny další komponentě (a v případě korutiny `reporter()` vypíšou), nebo zahodí, protože nesplňují dané kritérium.

Ještě před tím, než se podíváme na funkci `get_files()` a další korutiny, si ukážeme triviální korutinu `reporter()` (předávanou jako `receiver`), která vypisuje výsledky.

```
@coroutine
def reporter():
    while True:
        filename = (yield)
        print(filename)
```

Dekorátor @coroutine ➤ 387

Použili jsme stejný dekorátor `@coroutine`, který jsme vytvořili v předchozím pododdílu.

Korutina `get_files()` v podstatě jen zabaluje funkci `os.walk()` a očekává zadání cest nebo názvů souborů, s nimiž má pracovat.

os.walk() ➤ 219

```
@coroutine
def get_files(receiver):
    while True:
```

```
        path = (yield)
        if os.path.isfile(path):
            receiver.send(os.path.abspath(path))
        else:
            for root, dirs, files in os.walk(path):
                for filename in files:
                    receiver.send(os.path.abspath(os.path.join(root, filename)))
```

Tato korutina má strukturu, která je nám již nyní známá: nekonečný cyklus, v němž čekáme, až výraz `yield` vrátí hodnotu, kterou můžeme zpracovat, a poté odešleme výsledek příjemci.

```
@coroutine
def suffix_matcher(receiver, suffixes):
    while True:
        filename = (yield)
        if filename.endswith(suffixes):
            receiver.send(filename)
```

Tato korutina vypadá jednoduše (a taková skutečně je), všimněte si ale, že odesílá pouze názvy souborů, které odpovídají zadaným příponám (`suffixes`), takže všechny ostatní se z kolony odfiltrují pryč.

```
@coroutine
def size_matcher(receiver, minimum=None, maximum=None):
    while True:
        filename = (yield)
        size = os.path.getsize(filename)
        if ((minimum is None or size >= minimum) and
            (maximum is None or size <= maximum)):
            receiver.send(filename)
```

Tato korutina je téměř identická jako korutina `suffix_matcher()`, tedy až na to, že místo souborů, které nemají odpovídající příponu, odfiltruje ty, jejichž velikost není v požadovaném rozmezí.

Kolona, kterou jsme vytvořili, trpí několika neduhy. Jedním z nich je, že žádnou z korutin nikdy neuzavíráme. V tomto případě to sice nevadí, protože po dokončení zpracování program končí, je ale lepší navyknout tomu, že by se korutiny měly po dokončení práce uzavřít. Další problém tkví v tom, že můžeme operační systém žádat (v pozadí všeho dění) o odlišné části informace o tomtéž souboru v několika částech kolony, což může být pomalé. Řešením je upravit korutinu `get_files()` tak, aby pro každý soubor místo názvu souboru vracela n-tice se dvěma prvky (*`název_souboru`*, *`os.stat()`*) a ty pak předala další komponentě v koloně.* To by znamenalo, že získáme všechny relevantní informace pro každý soubor jen jednou. Oba tyto problémy budete moci vyřešit a k tomu přidat další funkčnost ve cvičení na konci této lekce.

* Funkce `os.stat()` přijímá název souboru a vrací pojmenovanou n-tici s nejrůznějšími informacemi o zadaném souboru, včetně jeho velikosti, režimu a datu a čase jeho poslední modifikace.

Tvorba korutin pro použití v kolonách vyžaduje určitou změnu v uvažování. Za odměnu ale získáme úžasnou flexibilitu a u rozsáhlých souborů dat můžeme očekávat minimalizaci množství dat uchovávaných v paměti a rychlejší propustnost.

Příklad: Valid.py

Deskriptory ➤ 360

V této části zkombinujeme deskriptory s dekorátory tříd pro vytvoření výkonného mechanizmu určeného ke tvorbě validovaných atributů.

Dekorátory tříd ➤ 365

Až dosud jsme pro zajištění, aby byl nějaký atribut nastaven pouze na platnou hodnotu, spoléhali na vlastnosti (nebo jsme používali metody pro čtení a zápis vlastností). Nevýhodou tohoto přístupu je, že musíme pro každý validovaný atribut v každé třídě přidávat validační kód. Mnohem pohodlnější a snadněji udržovatelné by bylo, kdybychom mohli přidávat atributy do tříd s potřebnou nezbytnou validací. Zde je příklad syntaxe, kterou bychom rádi používali:

```
@valid_string("name", empty_allowed=False)
@valid_string("productid", empty_allowed=False,
              regex=re.compile(r"[A-Z]{3}\d{4}"))
@valid_string("category", empty_allowed=False, acceptable=
              frozenset(["Consumables", "Hardware", "Software", "Media"]))
@valid_number("price", minimum=0, maximum=1e6)
@valid_number("quantity", minimum=1, maximum=1000)
class StockItem:

    def __init__(self, name, productid, category, price, quantity):
        self.name = name
        self.productid = productid
        self.category = category
        self.price = price
        self.quantity = quantity
```

Regulární výrazy ➤ 470

Všechny atributy třídy `StockItem` jsou validované. Kupříkladu atribut `productid` lze nastavit pouze na neprázdný řetězec, který začíná třemi velkými písmeny a končí čtyřmi číslicemi, atribut `category` lze nastavit pouze na neprázdný řetězec, který je jednou z uvedených hodnot, a atribut `quantity` lze nastavit pouze na číslo mezi 1 a 1000 včetně. Když se pokusíme nastavit neplatnou hodnotu, vyvolá se výjimka.

Dekorátory tříd ➤ 365

Validaci realizujeme kombinováním dekorátorů tříd s deskriptory. Jak jsme si řekli již dříve, dekorátory tříd mohou přijímat pouze jediný argument – třídu, kterou dekorují. Zde jsme tedy použili techniku, kterou jsme si ukázali, když jsme poprvé probírali dekorátory. Díky této technice mohou funkce `valid_string()` a `valid_number()` přijímat jakékoli argumenty a vrací dekorátor, který následně přijímá třídu a vrací její modifikovanou verzi.

Podívejme se nyní na funkce `valid_string()`:

```
def valid_string(attr_name, empty_allowed=True, regex=None,
                 acceptable=None):
```

```
    def decorator(cls):
        name = "__" + attr_name
        def getter(self):
            return getattr(self, name)
        def setter(self, value):
            assert isinstance(value, str), (attr_name +
                                            " musí být řetězec")
            if not empty_allowed and not value:
                raise ValueError("{0} nesmí být prázdné".format(
                                 attr_name))
            if ((acceptable is not None and value not in acceptable) or
                (regex is not None and not regex.match(value))):
                raise ValueError("{attr_name} nelze nastavit na "
                                 "{value}".format(**locals()))
            setattr(self, name, value)
        setattr(cls, attr_name, GenericDescriptor(getter, setter))
        return cls
    return decorator
```

Funkce začíná vytvořením funkce představující dekorátor třídy, která přijímá třídu jako svůj jediný argument. Dekorátor přidává do dekorované třídy dva atributy: soukromý datový atribut a deskriptor. Když funkci `valid_string()` zavoláme například s názvem „productid“, obdrží třída `StockItem` atribut `__productid`, který uchovává hodnotu identifikačního kódu produktu, a deskriptorový atribut `productid`, který se používá pro přístup k této hodnotě. Pokud například vytvoříme nějakou položku příkazem `item = StockItem("TV", "TVA4312", "Electrical", 500, 1)`, pak identifikační kód produktu získáme pomocí `item.productid` a nastavíme jej třeba pomocí `item.productid = "TVB2100"`.

Funkce pro čtení vlastnosti vytvořená dekorátorem jednoduše vrací hodnotu soukromého datového atributu pomocí globální funkce `getattr()`. Funkce pro zápis vlastnosti obsahuje validaci a na konci používá pro nastavení soukromého datového atributu na novou (a platnou) hodnotu funkci `setattr()`. Soukromý datový atribut se ve skutečnosti vytvoří až při jeho prvním nastavení.

Jakmile máme funkce pro čtení a zápis vlastnosti vytvořené, použijeme opět funkci `setattr()`, tentokrát ale pro vytvoření nového atributu třídy se zadaným názvem (např. `productid`) a s hodnotou nastavenou na deskriptor typu `GenericDescriptor`. Na konci vrátí dekorátorová funkce upravenou třídu a funkce `valid_string()` vrátí dekorátorovou funkci.

Funkce `valid_number()` je z hlediska struktury stejná jako funkce `valid_string()` a liší se jen v argumentech, které přijímá, a ve validačním kódu funkce pro zápis vlastnosti, takže si ji zde ukazovat nebudeme. (Kompletní zdrojový kód je k dispozici v modulu `Valid.py`.)

Nyní nám zbývá už jen třída `GenericDescriptor`, která je ze všeho nejjednodušší:

```
class GenericDescriptor:

    def __init__(self, getter, setter):
        self.getter = getter
```

```
        self.setter = setter

    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return self.getter(instance)

    def __set__(self, instance, value):
        return self.setter(instance, value)
```

Tento deskriptor uchovává funkce pro čtení a zápis každého atributu a jednoduše předává práci těmto funkcím.

Shrnutí

V této lekci jsme se dozvěděli mnohem více o podpoře Pythonu pro procedurální a objektově orientované programování a okusili jsme též jeho podporu pro funkcionální styl programování.

V první části jsme se naučili, jak vytvářet generátorové výrazy, a probrali jsme do větší hloubky generátorové funkce. Dále jsme se naučili, jak dynamicky importovat moduly, jak přistupovat k funkčnosti z takovýchto modulů a jak dynamicky provádět kód. V této části jsme viděli příklady tvorby a použití rekurzivních funkcí a nelokálních proměnných. Dozvěděli jsme se také, jak vytvářet vlastní dekorátory funkcí a metod a jak psát a využívat anotace funkcí.

Ve druhé části této lekce jsme studovali nejrůznější pokročilé stránky objektově orientovaného programování. Nejdříve jsme se naučili více o přístupu k atributům, například s použitím speciální metody `__getattr__()`. Potom jsme se seznámili s funktory a ukázali jsme si, jak lze pomocí nichž poskytovat funkce se stavem, čehož můžeme docílit přidáním vlastností do funkce nebo použitím uzávěrů. Oběma těmto tématům jsme se v této lekci podrobně věnovali. Naučili jsme se, jak používat příkaz `with` se správci kontextu a jak vytvářet vlastní správce kontextu. Objekty souboru jsou v Pythonu správce kontextu, a proto budeme od nynějška pracovat se soubory pomocí struktur `try with ... except`, které i bez bloků `finally` zajistí, aby se otevřené soubory uzavřely.

Ve druhé části jsme se dále věnovali pokročilejším prvkům objektově orientovaného programování. Začali jsme deskriptory. Ty lze použít mnoha způsoby a představují technologii, která tvoří základ řady standardních dekorátorů Pythonu, mezi něž patří `@property` a `@classmethod`. Dozvěděli jsme se, jak vytvářet vlastní deskriptory, a viděli jsme tři velmi odlišné ukázky jejich použití. Dále jsme prostudovali dekorátory tříd a ukázali jsme si, jak lze upravit třídu vesměs podobným způsobem, jakým dekorátor funkce upravuje funkci.

V posledních třech oddílech druhé části jsme se seznámili s podporou jazyka Python pro abstraktní bázové třídy, vícenásobnou dědičnost a metatřídy. Naučili jsme se, jak u svých vlastních tříd zajistit, aby zapadly mezi standardní abstraktní bázové třídy Pythonu, a jak vytvářet své vlastní abstraktní bázové třídy. Dále jsme si ukázali, jak pomocí vícenásobné dědičnosti sjednotit prvky odlišných tříd do jediné třídy. A při výkladu o metatřídách jsme se naučili, jak zasáhnout do tvorby a inicializace třídy (na rozdíl od tvorby a inicializace instance třídy).

V předposlední části jsme se seznámili s funkcemi a moduly poskytovanými Pythonem na podporu funkcionálního stylu programování. Naučili jsme se, jak používat běžné funkcionální postupy označované jako mapování, filtrování a redukování. Dále jsme se naučili, jak vytvářet částečné funkce a jak vytvářet a používat korutiny. V poslední části jsme si ukázali, jak kombinací dekorátorů tříd a deskriptorů nabídnout výkonný a flexibilní mechanizmus pro tvorbu validovaných atributů.

V této lekci jsme dokončili výklad věnovaný samotnému jazyku Python. V této a předchozích lekcích jsme sice neprobrali naprosto všechny prvky jazyka Python, avšak ty, o nichž jsme se nezmínili, jsou bezvýznamné a používají se jen velmi zřídka. V žádné z následujících lekcí se již s novým prvkem jazyka neseznámíme, přestože každá z nich využívá moduly ze standardní knihovny, které jsme dosud neprobírali, a některé z nich dále rozvíjejí techniky, které jsme si ukázali v této a v předchozích lekcích. Kromě toho programy, které si ukážeme v následujících lekcích, nepodléhají žádným omezením, s nimiž jsme museli dříve počítat (tj. používat pouze ty aspekty jazyka, s nimiž jsme se dosud seznámili), a proto se jedná o příklady po všech stránkách maximálně využívající potenciálu jazyka Python.

Cvičení

Žádné z prvních tří zde popsaných cvičení nevyžaduje psaní většího množství kódu (o čtvrtém to již říci nelze) a žádné z nich není vůbec jednoduché!

1. Zkopírujte program `magic-numbers.py`, vymažte jeho funkce `get_function()` a všechny funkce `load_modules()` kromě jedné. Přidejte třídu `GetFunction` představující funktor, která obsahuje dvě mezipaměti, jednu pro uchovávání nalezených funkcí a druhou pro uchovávání funkcí, které nebylo možné nalézt (aby nebylo nutné opakovaně hledat nějakou funkci v modulu, který ji nemá). Jedinou úpravou ve funkci `main()` je přidání příkazu `get_function = GetFunction()` před cyklus a použití příkazu `with` kvůli eliminaci bloku `finally`. Dále zkontrolujte, že funkce modulu jsou volatelné, k čemuž nepoužívejte funkci `hasattr()`, ale `collections.Callable`. Tuto třídu lze napsat asi na dvaceti řádcích. Řešení najdete v souboru `magic-numbers_ans.py`.

2. Vytvořte nový soubor modulu a v něm definujte tři funkce: funkci `is_ascii()`, která vrací hodnotu `True`, mají-li všechny znaky v zadaném řetězci kódové body menší než 127, funkci `is_ascii_punctuation()`, která vrací hodnotu `True`, patří-li všechny znaky do řetězce `string.punctuation`, a funkci `is_ascii_printable()`, která vrací hodnotu `True`, patří-li všechny znaky do řetězce `string.printable`. Poslední dvě funkce jsou z hlediska struktury stejné. Každá funkce by měla být vytvořena pomocí příkazu `lambda` a její tělo by nemělo být delší než jeden nebo dva řádky kódu ve funkcionálním stylu. Nezapomeňte pro každou přidat dokumentační řetězec s dokumentačními testy a zajistit, aby modul tyto testy spustil. Všechny funkce nevyžadují více než tři až pět řádků, přičemž pro celý modul stačí méně než 25 řádků včetně dokumentačních testů. Řešení najdete v souboru `Ascii.py`.

3. Vytvořte nový soubor modulu a definujte v něm třídu `Atomic` představující správce kontextu. Tato třída by měla pracovat podobně, jako třída `AtomicList`, kterou jsme si ukázali v této lekci, ovšem s tím, že místo seznamů bude pracovat s libovolným měnitelným typem představujícím kolekci. Metoda `__init__()` by měla zkontrolovat vhodnost kontejneru a místo ukládání příznaku pro mělkou či hloubkovou kopii by měla v závislosti na tomto příznaku přiřadit atributu `self.copy` vhodnou funkci a metodě `__enter__()` zavolat kopírovací funkci. Metoda `__exit__()`

je trošku složitější, což je dáno tím, že nahrazování obsahu u množin a slovníků je jiné než u seznamů, přičemž nemůžeme použít přiřazení, protože to by nemělo vliv na původní kontejner. Samotnou třídu lze napsat asi na třiceti řádcích, měli byste ale přidat také dokumentační testy. Řešení najdete v souboru `Atomic.py`, který má kolem 150 řádků včetně dokumentačních testů.

4. Vytvořte program, který vyhledává soubory na základě zadaných kritérií (podobně jako unixový program `find`). Jeho použití by mělo mít tvar `find.py` *`volby soubory_či_cesty`*. Všechny volby jsou volitelné a bez nich se vypíšou všechny soubory uvedené na příkazovém řádku a všechny soubory v adresářích uvedených na příkazovém řádku. Volby by měly omezit vypisované soubory následujícím způsobem: celé číslo `-d` nebo `--days` *`celé_číslo`* zahodí všechny soubory starší než uvedený počet dní, `-b` nebo `--bigger` *`celé_číslo`* zahodí všechny soubory *menší* než uvedený počet bajtů, `-s` nebo `--smaller` *`celé_číslo`* zahodí všechny soubory *větší* než uvedený počet bajtů, `-o` nebo `--output` *`co`*, kde *`co`* je „date“, „size“ nebo „date, size“ (v kterémkoli pořadí) stanoví, co by se mělo vypsat (názvy souborů se vypisují vždy), `-u` nebo `--suffix` zahodí všechny soubory, které nemají odpovídající příponu. (Lze zadat i více přípon vzájemně oddělených čárkou.) Je-li za číslem za volbou `-s` či `--smaller` nebo `-b` či `--bigger` písmeno „k“, pak by toto číslo mělo být považováno za kilobajty a vynásobeno 1024. Podobně funguje také písmeno „m“, které představuje megabajty, a vyžaduje tedy násobení číslem 10 242.

Například při spuštění `find.py -d1 -o date,size *.*` program vyhledá všechny dnes upravované soubory (přesněji řečeno soubory upravované v posledních 24 hodinách) a vypíše jejich název, datum a velikost. Podobně při spuštění `find.py -b1m -u png,jpg,jpeg -o size *.*` program vyhledá všechny obrázkové soubory větší než jeden megabajt a vypíše jejich názvy a velikosti.

Implementujte logiku programu vytvořením kolony s použitím korutin poskytujících filtrování dle zadaných kriterií podobně jako v oddílu této lekce o korutinách, tentokráte ale u každého souboru nepředávejte jen jeho název, ale dvojici (*`název_souboru`*`, os.stat()`). Dále se pokuste na konci všechny komponenty kolony uzavřít. Ve vzorovém řešení má největší funkce na starosti volby příkazového řádku. Zbytek je docela přímočarý, ne však triviální. Řešení `find.py` má kolem 170 řádků.

LEKCE 9

Ladění, testování a profilování

V této lekci:

- Ladění
- Testování jednotek
- Profilování

Psaní programů je směsicí umění, řemeslné dovednosti a vědy, a protože je prováděno lidmi, dochází k chybám. Naštěstí existují techniky, které nám mohou pomoci především se problémům vyhnout, a pak také techniky, které slouží k identifikaci a opravě chyb, které se již staly zjevnými.

Chyby spadají do několika kategorií. Nejrychleji lze odhalit a nejsnadněji opravit chyby syntaktické, poněvadž se většinou jedná o překlep. Mnohem náročnější jsou logické chyby. Program s nimi sice běží, ale některý z aspektů jeho chování neodpoví tomu, co jsme zamýšleli nebo očekávali. Řadě chyb tohoto druhu lze předejít pomocí vývoje řízeného testy (TDD – Test Driven Development), u něhož se při přidávání nového prvku začíná napsáním testu pro tento prvek (takový test selže, protože jsme prvek dosud nepřidali), načež se implementuje samotný prvek. Další chybou je vytvářet program, který má zbytečně chabý výkon. To je téměř vždy zapříčiněno nevhodnou volbou algoritmu nebo datové struktury nebo obou. Nicméně před pokusem o optimalizaci bychom měli nejdříve zjistit, kde přesně úzké místo leží (nemusí totiž být tam, kde jej očekáváme), a až poté bychom měli místo nahodilé činnosti uvážlivě rozhodnout, jakou optimalizaci provedeme.

V první části této lekce se podíváme na výpis zásobníku volání, abychom věděli, jak odhalovat a opravovat syntaktické chyby a jak se vypořádat s neobsloužených výjimkami. Poté se podíváme, jak aplikovat vědeckou metodu na ladění, aby bylo hledání chyb co možná nejrychlejší a nejméně bolestné. Podíváme se také na podporu ladění v Pythonu. Ve druhé části se podíváme na podporu Pythonu pro psaní testů jednotek, konkrétně na modul `doctest`, s nímž jsme se seznámili již dříve (v lekci 5 a 6), a na modul `unittest`. Ukážeme si, jak použít tyto moduly na podporu vývoje řízeného testy. V poslední části této lekce se ve stručnosti podíváme na profilování, které slouží k určení úzkých míst z hlediska výkonu, což nám pomůže lépe zacílit naše optimalizační úsilí.

Ladění

V této části začneme pohledem na to, co Python dělá, když se objeví syntaktická chyba, poté se podíváme na výpisy zásobníku volání, které Python vytváří v případě, kdy dojde k neobsloužené výjimce, a pak si ukážeme, jak aplikovat vědecké metody na ladění. Ze všeho nejdříve si stručně probereme zálohování a správu verzí.

Při opravě chyby v programu existuje vždy určité riziko, že skončíme s programem, který má kromě původní chyby ještě několik nových, a který je tedy ještě horší než na začátku! A pokud nemáme žádné záložní kopie (nebo máme, ale takové, které jsou již zastaralé) a nepoužíváme žádnou správu verzí, pak může být velice těžké vrátit program zpět do stavu, kdy obsahoval jen původní chybu.

Pravidelné zálohování je podstatnou součástí programování bez ohledu na spolehlivost používaného stroje a operačního systému a na řídký výskyt selhání, protože k selhání i tak občas dochází. Ovšem zálohy bývají spíše hrubozrnnější, se soubory starými několik hodin, nebo dokonce několik dní.

Systémy pro správu verzí umožňují inkrementálně ukládat změny na libovolné požadované úrovni. Lze tedy ukládat každou jednotlivou změnu nebo každou skupinu souvisejících změn nebo prostě jen provedenou práci za určitý počet minut. Systém pro správu verzí nám umožňuje aplikovat změny (např. pro experimentování s opravami chyb), které můžeme při jejich nefunkčnosti vrátit zpět na poslední „dobrou“ verzi kódu. Proto je před zahájením ladění vždy nejlepší zavést kód do systému pro správu verzí, abychom tak měli k dispozici známou pozici, na niž se můžeme v případě problémů kdykoli vrátit.

Existuje spousta kvalitních systémů pro správu verzí schopných pracovat na mnoha platformách. V této knize budeme používat systém Bazaar (viz *http://bazaar.canonical.com/en/*), mezi další oblíbené systémy patří například Mercurial (*http://mercurial.selenic.com/*), Git (*http://git-scm.com/*) a Subversion (*http://subversion.apache.org/*). Shodou okolností jsou systémy Bazaar a Mercurial z velké většiny napsány v Pythonu. Žádný z těchto systémů nevyžaduje složité ovládání (alespoň co se základních operací týče), s pomocí kteréhokoli z nich lze však mnohem lépe předejít spoustě zbytečných bolestí.

Syntaktické chyby

Pokud se pokusíme spustit program, který obsahuje syntaktickou chybu, pak Python zastaví provádění a vypíše název souboru, číslo řádku a příslušný řádek s kurzorem (^) umístěným pod místem, kde byla detekována chyba. Zde je příklad:

```
File "blocks.py", line 383
    if BlockOutput.save_blocks_as_svg(blocks, svg)
                                                  ^
SyntaxError: invalid syntax
```

Všimli jste si chyby? Na konec podmínky příkazu `if` jsme zapomněli umístit dvojtečky.

Zde je příklad, který se objevuje docela často, v němž ale není na první pohled problém jasně zřetelný:

```
File "blocks.py", line 385
except ValueError as err:
     ^
SyntaxError: invalid syntax
```

Na uvedeném řádku není žádná syntaktická chyba, takže je špatně číslo řádku a pozice kurzoru. Obecně lze říci, že při výskytu chyby, o které jsme přesvědčeni, že se na uvedeném řádku nevyskytuje, se tato chyba téměř vždy nachází na předchozím řádku. Následuje kód od příkazu `try` po `except`, kde Python hlásí výskyt chyby. Před dalším čtením si vyzkoušejte, zda dokážete odhalit chybu:

```
try:
    blocks = parse(blocks)
    svg = file.replace(".blk", ".svg")
    if not BlockOutput.save_blocks_as_svg(blocks, svg):
        print("Chyba: nemohu uložit {0}".format(svg)
except ValueError as err:
```

Našli jste problém? Je velice snadné jej přehlédnout, poněvadž se nachází na řádku, který je umístěn před řádkem, o němž Python hlásí, že obsahuje chybu. Uzavřeli jsme závorky metody `str.format()`, ne však závorky funkce `print()`, což znamená, že na konci řádku chybí uzavírací závorka, což si Python neuvědomí, dokud nenarazí na klíčové slovo `except` na následujícím řádku. Opomenutí závorky na konci řádku je docela běžné, zejména při použití funkce `print()` s metodou `str.format()`, ovšem chyba je obvykle hlášena až na následujícím řádku. Podobně také v případě, kdy chybí uzavírací hranatá závorka nebo složená závorka množiny či slovníku, bude Python obvykle

hlásit problém na následujícím (neprázdném) řádku. Nicméně plusem je, že takovéto syntaktické chyby lze velice snadno opravit.

Chyby za běhu programu

Pokud za běhu programu dojde k neobsloužené výjimce, pak Python zastaví provádění programu a vypíše zásobník volání. Zde je příklad výpisu zásobníku volání pro nějakou neobslouženou výjimku:

```
Traceback (most recent call last):
  File "blocks.py", line 392, in <module>
    main()
  File "blocks.py", line 381, in main
    blocks = parse(blocks)
  File "blocks.py", line 174, in recursive_descent_parse
    return data.stack[1]
IndexError: list index out of range
```

Takovéto výpisy zásobníku volání (označované též jako *zpětné výpisy volání*) se čtou od posledního řádku směrem k prvnímu. Na posledním řádku je uvedena neobsloužená výjimka, ke které došlo. Nad tímto řádkem je název souboru, číslo řádku a název funkce následovaný řádkem, který tuto výjimku způsobil (to vše rozložené na dva řádky). Pokud funkce, v níž došlo k vyvolání výjimky, byla zavolána jinou funkcí, pak se nad těmito informacemi objeví příslušný název souboru, číslo řádku a název funkce. A pokud byla i tato funkce zavolána jinou funkcí, postupuje se stejně, a to až po začátek zásobníku volání. (Je třeba poznamenat, že názvy souborů ve výpisu zásobníku volání obsahují celou cestu, kterou ale ve většině našich příkladů kvůli lepší čitelnosti vynecháme.)

Odkazy na funkce ➤ 331

V tomto příkladu tedy došlo k výjimce `IndexError`, což znamená, že `data.stack` je nějaký druh posloupnosti, která ale neobsahuje prvek na pozici 1. K chybě došlo na řádku 174 programu `blocks.py` ve funkci `recursive_descent_parse()`, která byla zavolána na řádku 381 ve funkci `main()`. (Důvodem, proč je název funkce na řádku 381 jiný, tj. `parse()` místo `recursive_descent_parse()`, je to, že proměnná `parse` je nastavena na jednu z několika různých funkcí v závislosti na argumentech příkazového řádku zadaných programu. V obvyklých případech jsou oba názvy shodné.) Volání funkce `main()` bylo provedeno na řádku 392, což je příkaz, od něhož bylo zahájeno provádění programu.

I když výpis zásobníku volání působí na první pohled odstrašujícím dojmem, díky znalosti jeho struktury můžeme nyní ocenit jeho užitečnost. V tomto případě se přesně dozvíme, kde máme problém hledat, i když jeho řešení už samozřejmě musíme vymyslet sami.

Zde je další příklad výpisu zásobníku zpětného volání:

```
Traceback (most recent call last):
  File "blocks.py", line 392, in <module>
    main()
  File "blocks.py", line 383, in main
    if BlockOutput.save_blocks_as_svg(blocks, svg):
  File "BlockOutput.py", line 141, in save_blocks_as_svg
    widths, rows = compute_widths_and_rows(cells, SCALE_BY)
```

```
  File "BlockOutput.py", line 95, in compute_widths_and_rows
    width = len(cell.text) // cell.columns
ZeroDivisionError: integer division or modulo by zero
```

Zde došlo k problému v modulu (`BlockOutput.py`), který je volán programem `blocks.py`. Výše uvedený výpis zásobníku volání nás sice zavede až do místa, kde je problém jasně vidět, ne však do místa, kde k němu došlo. Hodnota atributu `cell.columns` je ve funkci `compute_widths_and_rows()` na řádku 95 modulu `BlockOutput.py` zcela jasně 0 (koneckonců způsobila vyvolání výjimky `ZeroDivisionError`), my se ale musíme podívat na předchozí řádky a zjistit, kde a proč byla atributu `cell.columns` přiřazena nesprávná hodnota.

V některých případech odhalí výpis zásobníku volání výjimku, k níž došlo ve standardní knihovně Pythonu nebo v knihovně třetí strany. Přestože to může znamenat, že ona knihovna obsahuje nějakou chybu, jedná se v drtivé většině případů o chybu v našem vlastním kódu. Zde je příklad takového výpisu zásobníku volání s použitím Pythonu 3.0:

3.0

```
Traceback (most recent call last):
  File "blocks.py", line 392, in <module>
    main()
  File "blocks.py", line 379, in main
    blocks = open(file, encoding="utf8").read()
  File "/usr/lib/python3.0/lib/python3.0/io.py", line 278, in __new__
    return open(*args, **kwargs)
  File "/usr/lib/python3.0/lib/python3.0/io.py", line 222, in open
    closefd)
  File "/usr/lib/python3.0/lib/python3.0/io.py", line 619, in __init__
    _fileio._FileIO.__init__(self, name, mode, closefd)
IOError: [Errno 2] No such file or directory: 'hierarchy.blk'
```

Výjimka `IOError` uvedená na konci nám jasně říká, v čem je problém. Avšak tato výjimka byla vyvolána v modulu `io` standardní knihovny. V takovýchto případech je nejlepší pokračovat ve čtení výpisu směrem vzhůru, dokud nenajdeme první soubor, který je souborem našeho programu (nebo jedním z modulů, který jsme pro něj vytvořili). V tomto případě tedy zjistíme, že první odkaz na náš program je v souboru `blocks.py` na řádku 379 ve funkci `main()`. Vypadá to, že zde máme volání funkce `open()`, které jsme ale neumístili do bloku `try ... except` ani jsme pro něj nepoužili příkaz `with`.

Python 3.1 je oproti Pythonu 3.0 o něco chytřejší a uvědomí si, že chceme najít chybu v našem vlastním kódu, a ne ve standardní knihovně. Proto vytváří mnohem kompaktnější a užitečnější výpis zásobníku volání. Například:

3.1

```
Traceback (most recent call last):
  File "blocks.py", line 392, in <module>
    main()
  File "blocks.py", line 379, in main
    blocks = open(file, encoding="utf8").read()
IOError: [Errno 2] No such file or directory: 'hierarchy.blk'
```

3.1

Tímto způsobem jsme ušetřeni všech nepodstatných detailů a můžeme tak snadno zjistit, o jaký problém se jedná (spodní řádek) a kde k němu došlo (řádek nad ním).

Nehledě tedy na velikost výpisu zásobníku volání je neobsloužená výjimka uvedená vždy na posledním řádku, takže nám pak stačí jen postupovat zpět, dokud nenajdeme soubor našeho programu nebo některého z jeho modulů. Problém bude téměř jistě na řádku, který zde Python uvádí, nebo na řádku předchozím.

V tomto konkrétním příkladu je jasné, že bychom měli upravit program `blocks.py` tak, aby se elegantně vypořádal i s případem, kdy jsou zadány názvy neexistujících souborů. Jedná se o chybu použitelnosti, která by měla být také klasifikována jako logická chyba, protože ukončení programu a výpis zásobníku volání nelze považovat za přijatelné chování programu.

Ve skutečnosti bychom s ohledem na dobré zásady a ohleduplnost vůči našim uživatelům měli vždy zachytávat všechny *relevantní* výjimky a měli bychom identifikovat ty, které považujeme za možné, jako je například `EnvironmentError`. Obecně bychom *neměli* používat zachytávání všech typů výjimek ve stylu `except:` či `except Exception:`, i když použití druhé varianty umístěné na nejvyšší úrovni programu pro zamezení pádů programu může být vhodné, ovšem jen tehdy, pokud každou zachycenou výjimku vždy nahlásíme, aby se žádná z nich jen tak tiše nevytratila.

Výjimky, které zachytíme a ze kterých se již nemůžeme dostat, bychom měli nahlásit ve formě chybových zpráv a nevystavovat tak své uživatele výpisům zásobníku volání, jež na nezasvěcené působí děsivě. Pro programy s grafickým uživatelským rozhraním platí to stejné, tedy až na to, že pro upozornění uživatele na problém se obvykle používá okno se zprávou. A u serverových programů, které obyčejně běží bez obsluhy, bychom měli chybové zprávy zapisovat do protokolu daného serveru.

Hierarchie výjimek Pythonu byla navržena tak, aby zachycení výjimky `Exception` nepokrylo úplně všechny výjimky. Konkrétně se takto nezachytí výjimka `KeyboardInterrupt`, takže když v konzolové aplikaci uživatel stiskne Ctrl+C, program se ukončí. Pokud se rozhodneme tuto výjimku zachytit, pak riskujeme, že uživatele uzamkneme v programu, který nebude moci opustit. K tomu může dojít kvůli chybě v kódu obsluhující výjimku, která zamezí ukončení programu nebo propagaci výjimky. (Samozřejmě lze zabít i proces „nepřerušitelného“ program, což ale všichni uživatelé nevědí.) Pokud tedy zachytáváme výjimku `KeyboardInterrupt`, musíme být extrémně opatrní a provést jen minimální množství operací pro uložení a úklid paměti a poté program okamžitě ukončit. A u programů, které nepotřebují nic ukládat ani uklízet, je nejvhodnější výjimku `KeyboardInterrupt` vůbec nezachytávat a prostě nechat program skončit.

Jednou z obrovských předností Pythonu 3 je to, že činí jasný rozdíl mezi holými bajty a řetězci. To však může někdy vést k neočekávaným výjimkám během předávání objektu typu `bytes` tam, kde je očekáván řetězec, a naopak. Například:

```
Traceback (most recent call last):
  File "program.py", line 918, in <module>
    print(datetime.datetime.strptime(date, format))
TypeError: strptime() argument 1 must be str, not bytes
```

Když narazíme na problém, jako je tento, pak můžeme buď provést převod (v tomto případě předáním `date.decode("utf8")`), nebo opatrně procházet kód a zjistit, kde a proč je daná proměnná objektem typu `bytes` a ne `str`, a opravit problém přímo v jeho zdroji.

Když předáváme řetězec tam, kde je očekáván objekt typu `bytes`, pak obdržíme chybovou zprávu, která není příliš srozumitelná a v Pythonu 3.0 je jiná než v Pythonu 3.1. Například v Pythonu 3.0:

3.x

3.0

```
Traceback (most recent call last):
  File "program.py", line 2139, in <module>
    data.write(info)
TypeError: expected an object with a buffer interface
```

V Pythonu 3.1 je text chybové zprávy malinko vylepšen:

3.1

```
Traceback (most recent call last):
  File "program.py", line 2139, in <module>
    data.write(info)
TypeError: 'str' does not have the buffer interface
```

V obou případech tkví problém v tom, že předáváme řetězec v místě, kde se očekává objekt typu `bytes`, `bytearray` nebo podobného typu. Můžeme tedy provést buď převod (v tomto případě předáním `info.encode("utf8")`), nebo projít kód, najít zdroj problému a opravit jej tam.

Python 3.0 zavádí podporu pro řetězení výjimek, což znamená, že výjimka vyvolaná v reakci na jinou výjimku může obsahovat detaily o původní výjimce. Není-li obsloužena zřetězená výjimka, pak výpis zásobníku volání obsahuje nejen tuto nezachycenou výjimku, ale také výjimku, která ji způsobila (za předpokladu, že byla zřetězena). Postup při ladění zřetězených výjimek je v podstatě stejný jako předtím. Začneme na konci a postupujeme zpět, dokud nenajdeme problém v našem vlastním kódu. Toto však neprovádíme jen pro poslední výjimku, ale celý proces můžeme opakovat pro každou z výše umístěných řetězených výjimek, dokud nenajdeme skutečný zdroj problému.

Výhody řetězení výjimek můžeme využít také ve svém vlastním kódu. Například tehdy, když chceme použít třídu představující naši vlastní výjimku, ale zároveň chceme, aby byl viditelný i původní problém.

```
class InvalidDataError(Exception): pass

def process(data):
    try:
        i = int(data)
        ...
    except ValueError as err:
        raise InvalidDataError("Byla přijata neplatná data") from err
```

Pokud zde převod `int()` selže, vyvolá se výjimka `ValueError`. Poté vyvoláme naši vlastní výjimku, ovšem s klauzulí `from err`, která vytvoří zřetězenou výjimku obsahující naši vlastní výjimku plus výjimku v proměnné `err`. Dojde-li k vyvolání a nezachycení výjimky `InvalidDataError`, pak bude výsledný výpis zásobníku volání vypadat nějak takto:

```
Traceback (most recent call last):
  File "application.py", line 249, in process
    i = int(data)
ValueError: invalid literal for int() with base 10: '17.5 '

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "application.py", line 288, in <module>
    print(process(line))
  File "application.py", line 283, in process
    raise InvalidDataError("Byla přijata neplatná data") from err
__main__.InvalidDataError: Byla přijata neplatná data
```

Na konci naše vlastní výjimka a text vysvětlují, o jaký problém jde, přičemž řádky nad nimi ukazují, kde byla tato výjimka vyvolána (řádek 283) a kde byla způsobena (řádek 288). Můžeme jít ale ještě dále dozadu až do zřetězené výjimky, která poskytuje více podrobností o specifické chybě a která ukazuje řádek, který výjimku spustil (249). Detailní vysvětlení a další informace o zřetězených výjimkách najdete v dokumentu PEP 3134.

Vědecké ladění

Pokud náš program běží, ale nevykazuje očekávané nebo požadované chování, jedna se o chybu (logickou chybu), kterou musíme odstranit. Nejlepším způsobem pro odstranění takovýchto chyb je především zamezení jejich výskytu pomocí vývoje řízeného testy (Test Driven Development). Nicméně některé chyby se do programu vždy dostanou, takže i při použití vývoje řízeného testy je ladění nezbytnou dovedností, kterou je třeba si osvojit.

V tomto oddílu si nastíníme přístup k ladění na základě vědecké metody. Tento přístup je vysvětlen tak podrobně, že by to mohlo vypadat, že se pro odstranění „obyčejné“ chyby jedná o příliš mnoho práce. Nicméně vědomým sledováním tohoto procesu nebudeme muset plýtvat časem na „nahodilé“ ladění a po nějaké době tento proces přijmeme za vlastní, takže jej budeme provádět nevědomky, a proto velice rychle.*

Abychom byli schopni eliminovat chybu, musíme být schopni provést následující:

1. chybu reprodukovat,
2. chybu lokalizovat,
3. chybu opravit,
4. opravu otestovat.

Reprodukování chyby je někdy snadné, vyskytuje-li se při každém spuštění, a jindy obtížné, vyskytuje-li pouze občas. V každém případě bychom se měli pokusit zredukovat závislosti chyby, což znamená najít nejmenší vstup a nejmenší množství zpracování, které je nezbytné pro vytvoření chyby.

* Myšlenky použité v tomto oddílu byly inspirovány lekcí „Ladění“ z knihy „Dokonalý kód“ – viz *http://knihy.cpress.cz/K1148*.

Jakmile jsme schopni chybu reprodukovat, tak máme data (vstupní data a volby a nesprávné výsledky), která jsou nezbytná k tomu, abychom mohli aplikovat vědeckou metodu pro její nalezení a opravení. Tato metoda má tři kroky.

1. Vymyslete nějaké vysvětlení, tj. hypotézu, která rozumně objasňuje příčinu chyby.
2. Vytvořte experiment k otestování této hypotézy.
3. Spusťte experiment.

Spuštění experimentu by nám mělo pomoci s nalezením chyby a s proniknutím do jejího řešení. (Ke způsobu vytvoření a spuštění experimentu se vrátíme za okamžik.) Jakmile se rozhodneme, jak chybu eliminovat (a jakmile uložíme svůj kód do systému pro správu verzí, abychom v případě potřeby mohli opravu vrátit zpět), můžeme napsat opravu.

Po implementaci opravy ji musíme otestovat. Přirozeně musíme testováním zjistit, zda chyba, která má být opravena, je skutečně pryč. To však nestačí – koneckonců naše oprava mohla vyřešit chybu, kterou jsme se zabývali, ale zároveň zanést do programu jinou chybu, která ovlivní nějaký jiný aspekt programu. Kromě testování opravy chyby musíme proto také spustit všechny testy programu, abychom zvětšili naši jistotu, že oprava chyby nezpůsobila žádné nechtěné vedlejší efekty.

Některé chyby mají určitou strukturu, takže po každé opravě chyby je vhodné položit si otázku, zda existují ještě další místa v programu nebo jeho modulech, které mohou obsahovat podobné chyby. Je-li tomu tak, pak se můžeme podívat, zda již máme testy, které by tyto chyby odhalily, a pokud ne, tak bychom takovéto testy měli přidat. Pokud tímto způsobem odhalíme chyby, musíme se s nimi vypořádat výše uvedeným způsobem.

Nyní již tedy máme dobrý přehled o procesu ladění, a proto se zaměříme na to, jak vytvářet a spouštět experimenty k otestování našich hypotéz. Začneme pokusem o izolování chyby. V závislosti na povaze programu a chyby můžeme napsat testy, které vyzkoušejí program, například tím, že jej nakrmí daty, o nichž víme, že se zpracují správně, přičemž tato data postupně mění, abychom mohli přesně určit místo, kde dochází k selhání zpracování. Jakmile máme představu o tom, kde se problém nachází (ať už díky testování nebo na základě dedukce), můžeme své hypotézy otestovat.

Jaký druh hypotéz můžeme vymyslet? Inu, zpočátku může jít o něco tak jednoduchého, jako je podezření, že určitá funkce či metoda vrací při použití určitých vstupních dat a voleb chybná data. Prokáže-li se taková hypotéza jako správná, tak ji můžeme zjemnit, aby byla konkrétnější. Například identifikováním určitého příkazu či sady ve funkci, o které si myslíme, že v některých případech provádí nesprávný výpočet.

K otestování naší hypotézy musíme bezprostředně před ukončením funkce zkontrolovat argumenty, které přijímá, hodnoty jejích lokálních proměnných a její návratovou hodnotu. Poté můžeme spustit program s daty, o nichž víme, že vedou k chybám, a zkontrolovat podezřelou funkci. Pokud argumenty vstupující do funkce neodpovídají našemu očekávání, bude problém nejspíše někde výše na zásobníku volání, takže bychom měli začít celý proces znovu, ovšem tentokrát s podezřením na funkci, jež volá tu, kterou jsme nyní testovali. Jsou-li ale všechny příchozí argumenty vždy platné, musíme se podívat na lokální proměnné a návratovou hodnotu. Mají-li vždy správné hodnoty, pak musíme přijít s novou hypotézou, poněvadž podezřelá funkce se chová správně. Je-li však návratová hodnota nesprávná, pak víme, že musíme danou funkci prostudovat ještě podrobněji.

V jak ale praxi experiment provedeme? To znamená, jak otestujeme hypotézu, že se určitá funkce nechová správně? Jedna z možností, jak začít, spočívá v mentálním „provedení" dané funkce. Tato možnost je realizovatelná pro řadu malých funkcí a s jistou praxí i pro větší a její výhoda tkví v tom, že se dobře seznámíme s chováním dané funkce. V nejlepším případě může vést k vylepšené nebo specifičtější hypotéze – například, že zdrojem problému je určitý příkaz či sada. Ovšem ke správnému provedení experimentu musíme program instrumentovat tak, abychom viděli, co se při zavolání podezřelé funkce děje.

Existují dva způsoby instrumentace programu – intrusivně vkládáním příkazů `print()` nebo neintrusivně (což je obvyklejší) pomocí ladicího nástroje (debuggeru). Oba přístupy se používají pro dosažení stejného cíle a oba jsou platné, ovšem někteří programátoři dávají přednost jednomu či druhému. My si stručně popíšeme oba přístupy, přičemž začneme příkazy `print()`.

Při použití příkazů `print()` můžeme začít jeho umístěním na začátek funkce a nechat jej vypsat její argumenty. Poté přidáme těsně před každý příkaz `return` (nebo na konec funkce, nemá-li žádný příkaz `return`) volání `print(locals(), "\n")`. Vestavěná funkce `locals()` vrací slovník, jehož klíči jsou názvy lokálních proměnných a hodnotami jejich hodnoty. Můžeme samozřejmě vypsat pouze ty proměnné, které nás zajímají. Všimněte si, že na konec připojujeme znak nového řádku, který bychom mohli přidat i k prvnímu příkazu `print()`, aby mezi každou skupinou proměnných zůstal prázdný řádek zlepšující čitelnost. (Další možností je použít nějaký druh dekorátoru provádějícího protokolování, jako je například ten, který jsme vytvořili v lekci 8 – viz stranu 347.)

Pokud při spuštění instrumentovaného programu zjistíme, že argumenty jsou sice správné, ale návratová hodnota je chybná, pak víme, že jsme lokalizovali zdroj chyby a můžeme danou funkci dále prozkoumávat. Pokud pečlivé prostudování funkce nevede k odhalení místa problému, můžeme jednoduše přidat další příkaz `print(locals(), "\n")` přímo doprostřed funkce. Po opětovném spuštění programu bychom měli vědět, zda k problému dochází v první či druhé polovině funkce, umístit další příkaz `print(locals(), "\n")` doprostřed příslušné poloviny a celý proces opakovat znovu, dokud nenajdeme příkaz, který danou chybu způsobuje. Tímto způsobem se velice rychle dostaneme do místa, kde k problému dochází; ve většině případů představuje nalezení problému polovinu práce nezbytné pro jeho vyřešení.

Kromě přidávání příkazů `print()` můžeme použít ladicí nástroj. Python obsahuje dva standardní ladicí nástroje. Jeden je dodáván jako modul (`pdb`) a lze jej použít interaktivně na příkazovém řádku (např. `python3 -m pdb my_program.py`). (Ve Windows musíme samozřejmě nahradit název `python3` celou cestou, například `C:\Python31\python.exe`.) Nicméně nejjednodušší je přidat příkaz `import pdb` do samotného programu a na začátek funkce, kterou chceme prozkoumat, vložit příkaz `pdb.set_trace()`. Jakmile program spustíme, modul `pdb` jej v okamžiku zavolání funkce `pdb.set_trace()` zastaví a umožní nám provádět krokování, nastavovat zarážky a zkoumat proměnné.

Zde je ukázkový běh programu, který byl instrumentován přidáním příkazu import `pdb` mezi importy a příkazu `pdb.set_trace()` na začátek funkce `calculate_median()`. (Námi zadávaný vstup je zvýrazněn, ovšem stisky klávesy Enter uváděny nejsou.)

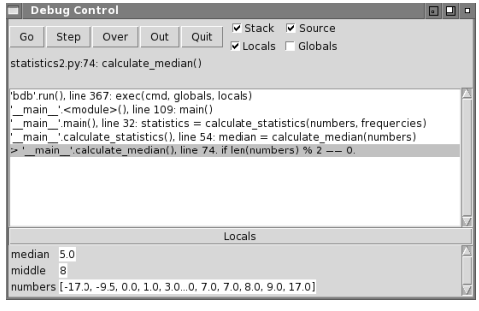
```
python3 statistics.py sum.dat
> statistics.py(73)calculate_median()
-> numbers = sorted(numbers)
(Pdb) s
```

```
> statistics.py(74)calculate_median()
-> middle = len(numbers) // 2
(Pdb)
> statistics.py(75)calculate_median()
-> median = numbers[middle]
(Pdb)
> statistics.py(76)calculate_median()
-> if len(numbers) % 2 == 0:
(Pdb)
> statistics.py(78)calculate_median()
-> return median
(Pdb) p middle, median, numbers
(8, 5.0, [-17.0, -9.5, 0.0, 1.0, 3.0, 4.0, 4.0, 5.0, 5.0, 5.0, 5.5,
6.0, 7.0, 7.0, 8.0, 9.0, 17.0])
(Pdb) c
```

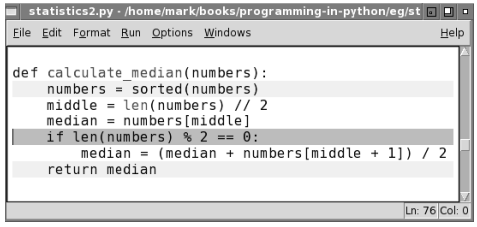
Příkazy se zadávají modulu `pdb` zápisem jejich názvu a stiskem klávesy Enter na příkazovém řádku `(Pdb)`. Pokud jen stiskneme samotnou klávesu Enter, zopakuje poslední příkaz. Zde jsme tedy zapsali příkaz `s` (což znamená step – krok, tj. provedení zobrazeného příkazu), jehož opakováním (stisky klávesy Enter) jsme krokovali příkazy ve funkci `calculate_median()`. Jakmile jsme dosáhli příkazu return, vypsali jsme příkazem `p` (print – výpis) hodnoty, které nás zajímají. A nakonec jsme příkazem `c` (continue – pokračovat) nechali program pokračovat dál. Z této kratičké ukázky by mělo být patrné, jakým způsobem se s modulem `pdb` pracuje. Tento modul nabízí samozřejmě mnohem více, než co jsme si zde ukázali.

Je mnohem snazší použít modul `pdb` na instrumentovaný program, jak jsme to provedli zde, než na neinstrumentovaný. To ale vyžaduje přidání příkazu `import` a zavolání funkce `pdb.set_trace()`, což znamená, že použití modulu `pdb` je stejně intrusivní jako použití příkazů `print()`, třebaže nabízí užitečné možnosti, jako jsou například zarážky.

Dalším standardním ladicím nástrojem je IDLE, který stejně jako modul `pdb` podporuje krokování, zarážky a zkoumání proměnných. Ladicí okno editoru IDLE je zachyceno na obrázku 9.1 a jeho okno pro úpravu kódu se zvýrazněním zarážek a aktuálního řádku je na obrázku 9.2.



Obrázek 9.1: Ladicí okno editoru IDLE ukazující zásobník volání a aktuální lokální proměnné



Obrázek 9.2: Okno editoru IDLE pro úpravu kódu během ladění

Velkou výhodou editoru IDLE oproti modulu `pdb` je, že svůj kód nemusíme nijak instrumentovat. Editor IDLE je totiž dostatečně chytrý, aby zvládl ladit náš kód tak jak je, takže není žádným způsobem intrusivní.

Naneštěstí je třeba poznamenat, že v době psaní této knihy je editor IDLE poněkud slabší při spouštění programů, které vyžadují argumenty na příkazovém řádku. Jediným způsobem, jak toto provést, je spustit editor IDLE z konzoly s požadovanými argumenty, například `idle3 -d -r statistics.py sum.dat`. Argument `-d` říká editoru IDLE, aby ihned spustil ladění, a argument `-r` říká, aby spustil následující program s libovolnými argumenty, které jsou umístěny za ním. Nicméně v případě programů, které nevyžadují argumenty na příkazovém řádku (nebo u kterých jsme ochotni upravit kód tak, aby byly argumenty umístěny přímo v kódu, čímž si usnadníme ladění), je ladění v editoru IDLE poměrně výkonné a pohodlné. (Kód zachycený na obrázku 9.2 obsahuje shodou okolností chybu – místo `middle + 1` by mělo být `middle - 1`.)

Ladění programů napsaných v jazyku Python není o nic těžší než ladění v kterémkoliv jiném jazyku. Ve srovnání s kompilovanými jazyky je navíc snazší, protože zde nemáme sestavování, které je jinak nutné provést po každé změně. A pokud budeme pečlivě používat tuto vědeckou metodu, pak bude hledání chyb docela přímočaré, i když jejich oprava je již o něčem jiném. Přesto se v ideálním případě chceme většině chyb vyhnout. A kromě důkladného promýšlení svého návrhu a pozorného psaní svého kódu je nejlepším způsobem pro zamezení chyb používat vývoj řízený testy, což je téma, s nímž se seznámíme v následující části.

Testování jednotek

Psaním testů pro naše programy (jsou-li napsány dobře) můžeme snížit výskyt chyb a zvýšit jistotu, že se naše programy chovají dle očekávání. Testování však obecně nemůže zaručit správnost, poněvadž pro většinu netriviálních programů je rozsah možných vstupů a rozsah možných výpočtů tak obrovský, že ve skutečnosti jsme schopni otestovat pouze jejich nepatrnou část. Nicméně pečlivým výběrem toho, co se má testovat, můžeme zlepšit kvalitu svého kódu.

Provádět můžeme množství odlišných druhů testování, mezi něž patří testování použitelnosti, testování funkčnosti a integrační testování. My se ale zaměříme jen na testování jednotek, což jsou testy jednotlivých funkcí, tříd a metod, které zajišťují, aby se chovaly dle našich očekávání.

Klíčovým principem vývoje řízeného testy je, že kdykoliv chceme přidat nějaký prvek (např. novou metodu do třídy), pak pro něj *nejdříve* napíšeme test. Ten samozřejmě selže, protože požadovanou metodu jsme dosud nenapsali. Nyní tuto metodu napíšeme, a jakmile projde testem, můžeme opětovně spustit *všechny* testy, abychom měli jistotu, že nově doplněný prvek nemá žádné vedlejší efekty.

Jakmile se všechny testy dokončí (včetně toho, který jsme přidali pro náš nový prvek), můžeme kód uložit se slušnou mírou jistoty, že dělá to, co má – ovšem za předpokladu, že naše testy mají adekvátní podobu. Pokud například chceme napsat funkci, která na určitou indexovou pozici vkládá řetězec, pak můžeme začít s použitím vývoje řízeného testy takto:

```
def insert_at(string, position, insert):
    """Vrátí kopii řetězce s argumentem insert vloženým na zadanou pozici

    >>> string = "ABCDE"
    >>> result = []
    >>> for i in range(-2, len(string) + 2):
    ... result.append(insert_at(string, i, "-"))
    >>> result[:5]
     ['ABC-DE', 'ABCD-E', '-ABCDE', 'A-BCDE', 'AB-CDE']
    >>> result[5:]
     ['ABC-DE', 'ABCD-E', 'ABCDE-', 'ABCDE-']
    """
    return string
```

Funkcím či metodám, které nic nevracejí (ve skutečnosti vracejí hodnotu `None`), dáváme obvykle sadu sestávající z příkazu `pass` a u těch, které vracejí nějakou hodnotu, vracíme buď nějakou konstantu (např. 0), nebo jeden z argumentů, který nijak neměníme, což jsme provedli ve výše uvedeném příkladu. (Ve složitějších situacích může být užitečnější vracet falešné objekty – pro tyto účely jsou k dispozici moduly třetích stran, které poskytují tzv. „mock-up" objekty neboli atrapy.)

Výše uvedený dokumentační test při svém spuštění selže, přičemž vypíše každý z řetězců (`'ABCD-EF'`, `'ABCDE-F'` atd.), který očekával, a řetězce, které ve skutečnosti obdržel (všechny jsou `'ABCDEF'`). Jakmile jsme přesvědčeni, že dokumentační test je dostatečný a správný, můžeme napsat tělo funkce, které je v tomto případě tvořeno jen příkazem `return string[:position] + insert + string[position:]`. (A pokud jsme napsali `return string[:position] + insert` a poté zkopírovali `string[:position]` a vložili na konec, abychom si ušetřili psaní, náš dokumentační test by tuto chybu odhalil.)

Standardní knihovna Pythonu nabízí dva moduly pro testování jednotek. Modul `doctest`, s nímž jsme se již stručně seznámili v této i předchozích lekcích (v lekci 5 na straně 203 a 6 na straně 242), a modul `unittest`. Kromě toho existují testovací nástroje pro Python od třetích stran. Mezi nejvýznamnější z nich patří modul `nose` (*code.google.com/p/python-nose*), který chce být obsáhlejší a užitečnější než standardní modul `unittest` a přitom s ním zůstat kompatibilní, a modul `py.test` (*codespeak.net/py/dist/test/test.html*), který se na rozdíl od modulu `unittest` vydává poněkud odlišnější cestou a snaží se v maximální možné míře eliminovat „omáčku" kolem testovaného kódu. Oba tyto nástroje třetích stran podporují objevování testů, takže není nutné psát zastřešující testovací program – testy budou automaticky vyhledány. Díky tomu lze snadno otestovat celý strom kódu nebo jen jeho část (např. jen ty moduly, na nichž jsme pracovali). Před rozhodnutím, který testovací nástroj použít, je vhodné oba tyto moduly třetích stran (a jakýkoliv další, který působí dobře) prozkoumat.

Tvorba dokumentačních testů je jednoduchá. Testy zapisujeme do dokumentačních řetězců modulu, funkce, třídy a metody a na konec modulů přidáme jen následující tři řádky kódu:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Dokumentační testy můžeme použít také uvnitř programů. Například program `blocks.py`, jehož moduly budeme probírat později (v lekci 14), obsahuje dokumentační testy pro své funkce, končí však tímto kódem:

```
if __name__ == "__main__":
    main()
```

Tento kód jednoduše zavolá funkci `main()` programu, ale neprovede jeho dokumentační testy. K provedení jeho dokumentačních testů můžeme postupovat dvěma způsoby. Jeden spočívá v importování modulu `doctest` a spuštění programu (například na příkazovém řádku napíšeme `python3 -m doctest blocks.py`, přičemž ve Windows nahradíme python3 celou cestou, jako je `C:\Python31\python.exe`). Pokud všechny testy proběhnou v pořádku, pak neobdržíme žádný výstup, a proto je někdy vhodnější spustit raději `python3 -m doctest blocks.py -v`, protože tak získáme výpis všech prováděných dokumentačních testů a na konci souhrnné výsledky.

Další možnost, jak provádět dokumentační testy, spočívá ve vytvoření samostatného testovacího programu pomocí modulu `unittest`. Tento modul je koncepčně modelován podle knihovny Javy pro testování jednotek s názvem JUnit a používá se pro vytváření testovacích sad, které obsahují testovací případy. Modul `unittest` dokáže vytvářet testovací případy na základě dokumentačních testů, aniž by musel vědět cokoliv o tom, co daný program či modul obsahuje, tedy kromě skutečnosti, že má dokumentační testy. K vytvoření testovací sady pro program `blocks.py` tedy můžeme vytvořit následující jednoduchý program (který jsme nazvali `test_blocks.py`):

```
import doctest
import unittest
import blocks
suite = unittest.TestSuite()
suite.addTest(doctest.DocTestSuite(blocks))
runner = unittest.TextTestRunner()
print(runner.run(suite))
```

Všimněte si, že při použití tohoto přístupu máme implicitní omezení na název našich programů, které musejí být platnými názvy modulů, takže program s názvem `convert-incidents.py` nemůže mít test, jako je tento, protože `import convert-incidents` není platný příkaz, neboť identifikátory jazyka Python nemohou obsahovat spojovníky. (Je sice možné toto omezení obejít, ale nejjednodušším řešením je používat pro programy takové názvy, které jsou platnými názvy modulů, například tedy nahradit spojovníky podtržítky.)

Výše uvedená struktura (vytvořit testovací sadu, přidat jeden či více testovacích případů či testovacích sad, spustit zastřešujících testovací sadu a vypsat výsledky) je typická pro testy založené na modulu `unittest`. Tento konkrétní příklad vytvoří při svém spuštění následující výstup:

```
...
----------------------------------------------------------------------
```

```
Ran 3 tests in 0.244s

OK
<unittest._TextTestResult run=3 errors=0 failures=0>
```

Při provedení každého testovacího případu se vypíše jedna tečka (proto jsou na začátku výpisu tři tečky), poté následuje řádek s pomlčkami a potom souhrn testu. (Pokud některé testy selžou, bude výstup přirozeně obsahovat mnohem více informací.)

Pokud se snažíme mít samostatné testy (obvykle jeden pro každý program a modul, který chceme testovat), pak může být lepší místo dokumentačních testů použít přímo možnosti modulu `unittest`, zejména pak tehdy, jsme-li zvyklí na testování ve stylu knihovny JUnit. Modul `unittest` udržuje naše testy oddělené od vlastního kódu, což je zvláště užitečné pro rozsáhlé projekty, v nichž nemusejí být tvůrci testů a vývojáři nutně stejnými osobami. Testy jednotek se při použití modulu `unittest` zapisují jako samostatné moduly Pythonu, takže nejsou omezeny tím, co lze pohodlně a rozumně zapsat do dokumentačního řetězce.

Modul `unittest` definuje čtyři hlavní pojmy. *Testovací přípravek* (test fixture) je termín používaný pro popis kódu nezbytného pro přípravu testu (a pro jeho úklid). Typickým příkladem je vytvoření vstupního souboru pro použití v testu a vymazání vstupního a výsledného výstupního souboru na konci. *Testovací sada* (test suite) je kolekce testovacích případů a *testovací případ* (test case) je základní jednotkou testování (testovací sady jsou kolekce testovacích případů nebo dalších testovacích sad). Praktické příklady si ukážeme za okamžik. *Spouštěč testů* (test runner) je objekt, který provede jeden nebo více testovacích sad.

Testovací sada se obvykle zhotoví vytvořením podtřídy třídy `unittest.TestCase`, kde každá metoda, která má název začínající na „test“, představuje testovací případ. Pokud potřebujeme provést jakoukoli přípravu, pak ji můžeme provést v metodě `setUp()`. A podobně jakýkoliv úklid můžeme implementovat v metodě s názvem `tearDown()`. Uvnitř testů můžeme využít řadu metod třídy `unittest.TestCase`, mezi něž patří `assertTrue()`, `assertEqual()`, `assertAlmostEqual()` (užitečné pro testování čísel s pohyblivou řádovou čárkou), `assertRaises()` a mnoho dalších včetně spousty obrácených testů, jako je `assertFalse()`, `assertNotEqual()`, `failIfEqual()`, `failUnlessEqual()` a tak dále.

Cvičení `Atomic.py` ➢ 396

Modul `unittest` je dobře zdokumentovaný a obsahuje spoustu funkčních prvků. My si zde ale prostudujeme jen velice jednoduchou testovací sadu, abychom získali alespoň základní představu o tom, jak se tento modul používá. Tento příklad je řešením jednoho ze cvičení zadaných na konci lekce 8. Měli jste vytvořit modul `Atomic`, který bylo možné použít jako správce kontextu pro zajištění, že se provedou buď všechny změny aplikované na seznam, množinu či slovník, nebo žádná z nich. Modul `Atomic.py`, který je k dispozici jako ukázkové řešení, používá 30 řádků kódu pro implementaci třídy `Atomic` a dalších přibližně 100 řádků pro dokumentační testy. Nyní vytvoříme modul `test_Atomic.py` pro nahrazení dokumentačních testů testy, které využívají modul `unittest`, takže budeme moci dokumentační testy vymazat a nechat v souboru `Atomic.py` jen kód nezbytný k poskytování jeho funkčnosti.

Ještě před tím, než se pustíme do psaní testovacího modulu, musíme promyslet, které testy budeme potřebovat. Potřebujeme otestovat tři odlišné druhy datových typů: seznamy, množiny a slovníky. U seznamů je třeba otestovat přidávání a vkládání prvku, mazání prvku a změnu hodnoty prvku. U množin musíme otestovat přidávání a zahazování prvku. A u slovníků musíme otestovat vkládání

prvku, změnu hodnoty prvku a mazání prvku. Musíme též otestovat, že se v případě selhání žádná z provedených změn neaplikuje na původní posloupnost.

Ze strukturálního hlediska je testování různých datových typů stejné, a proto napíšeme testovací případy pouze pro testování seznamů a ostatní necháme do cvičení. Modul `test_Atomic.py` musí importovat modul `unittest` a modul `Atomic`, který má testovat.

Při vytváření souborů pro modul `unittest` obvykle nevytváříme programy, ale moduly a uvnitř každého modulu definujeme jednu či více podtříd třídy `unittest.TestCase`. V případě modulu `test_Atomic.py` definujeme jedinou podtřídu třídy `unittest.TestCase` s názvem `TestAtomic` (na kterou se podíváme vzápětí) a modul zakončujeme následujícími dvěma řádky:

```
if __name__ == "__main__":
    unittest.main()
```

Díky těmto řádkům lze tento modul spustit samostatně. A samozřejmě jej lze též importovat a spustit z jiného testovacího programu, což by dávalo smysl, pokud by se jednalo o jednu z mnoha testovacích sad.

Pokud chceme modul `test_Atomic.py` spustit z jiného testovacího programu, můžeme napsat program, který je podobný tomu, který jsme použili k provedení dokumentačních testů pomocí modulu `unittest`. Například:

```
import unittest
import test_Atomic

suite = unittest.TestLoader().loadTestsFromTestCase(
        test_Atomic.TestAtomic)
runner = unittest.TextTestRunner()
print(runner.run(suite))
```

Zde jsme vytvořili jedinou sadu tím, že jsme modulu `unittest` řekli, aby přečetl modul `test_Atomic` a použil každou z jeho metod ve tvaru `test*()` (v tomto příkladu `test_list_success()` a `test_list_fail()`, jak uvidíme za okamžik) jako testovací případy.

Nyní se podíváme na implementaci třídy `TestAtomic`. Vůbec zde není nutné implementovat inicializační metodu, což je sice pro podtřídy obecně neobvyklé, ale ne tak pro podtřídy třídy `unittest.TestCase`. V tomto případě budeme potřebovat přípravnou metodu, úklidovou metodu však nikoli. Implementovat budeme dva testovací případy.

```
def setUp(self):
    self.original_list = list(range(10))
```

Metodu `unittest.TestCase.setUp()` jsme použili pro vytvoření jediného kousku testovacích dat.

```
def test_list_succeed(self):
    items = self.original_list[:]
    with Atomic.Atomic(items) as atomic:
        atomic.append(1999)
```

```
        atomic.insert(2, -915)
        del atomic[5]
        atomic[4] = -782
        atomic.insert(0, -9)
    self.assertEqual(items, [-9, 0, 1, -915, 2, -782, 5, 6, 7, 8, 9, 1999])
```

Tento testovací případ používáme k otestování, zda se na seznam správně aplikují všechny změny. Test provádí připojení, vložení doprostřed, vložení na začátek, vymazání a změnu hodnoty. Nejedná se sice o vyčerpávající test, pokrývá však základní operace.

Tento test by neměl vyvolat žádnou výjimku, a pokud by ji přece jen vyvolal, zachytí ji bázová třída `unittest.TestCase` a převede na odpovídající chybovou zprávu. Na konci očekáváme, že se prvky seznamu nebudou rovnat původnímu seznamu, ale seznamovému literálu v testu. Metoda `unittest.TestCase.assertEqual()` dokáže porovnat libovolné dva objekty Pythonu, avšak kvůli svému všeobecnému zaměření neumí poskytovat chybové zprávy s podrobnějšími informacemi.

Počínaje Pythonem 3.1 nabízí třída `unittest.TestCase` spoustu dalších metod, mezi něž patří mnoho metod pro aserce specifické pro konkrétní datové typy. V Pythonu 3.1 bychom tedy napsali:

3.1

```
self.assertListEqual(items, [-9, 0, 1, -915, 2, -782, 5, 6, 7, 8, 9, 1999])
```

Pokud se seznamy nerovnají, pak je modul `unittest` schopen nabídnout přesně informace o chybě včetně místa, kde se seznamy liší, což je dáno tím, že datové typy jsou známy.

```
def test_list_fail(self):
    def process():
        nonlocal items
        with Atomic.Atomic(items) as atomic:
            atomic.append(1999)
            atomic.insert(2, -915)
            del atomic[5]
            atomic[4] = -782
            atomic.poop() # Překlep

    items = self.original_list[:]
    self.assertRaises(AttributeError, process)
    self.assertEqual(items, self.original_list)
```

K otestování případu selhání, tj. když při provádění atomických operací dojde k vyvolání výjimky, musíme otestovat, že seznam nebyl změněn a že došlo k vyvolání příslušné výjimky. Pro kontrolu výjimky používáme metodu `unittest.TestCase.assertRaises()`, které v případě Pythonu 3.0 předáváme očekávanou výjimku a volatelný objekt, který by ji měl vyvolat. Tímto způsobem jsme nuceni kód, který chceme otestovat, zapouzdřit, a tudíž musíme vytvořit vnitřní funkci `process()`.

3.1

V Pythonu 3.1 lze metodu `unittest.TestCase.assertRaises()` použít jako správce kontextu, takže náš test pak můžeme napsat mnohem přirozenějším způsobem:

```
def test_list_fail(self):
    items = self.original_list[:]
```

3.1
```
        with self.assertRaises(AttributeError):
            with Atomic.Atomic(items) as atomic:
                atomic.append(1999)
                atomic.insert(2, -915)
                del atomic[5]
                atomic[4] = -782
                atomic.poop() # Překlep
        self.assertListEqual(items, self.original_list)
```

3.1 Zde jsme napsali testovaný kód přímo do testovací metody, aniž bychom museli vytvářet vnitřní funkci, protože jsme metodu `unittest.TestCase.assertRaised()` použili jako správce kontextu, který v tomto případě očekává, že náš kód vyvolá výjimku `AttributeError`. Dále jsme na konci použili metodu `unittest.TestCase.assertListEqual()` z Pythonu 3.1.

Jak jsme viděli, testovací moduly Pythonu se snadno používají a jsou extrémně užitečné, zejména pak tehdy, když používáme vývoj řízený testy. Kromě toho nabízejí mnohem více funkčních prvků a možností, než které jsme si zde ukázali (např. schopnost přeskakovat testy, což je užitečné s ohledem na odlišnosti jednotlivých platforem), a jsou navíc kvalitně zdokumentované. Jedinou funkcí, která zde chybí (a kterou nabízejí moduly `nose` a `py.test`), je objevování testů, ačkoliv tato funkce by se měla objevit v pozdějších verzích Pythonu (snad již v Pythonu 3.2).

Profilování

Pokud program běží velice pomalu nebo spotřebovává mnohem více paměti, než kolik očekáváme, pak je tento problém nejčastěji způsoben naší volbou algoritmu či datových struktur nebo tím, že máme neefektivní implementaci. Ať už je důvod tohoto problému jakýkoliv, nejlepší je nesnažit se prohlížet kód a optimalizovat jej, ale snažit se přesně lokalizovat místo problému. Nahodilá optimalizace může do programu zanést chyby nebo zrychlit ty části programu, které ve skutečnosti nemají na jeho celkový výkon žádný vliv, protože ke zlepšení nedošlo v místech, kde interpret tráví většinu svého času.

Než se pustíme dále do profilování, je dobré uvést několik zvyků programátorů v Pythonu, které se snadno osvojují a aplikují a které mají dobrý vliv na výkon. Žádná z těchto technik není závislá na určité verzi Pythonu a všechny se perfektně hodí ke stylu programování v jazyku Python. Za prvé: když potřebujeme posloupnost určenou pouze pro čtení, pak dávejte přednost n-ticím před seznamy. Za druhé: místo vytváření rozsáhlých n-tic či seznamů, které se mají procházet, používejte generátory. Za třetí: místo vlastní datových struktur implementovaných v Pythonu používejte vestavěné datové struktury (typy `dict`, `list` a `tuple`), které jsou ve všech směrech vysoce optimalizované. Za čtvrté: při vytváření rozsáhlých řetězců tvořených spoustou malých řetězců tyto malé řetězce nespojujte, ale nahromaďte je do seznamu a ten pak spojte do jediného řetězce. Za páté: pokud se k nějakému objektu (včetně funkce či metody) přistupuje mnohokrát pomocí přístupu k atributům (např. při přístupu k funkci v modulu) nebo z datové struktury, pak může být lepší vytvořit a používat lokální proměnnou, která ukazuje na tento objekt a která poskytuje rychlejší přístup.

Standardní knihovna Pythonu nabízí dva moduly, které jsou zvláště užitečné v situaci, kdy potřebujeme prověřit výkon svého kódu. Jedním z nich je modul `timeit`, který je užitečný pro časování malých kousků kódu jazyka Python a který lze použít například pro porovnání výkonu dvou či více implementací určité funkce či metody. Druhým modulem je `cProfile`, který lze použít k profilová-

ní výkonu programu. Poskytuje podrobný rozpis počtů a časů volání, a proto jej můžeme použít pro nalezení úzkého místa výkonu.*

Abychom si vyzkoušeli, jak se modul `timeit` používá, podíváme se na malý příklad. Předpokládejme, že máme funkce `function_a()`, `function_b()` a `function_c()`, které provádějí stejný výpočet, ovšem každá s použitím jiného algoritmu. Pokud všechny tyto funkce umístíme do nějakého modulu (nebo je importujeme), pak je můžeme spustit pomocí modulu `timeit` a zjistit, jak si vůči sobě vedou. Zde je kód, který bychom použili na konci modulu:

```
if __name__ == "__main__":
    repeats = 1000
    for function in ("function_a", "function_b", "function_c"):
        t = timeit.Timer("{0}(X, Y)".format(function),
                         "from __main__ import {0}, X, Y".format(function))
        sec = t.timeit(repeats) / repeats
        print("{function}() {sec:.6f}s".format(**locals()))
```

Prvním argumentem předaným konstruktoru `timeit.Timer()` je kód ve formě řetězce, který chceme provést a u kterého chceme změřit čas. Zde má tento řetězec při prvním průchodu cyklem hodnotu `"function_a(X, Y)"`. Druhý argument je volitelný a opět se jedná o řetězec, který se má provést, tentokráte však jako příprava *před* kódem, jehož čas se má změřit. Zde jsme importovali z modulu `__main__` (tj. z tohoto modulu) funkci, kterou chceme otestovat plus dvě proměnné, které jí předáváme jako vstupní data a které jsou dostupné jako globální proměnné v tomto modulu. Stejně tak bychom mohli importovat funkci a data z jiného modulu.

Při zavolání metody `timeit()` třídy `timeit.Timer` se nejdříve provede druhý argument konstruktoru (pokud byl zadán), který vše připraví, a poté se provede první argument konstruktoru a změří se doba, kterou toto provedení zabralo. Návratová hodnota metody `timeit.Timer.timeit()` představuje zabraný čas v sekundách a má podobu čísla typu `float`. Metoda `timeit()` se standardně opakuje milionkrát a vrací celkový počet sekund pro všechna tato provádění, avšak v tomto případě nám pro užitečné výsledky stačí pouze 1000 opakování, a proto počet opakování zadáváme explicitně. Po změření doby trvání každého volání funkce podělíme celkovou hodnotu počtem opakování a tím získáme střední dobu provádění (průměr), kterou společně s názvem funkce vypíšeme do konzoly.

```
function_a() 0.001618s
function_b() 0.012786s
function_c() 0.003248s
```

V tomto příkladu je funkce `function_a()` jasně nejrychlejší – tedy přinejmenším s použitými vstupními daty. V některých situacích (např. když výkon výrazně kolísá v závislosti na vstupních datech) můžeme otestovat každou funkci s více skupinami vstupních dat, abychom pokryli reprezentativní množinu případů, a poté porovnáme celkový nebo průměrný čas provádění.

* Modul `cProfile` je obvykle dostupný pro interprety CPython, avšak v případě ostatních není vždy k dispozici. Všechny knihovny Pythonu by měly mít modul `profile` napsaný čistě v Pythonu, který poskytuje stejné rozhraní API jako modul `cProfile` a provádí stejnou činnost, jen o něco pomaleji.

Ne vždy je vhodné instrumentovat kód pro získání údajů o jeho časování, a proto modul `timeit` nabízí způsob měření času kódu i z příkazového řádku. Kupříkladu pro změření doby provádění funkce `function_a()` z modulu `MyModule.py` zadáme do konzoly následující: `python3 -m timeit -n 1000 -s "from MyModule import function_a, X, Y" "function_a(X, Y)"`. (Jako obvykle ve Windows musíme nahradit název `python3` celou cestou, například `C:\Python31\python.exe`.) Volba `-m` je pro interpret Pythonu a říká mu, aby načetl zadaný modul (v tomto případě `timeit`), přičemž ostatní volby již zpracuje modul `timeit`. Volba `-n` stanoví počet opakování, volba `-s` určuje přípravný kód a posledním argumentem je kód, jehož doba provádění se má změřit. Jakmile tento příkaz dokončí svoji činnost, vypíše výsledky do konzoly, například:

```
1000 loops, best of 3: 1.41 msec per loop
```

Stejným způsobem bychom nyní mohli opakovat měření času i pro další dvě funkce a výsledky potom porovnat.

Modul `cProfile` (nebo modul `profile` – na oba se budeme odkazovat jako na modul `cProfile`) můžeme použít také k porovnání výkonu funkcí či metod. Na rozdíl od modulu `timeit`, který poskytuje jen holé měření času, ukazuje modul `cProfile` přesně, co se volá a jak dlouho každé volání trvá. Zde je kód, který bychom použili pro porovnání stejných tří funkcí jako výše:

```
if __name__ == "__main__":
    for function in ("function_a", "function_b", "function_c"):
        cProfile.run("for i in range(1000): {0}(X, Y)".format(function))
```

Počet opakování musíme umístit do kódu, který předáváme funkci `cProfile. run()`, nemusíme ale provádět žádnou přípravu, protože tato funkce používá introspekci pro nalezení funkcí a proměnných, které chceme použít. Nemáme zde žádný explicitní příkaz `print()`, protože funkce `cProfile. run()` standardně vypisuje svůj výstup do konzoly. Zde jsou výsledky pro všechny funkce (několik irelevantních řádků jsme vynechali a výstup jsme malinko upravili, aby pasoval na stránku):

```
1003 function calls in 1.661 CPU seconds
 ncalls tottime percall cumtime percall filename:lineno(function)
      1   0.003   0.003   1.661   1.661 <string>:1(<module>)
   1000   1.658   0.002   1.658   0.002 MyModule.py:21(function_a)
      1   0.000   0.000   1.661   1.661 {built-in method exec}

5132003 function calls in 22.700 CPU seconds
 ncalls tottime percall cumtime percall filename:lineno(function)
      1   0.487   0.487  22.700  22.700 <string>:1(<module>)
   1000   0.011   0.000  22.213   0.022 MyModule.py:28(function_b)
5128000   7.048   0.000   7.048   0.000 MyModule.py:29(<genexpr>)
   1000   0.005   0.000   0.005   0.000 {built-in method bisect_left}
      1   0.000   0.000  22.700  22.700 {built-in method exec}
   1000   0.001   0.000   0.001   0.000 {built-in method len}
   1000  15.149   0.015  22.196   0.022 {built-in method sorted}

5129003 function calls in 12.987 CPU seconds
 ncalls tottime percall cumtime percall filename:lineno(function)
```

```
      1   0.205   0.205  12.987  12.987 <string>:1(<module>)
   1000   6.472   0.006  12.782   0.013 MyModule.py:36(function_c)
5128000   6.311   0.000   6.311   0.000 MyModule.py:37(<genexpr>)
      1   0.000   0.000  12.987  12.987 {built-in method exec}
```

Sloupec `ncalls` („number of calls“) uvádí počet volání určité funkce (uvedené ve sloupci `filename:lineno(function)`). Vzpomeňte si, že volání opakujeme 1000krát, což musíme mít stále na paměti. Sloupec `tottime` („total time“) uvádí celkový čas strávený ve funkci, avšak bez času stráveného uvnitř funkcí volaných z této funkce. První sloupec `percall` obsahuje průměrný čas každého volání dané funkce (`tottime // ncalls`). Sloupec `cumtime` („cumulative time“) uvádí čas strávený ve funkci a zahrnuje též čas strávený uvnitř funkcí volaných z této funkce. Druhý sloupec `percall` obsahuje průměrný čas každého volání funkce včetně funkcí, které z ní voláme.

Tento výstup poskytuje mnohem více informací než holé měření časů modulem `timeit`. Okamžitě tak vidíme, že obě funkce `function_b()` a `function_c()` používají generátory, které se volají více než 5000krát, což je činí minimálně 10krát pomalejší ve srovnání s funkcí `function_a()`. Ba co víc, funkce `function_b()` volá více funkcí, mezi něž patří i volání vestavěné funkce `sorted()`, kvůli čemuž je přinejmenším dvakrát pomalejší než funkce `function_c()`. Samozřejmě také modul `timeit` nabízí dostatečné informace pro zjištění těchto rozdílů, avšak díky modulu `cProfile` si můžeme podrobně prohlédnout příčiny těchto rozdílů.

Modul `cProfile` nám umožňuje stejně jako modul `timeit` měřit čas kódu bez jeho instrumentace. Nicméně při použití modulu `cProfile` z příkazového řádku nemůžeme přesně stanovit, co se má provést. V tomto případě se prostě provede zadaný program či modul a vypíší se výsledky měření času všeho, co obsahuje. Na příkazový řádek zapíšeme `python3 -m cProfile programNeboModul.py` a obdržíme výstup ve stejném formátu jako výše. Zde je jeho trošku upravená část, z níž jsme vypustili většinu řádků:

```
10272458 function calls (10272457 primitive calls) in 37.718 CPU secs
  ncalls tottime percall cumtime percall filename:lineno(function)
       1   0.000   0.000  37.718  37.718 <string>:1(<module>)
       1   0.719   0.719  37.717  37.717 <string>:12(<module>)
    1000   1.569   0.002   1.569   0.002 <string>:20(function_a)
    1000   0.011   0.000  22.560   0.023 <string>:27(function_b)
 5128000   7.078   0.000   7.078   0.000 <string>:28(<genexpr>)
    1000   6.510   0.007  12.825   0.013 <string>:35(function_c)
 5128000   6.316   0.000   6.316   0.000 <string>:36(<genexpr>)
```

V terminologii modulu `cProfile` je *primitivní* volání nerekurzivní volání funkce.

Použití modulu `cProfile` tímto způsobem může být užitečné pro identifikování oblastí, které stojí za další prozkoumání. Například zde můžeme jasně vidět, že funkce `function_b()` trvá dlouhou dobu. Jak se ale propracovat k podrobnostem? Mohli bychom instrumentovat program nahrazením volání funkce `function_b()` příkazem `cProfile.run("function_b()")`. Nebo bychom mohli uložit kompletní profilová data a analyzovat je pomocí modulu `psats`. Pro uložení profilu musíme upravit volání na příkazovém řádku: `python3 -m cProfile -o datovySouborSProfilem programNeboModul.py`. Nyní můžeme analyzovat data profilu, což lze provést například tak, že spustíme editor IDLE, importujeme modul `pstats` a předáme mu uložené `datovýSouborSProfilem` nebo použijeme modul `pstats`

interaktivně na příkazovém řádku. Zde je kratičký příklad konzolové relace, kterou jsme malinko poupravili pro účely tisku a ve které je zvýrazněn náš vstup:

```
$ python3 -m cProfile -o profile.dat MyModule.py
$ python3 -m pstats
Welcome to the profile statistics browser.
% read profile.dat
profile.dat% callers function_b
   Random listing order was used
   List reduced from 44 to 1 due to restriction <'function_b'>
Function was called by...
                        ncalls tottime cumtime
<string>:27(function_b) <- 1000   0.011  22.251 <string>:12(<module>)

profile.dat% callees function_b
   Random listing order was used
   List reduced from 44 to 1 due to restriction <'function_b'>
Function called...
                   ncalls tottime cumtime
<string>:27(function_b) ->
                     1000   0.005   0.005 built-in method bisect_left
                     1000   0.001   0.001 built-in method len
                     1000  15.297  22.234 built-in method sorted
profile.dat% quit
```

K získání seznamu příkazů napište `help` a pro informace o určitém příkazu napište `help` *`příkaz`*. Například `help stats` vypíše, jaké argumenty lze předat příkazu `stats`. K dispozici jsou také nástroje, které dokážou vytvořit grafickou vizualizaci profilových dat. Jedná se například o nástroj RunSnakeRun (*www.vrplumber.com/programming/runsnakerun*), který využívá knihovnu GUI wxPython.

Pomocí modul `timeit` a `cProfile` můžeme identifikovat oblasti svého kódu, které mohou trvat nad očekávání dlouho, a s použitím modulu `cProfile` můžeme přesně zjistit, které části spotřebovávají nejvíce času.

Shrnutí

Obecně lze říci, že hlášení syntaktických chyb v Pythonu je velice přesné a zahrnuje správně identifikovaný řádek a pozici na tomto řádku. Jediný případ, kdy toto dobře nefunguje, nastává tehdy, když zapomeneme uzavřít závorku (kulatou, hranatou či složenou), přičemž je chyba obvykle hlášena až na následujícím neprázdném řádku. Syntaktické chyby lze naštěstí téměř vždy snadno odhalit a opravit.

Dojde-li k vyvolání neobsloužené výjimky, pak Python ukončí program a vypíše zásobník volání. Tyto výpisy mohou být pro koncové uživatele odstrašující, pro nás programátory však poskytují užitečné informace. V ideálním případě bychom měli vždy obsloužit každý typ výjimky, o které se domníváme, že se v našem programu může vyskytnout, a v případě nutnosti prezentovat vzniklý problém uživateli ve formě chybové zprávy, okna se zprávou nebo zaprotokolované zprávy – ne ale jako holý

výpis zásobníku volání. Nicméně bychom neměli používat obsluhu `except:` pro zachycení všech výjimek. Chceme-li zachytit všechny výjimky (např. na nejvyšší úrovni programu), můžeme použít `except Exception as err` a chybu `err` vždy hlásit, protože tiché obsluhování výjimek může vést k tomu, že náš program bude později selhávat zákeřným a nepozorovaným způsobem (např. kvůli narušeným datům). A během vývoje je pravděpodobně nejlepší žádnou obsluhu výjimky na nejvyšší úrovni vůbec nemít a nechat program prostě havarovat s výpisem zásobníku volání.

Ladění nemusí (a nemělo by) být záležitostí nahodilého přístupu. Zúžením vstupu nezbytného k reprodukování chyby na nepatrné minimum, uvážlivým vyslovováním hypotéz o problému a následným testováním těchto hypotéz (pomocí příkazů `print()` nebo ladicího nástroje) můžeme často lokalizovat zdroj chyby mnohem rychleji. A pokud nás naše hypotézy úspěšně zavedou k chybě, pak nám s největší pravděpodobností pomohou také s jejím řešením.

Pro účely testování nabízejí moduly `doctest` a `unittest` své vlastní specifické přednosti. Dokumentační testy bývají pohodlné a užitečné u malých knihoven a modulů, protože dobře zvolené testy mohou snadno odhalit a otestovat hraniční i běžné případy, přičemž samotné psaní testů je samozřejmě pohodlné a snadné. Na druhou stranu testy jednotek nejsou omezeny jen na zápis uvnitř dokumentačních řetězců, ale zapisují se též jako samostatné moduly, což je obvykle lepší volba v situaci, kdy máme napsat komplexnější a sofistikovanější testy, což se týká zvláště testů, které vyžadují přípravu a úklid. U rozsáhlých projektů můžeme pomocí modulu `unittest` (nebo nějakého modulu třetí strany pro testování jednotek) udržovat testy a testované programy a moduly odděleně, což je ve srovnání s dokumentačními testy obecně flexibilnější a výkonnější.

Pokud narazíme na problémy s výkonem, pak je příčina nejčastěji v našem vlastním kódu, především v naší volbě algoritmů a datových struktur nebo v jejich neefektivní implementaci. Když čelíme takovýmto problémům, pak je vždy rozumné nehádat, ale přesně zjistit, kde se nachází úzké místo výkonu, protože jinak strávíme čas optimalizací něčeho, co ve skutečnosti celkový výkon nijak nezlepší. Modul Pythonu `timeit` můžeme použít k získání holého měření času volání funkcí nebo libovolných úryvků kódu, je tedy zvláště užitečný pro porovnání alternativních implementací funkce. A pro hloubkovou analýzu máme k dispozici modul `cProfile`, který nabízí informace o měření času i o počtu volání, takže můžeme identifikovat nejen to, které funkce spotřebovávají nejvíce času, ale také to, které funkce následně volají.

Celkově lze říci, že Python nabízí skvělou podporu pro ladění, testování a profilování, která je ihned připravena k použití. Nicméně zvláště u rozsáhlých projektů stojí za uváženou některé z testovacích nástrojů třetích stran, které mohou ve srovnání s testovacími moduly standardní knihovny nabízet více funkcí a pohodlnější používání.

LEKCE 10

Procesy a vlákna

V této lekci:

- Modul pro práci s více procesy
- Modul pro práci s vlákny

S nástupem vícejádrových procesorů, které se již staly běžnou součástí většiny počítačů, je nyní mnohem svůdnější a praktičtější rozložení zátěže takovým způsobem, aby se maximálně využila všechna dostupná jádra. Existují dva hlavní způsoby, jak rozkládat zátěž. Jedním je použít více procesů a druhým více vláken. V této lekci si ukážeme, jak v praxi použít oba způsoby.

Použití více procesů, a tedy spouštění samostatných programů, má tu výhodu, že každý proces běží nezávisle na ostatních. Díky tomu jsou veškeré obtíže spojené s obsluhou souběžného přístupu ponechány na operačním systému. Nevýhodou ovšem je, že komunikace a sdílení dat mezi zahajovacím programem a samostatnými procesy, které zahájil, může být nepohodlná. Na unixových systémech to lze vyřešit pomocí příkazů operačního systému `exec` a `fork`, avšak u programů běžících na více platformách je třeba sáhnout po jiném řešení. Nejjednodušší řešení, a také jediné, které si zde ukážeme, spočívá v tom, že zahajovací program naplní daty procesy, které spouští, a nechá je, aby své výstupy vytvořily nezávislé na sobě. Flexibilnější přístup, který značně zjednodušuje obousměrnou komunikaci, tkví ve využití sítě. Samozřejmě v řadě situací tato komunikace není nutná a stačí nám, když spustíme jeden či více dalších programů z jednoho instrumentačního programu.

Další možností, jak předávat práci nezávislým procesům, je vytvořit vícevláknový program, který rozděluje práci nezávislým prováděcím vláknům. Výhodou tohoto přístupu je, že můžeme komunikovat prostým sdílením dat (ovšem za předpokladu, že zajistíme, aby ke sdíleným datům přistupovalo v jednom okamžiku pouze jediné vlákno), čímž ale zůstávají veškeré obtíže související se správou souběžného přístupu na programátorovi. Python nabízí kvalitní podporu pro vytváření vícevláknových programů, čímž minimalizuje práci, která zůstává na nás. Nicméně vícevláknové programy jsou podstatně složitější než jednovláknové programy a při své tvorbě a údržbě vyžadují mnohem více péče.

V první části této lekce vytvoříme dva malé programy. První spouští uživatel a druhý se pro každý samostatný proces, který je vyžadován, spouští z prvního programu. Ve druhé části začneme seznámením se základními fakty o programování s vlákny. Poté vytvoříme vícevláknový program, který má stejnou funkčnost jako oba programy z první části spojené dohromady, abychom si ukázali rozdíl mezi programováním s více procesy a s více vlákny. A potom se podíváme na další, složitější vícevláknový program, který rozděluje práci a shromažďuje všechny výsledky.

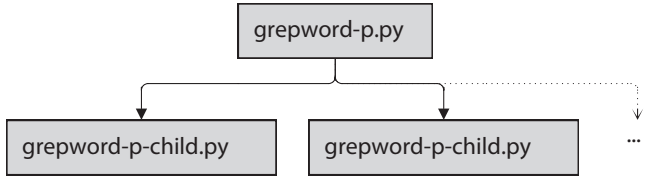
Modul pro práci s více procesy

V některých situacích již programy, které poskytují potřebnou funkčnost, máme, avšak potřebujeme automatizovat jejich použití. To můžeme provést pomocí modulu Pythonu s názvem `subprocess`, který poskytuje prostředky pro spouštění dalších programů, předávání případných voleb příkazového řádku, a je-li to požadováno, pak také pro komunikaci s nimi pomocí kanálů. S jednoduchým příkladem jsme se již setkali v lekci 5, kde jsme pomocí funkce `subprocess.call()` vymazali konzolu způsobem specifickým pro určitou platformu. Tyto prostředky ovšem můžeme využít také ke tvorbě dvojic programů ve stylu „nadřízený-podřízený“, kde uživatel spouští nadřízený program a ten následně dle potřeby spouští mnoho instancí podřízených programů, z nichž každá provádí jinou činnost. A právě tomuto přístupu se budeme v této části věnovat.

V lekci 3 jsme si ukázali velice jednoduchý program `grepword.py`, který hledá slovo zadané na příkazovém řádku v souborech uvedených za tímto slovem. V této části vyvineme jeho poněkud sofistikovanější verzi, která dokáže při hledání souborů rekurzivně sestupovat do podadresářů a která

navíc dokáže předávat práci tolika samostatným podřízeným procesům, kolik jen chceme. Výstupem je prostý seznam souborů (s cestami), které obsahují zadané slovo.

Nadřízeným programem je `grepword-p.py` a podřízeným `grepword-pchild.py`. Vztah mezi těmito dvěma spuštěnými programy je schematicky znázorněn na obrázku 10.1.



Obrázek 10.1: Nadřízený program a programy podřízené

Srdce programu `grepword-p.py` je zapouzdřené do funkce `main()`, kterou si rozdělíme na tři části:

```
def main():
    child = os.path.join(os.path.dirname(__file__),
                         "grepword-p-child.py")
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    files_per_process = len(filelist) // opts.count
    start, end = 0, files_per_process + (len(filelist) % opts.count)
    number = 1
```

Začínáme získáním názvu podřízeného programu. Potom vezmeme volby uživatele na příkazovém řádku. Funkce `parse_options()` využívá modul `optparse` a vrací pojmenovanou n-tici `opts`, která obsahuje informace o tom, zda má program rekurzivně sestupovat do podadresářů a kolik procesů má použít (výchozí hodnota je 7) – pro náš program jsme nahodile zvolili maximálně 20 procesů. Dále vrací slovo, které se má hledat, a seznam názvů (názvů souborů a adresářů) zadaných na příkazovém řádku. Funkce `get_files()` vrací seznam souborů, které se budou číst.

`get_files()` ➤ 333

Jakmile máme informace nezbytné k provedení úkolu, můžeme vypočítat, kolik souborů máme přidělit každému procesu. Proměnné `start` a `end` používáme pro stanovení řezu v seznamu `filelist`, který bude předán následujícímu podřízenému procesu na zpracování. Počet souborů většinou nebude přesným násobkem počtu procesů, a proto jejich počet předávaný prvnímu procesu zvýšíme o zbytek. Proměnnou `number` používáme čistě pro ladicí účely, abychom měli přehled, který proces vytvořil jednotlivé řádky výstupu.

```
    pipes = []
    while start < len(filelist):
        command = [sys.executable, child]
        if opts.debug:
            command.append(str(number))
        pipe = subprocess.Popen(command, stdin=subprocess.PIPE)
        pipes.append(pipe)
```

```
        pipe.stdin.write(word.encode("utf8") + b"\n")
        for filename in filelist[start:end]:
            pipe.stdin.write(filename.encode("utf8") + b"\n")
        pipe.stdin.close()
        number += 1
        start, end = end, end + files_per_process
```

Pro každý řez `start:end` seznamu `filelist` vytváříme seznam `command` skládající se z interpretu jazyka Python (dostupného v proměnné `sys.executable`), podřízeného programu, který má Python provést, a voleb příkazového řádku – v tomto případě jen číslo potomka, probíhá-li ladění. Má-li podřízený program vhodný řádek shebang nebo souborovou asociaci, pak bychom jej mohli uvést jako první a nezatěžovat se s interpretem jazyka Python. My však dáváme přednost tomuto postupu, protože nám zajišťuje, že podřízený program používá stejný interpret jazyka Python jako náš nadřízený program.

Jakmile máme povel připravený, vytvoříme objekt typu `subprocess.Popen` a předáme připravený povel k provedení (jako seznam řetězců), přičemž v tomto případě si vyžádáme zápis do standardního vstupu procesu. (Nastavením podobného klíčovaného argumentu je také možné číst ze standardního výstupu procesu.) Poté zapíšeme hledané slovo, nový řádek a pak každý soubor v příslušném řezu seznamu souborů. Modul `subprocess` čte a zapisuje bajty, ne řetězce, ovšem procesy, které vytváří, vždy předpokládají, že bajty přijímané ze vstupu `sys.stdin` jsou řetězce v místním kódování, a to i tehdy, pokud by námi odeslané bajty používaly odlišné kódování, jako je například UTF-8, které jsme použili v tomto příkladu. Za okamžik si ukážeme, jak tento nepříjemný problém obejít. Jakmile slovo a seznam souborů zapíšeme do podřízeného procesu, uzavřeme jeho standardní vstup a přesuneme se dále.

Není naprosto nezbytné uchovávat odkaz na každý proces (proměnná `pipe` se při každém průchodu cyklem opětovně svazuje s novým objektem typu `subprocess.Popen`), protože každý proces běží nezávisle. My ale jednotlivé procesy přidáváme do seznamu, abychom je mohli v případě potřeby přerušit. Dále neshromažďujeme výsledky, ale místo toho necháme každý proces zapsat své výsledky do konzoly v čase, který si sám určí. To znamená, že výstupy z různých procesů se mohou navzájem prolínat. (Prolínání budete moci odstranit ve cvičení.)

```
    while pipes:
        pipe = pipes.pop()
        pipe.wait()
```

Jakmile máme všechny procesy spuštěné, čekáme, až se každý podřízený proces dokončí. To není nezbytně nutné, ale na unixových systémech takto zajistíme, že se na příkazový řádek konzoly vrátíme až v okamžiku, kdy budou hotové všechny procesy (v opačném případě bychom museli po jejich dokončení stisknout klávesu Enter). Další výhodou čekání je, že pokud dojde k přerušení programu (např. stiskem Ctrl+C), pak se všechny aktuálně běžící procesy přeruší a ukončí s nezachycenou výjimkou `KeyboardInterrupt`. Pokud bychom nečekali, hlavní program by skončil (a tudíž by nešel přerušit) a podřízené procesy by pokračovaly dále (dokud by je nezabil povel `kill` nebo Správce úloh).

Zde je kompletní program `grepword-p-child.py` bez komentářů a příkazů `import`. Podíváme se na něj ve dvou částech, přičemž první část bude mít dvě verze, jednu pro Python 3.x a druhou pro Python 3.1 a novější verze:

3.x

```
BLOCK_SIZE = 8000

number = "{0}: ".format(sys.argv[1]) if len(sys.argv) == 2 else ""
stdin = sys.stdin.buffer.read()
lines = stdin.decode("utf8", "ignore").splitlines()
word = lines[0].rstrip()
```

3.0

Program začíná nastavením řetězce `number` na zadané číslo nebo prázdný řetězec, pokud neprobíhá ladění. Tento program běží jako podřízený proces, přičemž modul `subprocess` pouze čte a zapisuje binární data a vždy používá místní kódování, a proto musíme sami přečíst paměť s binárními daty ze vstupu `sys.stdin` a provést dekódování.* Jakmile binární data načteme, dekódujeme je do řetězce ve znakové sadě Unicode a rozdělíme na řádky. Podřízený proces poté přečte první řádek, který obsahuje hledané slovo.

Zde jsou řádky, které se v případě Pythonu 3.1 liší:

3.1

```
sys.stdin = sys.stdin.detach()
stdin = sys.stdin.read()
lines = stdin.decode("utf8", "ignore").splitlines()
```

Python 3.1 nabízí metodu `sys.stdin.detach()`, která vrací objekt představující binární soubor. Poté přečteme všechna data, dekódujeme je do znakové sady Unicode s kódováním dle našeho výběru a potom řetězec ve znakové sadě Unicode rozdělíme na řádky.

```
for filename in lines[1:]:
    filename = filename.rstrip()
    previous = ""
    try:
        with open(filename, "rb") as fh:
            while True:
                current = fh.read(BLOCK_SIZE)
                if not current:
                    break
                current = current.decode("utf8", "ignore")
                if (word in current or
                    word in previous[-len(word):] +
                            current[:len(word)]):
                    print("{0}{1}".format(number, filename))
                    break
                if len(current) != BLOCK_SIZE:
```

* Je možné, že budoucí verze Pythonu budou obsahovat verzi modulu `subprocess`, který bude obsahovat argumenty `encoding` a `errors`, jež nám umožní použít námi preferované kódování bez toho, abychom sami museli přistupovat ke vstupu `sys.stdin` v binárním režimu a provádět dekódování. Viz *bugs.python.org/issue6135*.

```
                    break
                previous = current
    except EnvironmentError as err:
        print("{0}{1}".format(number, err))
```

Všechny řádky po prvním řádku jsou názvy souborů (včetně cest). Pro každý otevřeme příslušný soubor, načteme jej a vypíšeme jeho název, obsahuje-li hledané slovo. Některé soubory mohou být velmi velké, což by mohl být problém, zejména pak v situaci, kdy všech 20 podřízených procesů běží současně a všechny čtou velké soubory. Tento problém vyřešíme tak, že každý soubor čteme po blocích, přičemž si ukládáme předchozí blok, abychom měli jistotu, že nám neuniknou případy, kdy se hledané slovo vyskytuje na rozmezí dvou bloků. Další výhodou čtení po blocích je, že pokud se hledané slovo objeví blíže k začátku souboru, pak můžeme s daným souborem skončit, aniž bychom jej museli načíst celý, protože nás nezajímá, kde se hledané slovo objevuje, ale jen to, zda se vůbec v daném souboru vyskytuje.

Kódování znaků ➢ 95

Soubory čteme v binárním režimu, takže musíme každý blok před prohledáním převést na řetězec, poněvadž hledané slovo je řetězec. Předpokládáme, že všechny soubory používají kódování UTF-8, což je ale ve většině případů nesprávný předpoklad. Sofistikovanější program by se mohl pokusit zjistit skutečné kódování a poté soubor uzavřít a znovu otevřít pomocí správného kódování. Jak jsme si řekli již v lekci 2, na stránce se seznamem balíčku pro jazyk Python (viz *pypi.python.org/pypi*) máme k dispozici přinejmenším dva balíčky pro automatickou detekci kódování souboru. (Může být svůdné dekódovat hledané slovo do objektu typu `bytes` a porovnávat bajty s bajty, tento postup ale není spolehlivý, protože některé znaky mají v kódování UTF-8 více než jednu platnou reprezentaci.)

Modul `subprocess` nabízí mnohem více funkcí, než které jsme zde potřebovali, včetně ekvivalentních prvků pro zpětné uvozovky (substituce) a kanály shellu a funkcí `os.system()` a `os.spawn*()`.

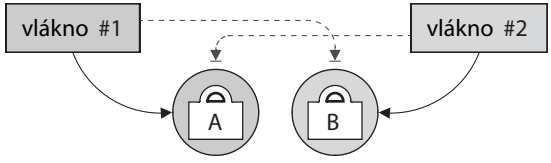
V následující části si ukážeme vláknovou verzi programu `grepword-p.py`, kterou porovnáme s verzí využívající nadřízený proces a podřízené procesy. Dále se podíváme na sofistikovanější vícevláknový program, který rozdělí práci a poté shromáždí výsledky, díky čemuž má větší kontrolu nad způsobem jejich zobrazení.

Modul pro práci s vlákny

Příprava dvou či více samostatných prováděcích vláken je v Pythonu docela přímočarou záležitostí. Složitější to začne být až ve chvíli, kdy chceme, aby samostatná vlákna sdílela data. Představte si, že máme dvě vlákna sdílející nějaký seznam. Jedno vlákno může začít procházet seznam pomocí příkazu `for x in L` a poté někde uprostřed druhé vlákno vymaže některé prvky seznamu. Tato situace povede v nejlepším případě k nevysvětlitelnému pádu a v nejhorším případě k nesprávným výsledkům.

Běžným řešením je použít nějaký druh zámků. Jedno vlákno si tak například může vyžádat zámek a *poté* začít procházet seznam, přičemž druhé vlákno bude blokováno tímto zámkem. Ve skutečnosti věci nejsou tak jasné jako tento příklad. Vztah mezi zámkem a uzamknutými daty existuje jen v naší představivosti. Pokud si jedno vlákno přivlastní zámek a druhé se pokusí o získání stejného zámku, pak druhé vlákno bude blokováno, dokud první zámek neuvolní. Umístěním přístupu ke sdíleným datům do oblasti působnosti získaného zámku můžeme zajistit, že ke sdíleným datům přistupuje v jednom okamžiku pouze jedno vlákno, přestože je tato ochrana nepřímá.

Se zamykáním souvisí jeden problém: riziko uváznutí. Představte si, že *vlákno #1* získá zámek A, takže může přistupovat ke sdíleným datům `a`, a poté se v rámci zámku A pokusí získat zámek B, aby mohlo přistupovat ke sdíleným datům `b`. Zámek B však nezíská, protože *vlákno #2* získalo zámek B, aby mohlo přistupovat k `b`, a samo se nyní pokouší získat zámek A, aby mohlo přistupovat k `a`. Takže *vlákno #1* drží zámek A a pokouší se získat zámek B, zatímco *vlákno #2* drží zámek B a pokouší se získat zámek A. Výsledkem je, že obě vlákna jsou blokována, takže program je ve stavu uváznutí, což zachycuje obrázek 10.2.



Obrázek 10.2: Uváznutí: dvě či více blokovaných vláken se pokouší získat zámky ostatních

Přestože toto konkrétní uváznutí je snadné si vyobrazit, v praxi mohou být uváznutí obtížně odhalitelná, protože ne vždy jsou takto zřejmá. Některé knihovny pro práci s vlákny jsou schopné pomáhat varováním ohledně potenciálních uváznutí, pro jejich předcházení je ale i tak třeba lidská péče a pozornost.

Jednoduchý, nicméně efektivní způsob, jak se vyhnout uváznutím, spočívá v existenci nějaké zásady, která definuje pořadí, v němž by se měly zámky získávat. Pokud bychom například měli zásadu, že zámek A musí být vždy získán před zámkem B, a chtěli bychom získat zámek B, pak bychom podle této zásady museli nejdříve získat zámek A. To by zajistilo, že k výše popsanému uváznutí by nedošlo (protože obě vlákna by se pokoušela získat zámek A a první, kterému by se to podařilo, by pokračovalo k zámku B), dokud by někdo na tuto zásadu nezapomněl.

Další problém se zamykáním tkví v tom, že pokud na získání zámku čeká více vláken, pak jsou tato vlákna blokována a neprovádějí žádnou užitečnou práci. To lze částečně zmírnit drobnými změnami ve stylu programování s cílem minimalizovat množství práce provedené v rámci kontextu zámku.

Každý program Pythonu má nejméně jedno hlavní vlákno. Pro vytvoření dalších vláken musíme importovat modul `threading` a pomocí něho vytvořit tolik vláken, kolik chceme. Vlákna můžeme vytvářet dvěma způsoby. Můžeme zavolat funkci `threading.Thread()` a předat jí volatelný objekt nebo můžeme vytvořit podtřídu třídy `threading.Thread`. V této lekci si ukážeme oba přístupy. Vytvoření podtřídy je nejflexibilnější a docela přímočarý přístup. Podtřídy mohou reimplementovat metodu `__init__()` (musejí pak ale zavolat implementaci bázové třídy) a musejí reimplementovat metodu `run()` – jedná se totiž o metodu, v níž vlákno provádí svoji činnost. Metodu `run()` *nesmíme* nikdy zavolat ze svého kódu – vlákna se spouštějí zavoláním metody `start()`, která pak sama zavolá metodu `run()`. Žádné další metody třídy `threading.Thread` nemůžeme reimplementovat, nové metody však přidávat můžeme.

Příklad: Vícevláknový program pro hledání slova

V tomto oddílu se podíváme na kód programu `grepword-t.py`. Tento program provádí stejnou činnost jako program `grepword-p.py`, ovšem s tím, že práci nepřiděluje procesům, ale vláknům. Schematicky to znázorňuje obrázek 10.3.

Obrázek 10.3: Vícevláknový program

Obzvláště zajímavé na tomto programu je, že to vůbec nevypadá, že by používal nějaké zámky. To je dáno tím, že jedinými sdílenými daty je seznam souborů, pro které používáme třídu `queue.Queue`. Její zvláštnost spočívá v tom, že se o veškeré zamykání stará interně sama, takže při každém přidání či odebrání prvku se můžeme spolehnout na samotnou frontu, že tyto přístupy *serializuje*. V kontextu vláken znamená serializace přístupu k datům zajištění, aby v jednom okamžiku mělo přístup k datům pouze jediné vlákno. Další výhodou použití třídy `queue.Queue` je, že nemusíme sami rozdělovat práci. Stačí přidat pracovní prvky do fronty a nechat pracovní vlákna, aby si v momentě, kdy jsou připravena, sama vyzvedla další práci.

Třída `queue.Queue` funguje na bázi fronty typu FIFO (First In, First Out – první dovnitř, první ven). Modul `queue` nabízí také třídu `queue.LifoQueue` představující frontu typu LIFO (Last In, First Out – poslední dovnitř, první ven) a třídu `queue.PriorityQueue`, která přijímá n-tice, jako je například dvojice (priorita, prvek), přičemž prvky s nejnižším číslem priority se zpracují jako první. Všechny typy front lze vytvářet s uvedením maximální velikosti. Je-li této maximální velikosti dosaženo, pak bude fronta blokovat pokusy o přidání prvků, dokud se nějaké prvky neodstraní.

Program `grepword-t.py` si rozdělíme na tři části a začneme celou funkcí `main()`:

```
def main():
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    work_queue = queue.Queue()
    for i in range(opts.count):
        number = "{0}: ".format(i + 1) if opts.debug else ""
        worker = Worker(work_queue, word, number)
        worker.daemon = True
        worker.start()
    for filename in filelist:
        work_queue.put(filename)
    work_queue.join()
```

Způsob získání voleb a seznamu souborů od uživatele je stejný jako dříve. Jakmile máme nezbytné informace, vytvoříme objekt typu `queue.Queue` a poté procházíme cyklem tolikrát, kolik má být vytvářeno vláken (výchozí počet vláken je 7). Pro každé vlákno připravíme řetězec `number` pro účely ladění (prázdný řetězec, pokud ladění neprobíhá) a pak vytvoříme instanci třídy `Worker` (podtřída třídy `threading.Thread`) – k nastavení vlastnosti `daemon` se vrátíme za chvíli. V dalším kroku vlákno spustíme, třebaže v tomto okamžiku ještě nemá žádnou práci, protože pracovní fronta (`work_queue`) je prázdná. Tím se vlákno okamžitě zablokuje a čeká na práci.

Po vytvoření všech vláken a jejich připravení na práci procházíme všechny soubory a přidáváme je do pracovní fronty. Ihned po přidání prvního souboru do fronty si jej může převzít jedno z vláken a začít na něm pracovat. Tímto způsobem se pokračuje dál, dokud všechna vlákna nemají přidělený soubor. Jakmile vlákno dokončí práci na souboru, může si vzít další, dokud se všechny soubory nezpracují.

Všimněte si, že tento postup je jiný než v případě programu `grepword-p.py`, kde jsme museli každému podřízenému procesu přidělovat řezy seznamu souborů a kde se podřízené procesy spouštěly a dostávaly své seznamy sekvenčně. Použití vláken je v takovýchto případech potenciálně efektivnější. Pokud by například velikost prvních pěti souborů byla velká a velikost zbývajících souborů malá, pak by každý velký soubor zpracovalo jiné vlákno, protože každé vlákno si pokaždé bere jen jednu činnost, čímž se celá práce pěkně rozdělí. Ovšem při použití procesů jako v programu `grepword-p.py` by se všechny velké soubory přidělily prvnímu procesu a malé soubory ostatním, takže první proces by prováděl většinu práce, zatímco ostatní by rychle skončily, aniž by odvedly významnější množství práci.

Program neskončí, dokud některé z vláken běží. To představuje problém, protože jakmile pracovní vlákna odvedou svoji práci, pak z technického hlediska stále běží, třebaže svoji činnost již dokončila. Řešením je převést vlákna na démony. Výsledkem pak je, že program skončí ihned, jakmile v něm již neběží žádné vlákno, které není démonem. Hlavní vlákno není démon, takže jakmile svoji činnost dokončí, program čistě ukončí každé vlákno ve formě démona a poté ukončí i sám sebe. Tím ovšem může vzniknout opačný problém – jakmile jsou vlákna vytvořena a běží, pak musíme zajistit, aby hlavní vlákno neskončilo dříve, než bude dokončena veškerá práce. Toho docílíme zavoláním metody `queue.Queue.join()`, která blokuje, dokud se fronta nevyprázdní.

Zde je začátek třídy `Worker`:

```
class Worker(threading.Thread):

    def __init__(self, work_queue, word, number):
        super().__init__()
        self.work_queue = work_queue
        self.word = word
        self.number = number

    def run(self):
        while True:
            try:
                filename = self.work_queue.get()
                self.process(filename)
            finally:
                self.work_queue.task_done()
```

Metoda `__init__()` musí zavolat metodu `__init__()` bázové třídy. Pracovní fronta je stejným objektem typu `queue.Queue` sdíleným všemi vlákny.

V metodě `run()` jsme vytvořili nekonečný cyklus, což je pro vlákna ve formě démonů typické. My jsme tento způsob zvolili z toho důvodu, že nevíme, kolik souborů musí dané vlákno zpracovat. V kaž-

dé iteraci voláme metodu `queue.Queue.get()` pro získání dalšího souboru, který se má zpracovat. Toto volání bude blokovat, je-li fronta prázdná, a nemusíme jej chránit zámkem, protože o ten se za nás automaticky postará třída `queue.Queue`. Jakmile máme soubor, zpracujeme jej a potom musíme říci frontě, že jsme již tuto činnost dokončili – volání metody `queue.Queue.task_done()` je zásadní pro správné fungování metody `queue.Queue.join()`.

Funkci `process()` jsme si neukázali, protože její kód je kromě řádku s příkazem `def` stejný jako kód použitý v programu `grepword-p-child.py` od řádku `previous = ""` až po konec (srana 427).

Ještě je třeba poznamenat, že mezi zdrojovými kódy dodávanými s touto knihou je též program `grepwordm.py`, který je téměř stejný jako zde probíraný program `grepword-t.py`, tedy až na to, že místo modulu `threading` používá modul `multiprocessing`. V kódu jsou jen tři odlišnosti. Za prvé, místo modulů `queue` a `threading` importujeme modul `multiprocessing`. Za druhé, třída `Worker` není odvozena od třídy `threading.Thread`, ale od třídy `multiprocessing.Process`. A za třetí, pracovní fronta není implementována jako objekt typu `queue.Queue`, ale jako objekt typu `multiprocessing.JoinableQueue`.

Modul `multiprocessing` poskytuje funkčnost podobnou vláknům pomocí systémového volání `fork` na systémech, které jej podporují (Unix), a podřízených procesů na těch, které nikoliv (Windows), takže zamykání není vždy vyžadováno a procesy poběží na kterýchkoli jádrech procesoru dostupných v daném operačním systému. Tento balíček poskytuje několik způsobů pro předávání dat mezi procesy, mezi něž patří i fronta, kterou lze použít k poskytování práce pro procesy podobným způsobem, jakým objekty typu `queue.Queue` poskytují práci vláknům.

Hlavní přednost verze využívající modul `multiprocesing` spočívá v tom, že na vícejádrových strojích dokáže běžet potenciálně rychleji než vláknová verze, protože své procesy umí spouštět na všech dostupných jádrech. Naproti tomu standardní interpret jazyka Python (napsaný v jazyku C, takže se někdy označuje jako CPython) obsahuje zámek GIL (Global Interpreter Lock – globální zámek interpretu), což znamená, že kód napsaný v jazyku Python může v kterémkoli okamžiku provádět pouze jedno vlákno. Toto omezení představuje implementační detail a nemusí nutně platit i pro jiné interprety jazyka Python, jako je například Jython.*

Příklad: Vícevláknový program pro hledání duplicitních souborů

Druhý příklad využívající vlákna má podobnou strukturu jako první, je však v několika směrech sofistikovanější. Používá dvě fronty, jednu pro práci a druhou pro výsledky, a dále obsahuje samostatné vlákno pro zpracování výsledků, které výsledky vypisuje ihned, jakmile jsou dostupné. V příkladu používáme jak odvození nové třídy od třídy `threading.Thread`, tak i volání konstruktoru `threading.Thread()` s funkcí. Přístup ke sdíleným datům (instance třídy `dict`) pak serializujeme pomocí zámku.

Program `findduplicates-t.py` je pokročilejší verzí programu `finddup.py` z lekce 5. Prochází všechny soubory v aktuálním adresáři (nebo na zadané cestě) a při tom rekurzivně sestupuje do podadresářů. Porovnává délky všech souborů se stejným názvem (podobně jako program `finddup.py`) a u těch, které mají stejný název i stejnou velikost, pak pomocí algoritmu MD5 (Message Digest – otisk zprávy) zkontroluje, zda mají stejný obsah. Všechny soubory, které se takto shodují, vypíše.

* Stručné vysvětlení důvodu, proč interpret CPython používá zámek GIL, najdete na stránce *http://www.python.org/doc/faq/library/#can-t-we-get-rid-of-the-global-interpreter-lock*.

Začneme pohledem na funkci `main()`, kterou si rozdělíme na čtyři části.

```
def main():
    opts, path = parse_options()
    data = collections.defaultdict(list)
    if opts.verbose:
        print("Vytvářím seznam souborů...")
    for root, dirs, files in os.walk(path):
        for filename in files:
            fullname = os.path.join(root, filename)
            try:
                key = (os.path.getsize(fullname), filename)
            except EnvironmentError:
                continue
            if key[0] == 0:
                continue
            data[key].append(fullname)
```

Každým klíčem výchozího slovníku `data` je n-tice se dvěma prvky (velikost, název souboru), kde název souboru neobsahuje cestu, a každá hodnota je seznam názvů souborů (včetně cest). Jakékoli prvky, jejichž hodnota obsahuje více než jeden název souboru, mají potenciální duplicity. Tento slovník naplňujeme procházením všech souborů na zadané cestě a přitom přeskakujeme všechny soubory, u nichž nemůžeme zjistit velikost (třeba kvůli problémům s oprávněním nebo proto, že se nejedná o běžné soubory) a jejichž velikost je 0 (protože všechny soubory s nulovou délkou jsou stejné).

```
    work_queue = queue.PriorityQueue()
    results_queue = queue.Queue()
    md5_from_filename = {}
    for i in range(opts.count):
        number = "{0}: ".format(i + 1) if opts.debug else ""
        worker = Worker(work_queue, md5_from_filename, results_queue,
                        number)
        worker.daemon = True
        worker.start()
```

Se všemi daty na svém místě jsme připraveni vytvořit pracovní vlákna. Začínáme tvorbou pracovní fronty a fronty s výsledky. Pracovní frontu implementujeme jako prioritní frontu, takže bude vždy jako první vracet prvky s nejnižší prioritou (v našem případě nejmenší soubory). Dále vytváříme slovník, jehož klíči jsou názvy souborů (včetně cesty) a hodnotami otisk MD5 příslušného souboru. Smyslem tohoto slovníku je zajistit, abychom pro žádný soubor nepočítali otisk MD5 dvakrát (poněvadž jeho výpočet je drahý).

S kolekcí sdílených dat na svém místě provádíme průchod cyklem podle počtu vláken, jež máme vytvořit (výchozí počet průchodů je 7). Podtřída `Worker` je podobná té, kterou jsme vytvořili již dříve, tentokrát ale předáváme obě fronty a slovník s otisky MD5. Stejně jako předtím i nyní ihned každé pracovní vlákno spouštíme a každé zůstane blokované, dokud nebudou k dispozici položky na zpracování.

```
    results_thread = threading.Thread(
                        target=lambda: print_results(results_queue))
    results_thread.daemon = True
    results_thread.start()
```

Pro zpracování výsledků nevytváříme podtřídu třídy `threading.Thread`, ale funkci, kterou předáváme konstruktoru `threading.Thread()`. Návratovou hodnotou je vlastní vlákno, které bude po svém spuštění volat zadanou funkci. Té předáváme frontu s výsledky (která je nyní samozřejmě prázdná), takže vlákno bude okamžitě blokovat.

V tomto okamžiku máme vytvořená všechna pracovní vlákna a vlákno pro zpracování výsledků, přičemž všechna čekají zablokována na práci.

```
    for size, filename in sorted(data):
        names = data[size, filename]
        if len(names) > 1:
            work_queue.put((size, names))
    work_queue.join()
    results_queue.join()
```

Nyní procházíme data a pro každou dvojici (velikost, název souboru), která obsahuje seznam dvou či více potenciálně duplicitních souborů, přidáme tuto velikost a názvy souborů s cestami jako položku na zpracování do pracovní fronty. Tato fronta je třídou z modulu `queue`, a proto se nemusíme starat o zamykání.

Nakonec pracovní frontu a frontu s výsledky spojíme, čímž blokujeme provádění, dokud se tyto fronty nevyprázdní. Tím zajistíme, že program poběží, dokud se nedokončí veškerá zpracování a nevypíší se všechny výsledky, a až poté se čistě ukončí.

```
def print_results(results_queue):
    while True:
        try:
            results = results_queue.get()
            if results:
                print(results)
        finally:
            results_queue.task_done()
```

Tuto funkci předáváme jako argument konstruktoru `threading.Thread()` a zavolá se při spuštění vlákna, kterému ji předáváme. Obsahuje nekonečný cyklus, protože se bude používat jako vlákno představující démona. Provádí jen to, že bere výsledky (víceřádkové řetězce), a dokud jsou k dispozici, tak je v případě, že je řetězec neprázdný, vypisuje.

Začátek třídy `Worker` je stejný jako v její předchozí verzi:

```
class Worker(threading.Thread):

    Md5_lock = threading.Lock()
```

```
    def __init__(self, work_queue, md5_from_filename, results_queue,
                 number):
        super().__init__()
        self.work_queue = work_queue
        self.md5_from_filename = md5_from_filename
        self.results_queue = results_queue
        self.number = number

    def run(self):
        while True:
            try:
                size, names = self.work_queue.get()
                self.process(size, names)
            finally:
                self.work_queue.task_done()
```

Rozdíl je v tom, že zde uchováváme více sdílených dat a vlastní funkci `process()` voláme s odlišnými argumenty. O fronty se starat nemusíme, protože zajišťují serializovaný přístup, u ostatních dat, v tomto případě u slovníku `md5_from_filename`, však musíme serializovaný přístup zajistit sami pomocí zámku. Zámek je atributem třídy, protože chceme, aby každá instance třídy `Worker` používala stejný zámek, takže pokud jej jedna instance získá, budou ostatní při pokusu o jeho získání blokovány.

Funkci `process()` si rozdělíme na dvě části.

```
    def process(self, size, filenames):
        md5s = collections.defaultdict(set)
        for filename in filenames:
            with self.Md5_lock:
                md5 = self.md5_from_filename.get(filename, None)
            if md5 is not None:
                md5s[md5].add(filename)
            else:
                try:
                    md5 = hashlib.md5()
                    with open(filename, "rb") as fh:
                        md5.update(fh.read())
                    md5 = md5.digest()
                    md5s[md5].add(filename)
                    with self.Md5_lock:
                        self.md5_from_filename[filename] = md5
                except EnvironmentError:
                    continue
```

Začínáme prázdným slovníkem s výchozími hodnotami, jehož klíče jsou otisky MD5 a hodnoty množiny s názvy souborů, které mají odpovídající otisk MD5. Potom procházíme všechny soubory a pro každý vezmeme jeho otisk MD5, pokud jsme jej již vypočítali, nebo jej v opačném případě vypočítáme.

Správci kontextu ➢ 357

Ať už přistupujeme ke slovníku `md5_from_filename` kvůli zápisu nebo kvůli čtení, celý přístup umisťujeme do kontextu zámku. Instance třídy `threading.Lock()` jsou správce kontextu, které při vstupu získají zámek a při výstupu jej uvolní. Příkazy `with` budou blokovat, má-li zámek `Md5_lock` jiné vlákno, a to až do okamžiku uvolnění tohoto zámku. U prvního příkazu `with` bereme při získávání zámku otisk MD5 ze slovníku (z něhož obdržíme hodnotu `None`, pokud příslušný otisk neobsahuje). Má-li tisk MD5 hodnotu `None`, musíme jej spočítat a poté jej uložit do slovníku `md5_from_filename`, abychom jej v případě potřeby nemuseli počítat znovu.

Všimněte si, že se vždy snažíme minimalizovat množství práce prováděné v kontextu zámku, abychom tak udržovali blokování na minimum. V tomto případě provádíme jen jeden přístup do slovníku.

Zámek GIL ➢ 432

Přesněji řečeno, pokud používáme interpret CPython, pak žádný zámek používat nemusíme, protože zámek GIL efektivně synchronizuje přístupy do slovníku za nás. My jsme se ovšem rozhodli programovat bez spoléhání se na implementační detail ve formě zámku GIL, a proto používáme explicitní zámek:

```
for filenames in md5s.values():
    if len(filenames) == 1:
        continue
    self.results_queue.put("{0} Duplicitních souborů ({1:n} bajtů):"
                           "\n\t{2}".format(self.number, size,
                              "\n\t".join(sorted(filenames))))
```

Na konci procházíme lokální slovník s výchozími hodnotami `md5s` a pro každou množinu názvů, jež obsahuje více než jeden název, přidáme do fronty s výsledky víceřádkový řetězec. Tento řetězec obsahuje číslo pracovního vlákna (výchozí je prázdný řetězec), velikost souboru v bajtech a názvy všech duplicitních souborů. Pro přístup k frontě s výsledky nemusíme používat zámek, protože se jedná o objekt typu `queue.Queue`, který se o zamykání automaticky postará.

Třídy modulu `queue` výrazně zjednodušují vícevláknové aplikace, a když potřebujeme použít explicitní zámky, nabízí nám modul `threading` řadu možností. Zde jsme použili nejjednodušší z nich, třídu `threading.Lock`, k dispozici jsou však i další, kupříkladu `threading.RLock` (zámek, který lze opětovně získat vláknem, které jej již vlastní), `threading.Semaphore` (zámek, který lze použít k ochraně určitého počtu zdrojů) a `threading.Condition`, což je třída poskytující čekací podmínku.

Zámek GIL ➢ 432

Použití více vláken může často vést k čistšímu řešení než použití modulu `subprocess`, ovšem pravdou je, že vícevláknové programy napsané v jazyku Python nedosahují ve srovnání s použitím více procesů nutně nejlepšího možného výkonu. Jak jsme si řekli již dříve, tento problém postihuje standardní implementaci Pythonu, neboť interpret CPython dokáže provádět kód jazyka Python v jednom okamžiku pouze na jediném procesoru, a to i při použití více vláken.

Balíčkem, který se snaží tento problém vyřešit, je modul `multiprocessing`, a jak jsme si řekli již dříve, k dispozici je verze programu `grepword-t.py` s názvem `grepword-m.py`, která využívá modul `multiprocessing` a která se liší pouze na třech řádcích kódu. Podobnou transformaci bychom mohli

aplikovat na výše probíraný program `findduplicates-t.py`, v praxi se to ale nedoporučuje. Přestože modul `multiprocessing` nabízí rozhraní API (Application Programming Interface – aplikační programovací rozhraní), které pro snazší přechod odpovídá rozhraní API modulu `threading`, nejsou tato dvě rozhraní zcela stejná a každé má své klady a zápory. Kromě toho provádění mechanického přechodu z modulu `threading` na modul `multiprocessing` bude mít pravděpodobně úspěch jen u malých, jednoduchých programů, jako je `grepword-t.py`. Aplikace tohoto postupu na program `findduplicates-t.py` by byla příliš surová, přičemž obecně je nejlepší brát do úvahy využití modulu `multiprocessing` již od začátku návrhu programu. (V rámci zdrojových kódů ke knize je k dispozici také program `findduplicates-m.py`, který provádí stejnou činnost jako program `findduplicatest.py`, avšak funguje naprosto odlišným způsobem a používá modul `multiprocessing`.)

Dalším řešením, které se v současnosti vyvíjí, je verze interpretu CPython přívětivá k vláknům (nejnovější stav projektu můžeme zjistit na stránce *http://code.google.com/p/python-safethread/*).

Shrnutí

V této lekci jsme si ukázali, jak vytvářet programy, které dokážou spouštět jiné programy pomocí modulu `subprocess` ze standardní knihovny. Programům, které se spouštějí modulem `subprocess`, lze předávat data na příkazovém řádku, lze je krmit daty prostřednictvím jejich standardního vstupu a dále lze číst jejich standardní výstup (a standardní chybový výstup). Použitím podřízených procesů můžeme maximalizovat výhody vícejádrových procesorů a ponechat problémy se souběžností na operačním systému. Nevýhodou ale je, že pokud potřebujeme sdílet data nebo synchronizovat procesy, pak musíme vymyslet nějaký druh komunikačního mechanizmu, jako je například sdílená paměť (např. pomocí modulu `mmap`), sdílené soubory nebo komunikace prostřednictvím síťové vrstvy, což může ke správnému fungování vyžadovat určité úsilí.

V této lekci jsme si dále ukázali, jak vytvářet vícevláknové programy. Takové programy však nemohou naneštěstí využít plného potenciálu více jader (při spouštění pomocí standardního interpretu CPython), a proto je pro Python v případě požadavků na výkon mnohdy praktičtější použít více procesů. Nicméně viděli jsme, že díky modulu `queue` a mechanizmu zamykání Pythonu (např. ve formě třídy `threading.Lock`) je práce s více vlákny velice přímočarou záležitostí a že u jednoduchých programů, které si vystačí pouze s frontami typu `queue.Queue` a `queue.PriorityQueue`, se nemusíme explicitními zámky vůbec zabývat.

Přestože vícevláknové programování je nepochybně módní, může být psaní, údržba a ladění vícevláknových programů ve srovnání s jednovláknovými programy mnohem náročnější. Vícevláknové programy na druhou stranu umožňují oproti podřízeným procesům přímočarou komunikaci, například pomocí sdílených dat (ovšem za předpokladu, že použijeme třídu `queue` nebo zamykání), a lze je mnohem snadněji synchronizovat (např. pro shromažďování výsledků). Vlákna mohou být také velice užitečná v programech s rozhraním GUI (Graphical User Interface – grafické uživatelské rozhraní), které musejí provádět dlouho běžící úkoly a přitom poskytovat stejnou responsivnost a schopnost právě běžící úkol kdykoli zrušit. Pokud ale použijeme kvalitní mechanizmus pro komunikaci mezi procesy, jako je kupříkladu sdílená paměť nebo fronta transparentní vzhledem k procesům nabízená balíčkem `multiprocessing`, pak se použití více procesů může často stát reálnou alternativou k více vláknům.

V následující lekci si ukážeme další příklad vícevláknového programu. Jedná se o server, který obsluhuje každý klientský požadavek v samostatném vlákně a který používá zámky k ochraně sdílených dat.

Cvičení

1. Zkopírujte program `grepword-p.py` a upravte jej tak, aby podřízené procesy svůj výstup nevypisovaly, ale aby výsledky shromažďoval hlavní program a po dokončení všech podřízených procesů je seřadil a vypsal. K tomu stačí jen upravit funkci `main()`, změnit tři řádky a další tři přidat. U tohoto cvičení musíme postupovat s rozvahou a pečlivostí a dále je třeba pročíst dokumentaci k modulu `subprocess`. Řešení je k dispozici v souboru `grepword-p_ans.py`.

2. Napište vícevláknový program, který čte soubory uvedené na příkazovém řádku (a rekurzivně také soubory v adresářích uvedených na příkazovém řádku). Všechny soubory ve formátu XML (tj. začínající znaky „<?xml“) analyzujte pomocí analyzátoru jazyka XML a vytvořte seznam jedinečných značek použitých v daném souboru nebo vypište chybovou zprávu, pokud dojde během analýzy k chybě. Zde je jeden ukázkový běh programu a jeho výstup:

```
.\data\dvds.xml je soubor XML, který používá následující značky:
    dvd
    dvds
.\data\bad.aix je soubor XML, který obsahuje následující chybu:
    mismatched tag: line 7889, column 2
.\data\incidents.aix je soubor XML, který používá následující značky:
    airport
    incident
    incidents
    narrative
```

Nejjednodušší způsob vytvoření tohoto programu spočívá v úpravě kopie programu `findduplicates-t.py`, i když vám samozřejmě nic nebrání napsat celý program úplně od začátku. Malé změny bude třeba provést v metodách `__init__()` a `run()` třídy `Worker`, přičemž metodu `process()` je nutné zcela přepsat (vystačí si s přibližně dvaceti řádky). Funkce `main()` programu bude potřebovat několik zjednodušení stejně jako jeden řádek funkce `print_results()`. Dále bude nutné upravit zprávu o použití programu, aby odpovídala následujícímu výpisu:

```
Použití: xmlsummary.py [volby] [cesta]
vypíše souhrnné informace o souborech XML na zadané cestě
výchozí hodnotou cesty je .

Options:
  -h, --help    show this help message and exit
  -t COUNT, --threads=COUNT
                počet vláken, které se mají použít (1..20) [výchozí 7]
  -v, --verbose
  -d, --debug
```

Nezapomeňte zkusit spustit program s příznakem `-d` (ladění), abyste mohli ověřit, že se vlákna spouštějí a že každé provádí svůj díl práce. Řešení najdete v souboru `xmlsummary.py`, který má něco málo přes 100 řádků a nepoužívá explicitní zámky.

LEKCE 11

Propojení v síti

V této lekci:

- Tvorba klienta TCP
- Tvorba serveru TCP

Propojení v síti umožňuje počítačovým programům navzájem komunikovat, a to i tehdy, běží-li na různých strojích. U programů, jako jsou webové prohlížeče, se jedná o podstatu jejich činnosti, zatímco u ostatních přidává propojení v síti další dimenze k jejich funkčnosti, jako jsou například vzdálené operace či protokolování nebo schopnost získávat či poskytovat data ostatních strojům. Většina programů propojených v síti funguje buď na bázi peer-to-peer (stejný program běží na různých strojích), nebo na bázi klient-server (klientské programy posílají požadavky na server), což je obvyklejší.

V této lekci vytvoříme základní aplikaci s architekturou klient-server. Aplikace tohoto typu se obvykle implementují jako dva samostatné programy: server, který čeká na požadavky a odpovídá na ně, a jeden či více klientů, kteří odesílají požadavky na server a přijímají odpověď serveru. Aby to vše fungovalo, musejí klienti vědět, kde se mohou k serveru připojit, tj. musejí znát IP adresu a číslo portu serveru.* Klienti i server musejí odesílat a přijímat data pomocí předem dohodnutého protokolu s použitím takových datových formátů, kterým oba rozumí.

Nízkoúrovňový modul Pythonu s názvem `socket` (na němž jsou založeny všechny ostatní moduly Pythonu vyšší úrovně pro propojení v síti) podporuje adresy protokolu IPv4 a IPv6. Dále podporuje většinu běžně užívaných síťových protokolů, včetně protokolu UDP (User Datagram Protocol – uživatelský datagramový protokol), což je odlehčený, nespolehlivý a nespojovaný protokol, který posílá data jako samostatné pakety (datagramy) bez záruky na doručení, a protokolu TCP (Transmission Control Protocol – protokol pro řízení přenosu), což je spolehlivý, spojovaný a proudově orientovaný protokol. Prostřednictvím protokolu TCP lze odesílat a přijímat libovolné množství dat, přičemž soket je odpovědný za rozbití dat na kousky dostatečně malé k odeslání a za jejich rekonstrukci na druhém konci.

Protokol UDP se často používá k monitorování přístrojů, které poskytují nepřetržité údaje a u nichž není příležitostná ztráta údajů podstatná. V případech, kdy je občasně chybějící snímek přijatelný, se používá také pro streamování audia a videa. Protokoly FTP a HTTP jsou postaveny na protokolu TCP a aplikace typu klient-server běžně používají protokol TCP, protože potřebují spojově orientovanou komunikaci a spolehlivost, což nabízí právě protokol TCP. V této lekci vyvineme program s architekturou klient-server, a proto budeme používat protokol TCP.

Naložené objekty
➢ 285

Dalším rozhodnutím, které je třeba udělat, je to, zda se budou data odesílat a přijímat jako řádky textu nebo jako bloky binárních dat. Půjde-li o bloky binárních dat, pak je třeba určit jejich formu. V této lekci použijeme bloky binárních dat, v nichž první čtyři bajty představují délku následujících dat (zakódovaných jako celé číslo bez znaménka pomocí modulu `struct`), která jsou sama o sobě naloženým objektem (pickle). Výhodou tohoto přístupu je, že můžeme použít stejný kód pro odesílání a přijímání v libovolné aplikaci, protože do naloženého objektu můžeme uložit téměř jakákoli data. Nevýhodou pak je, že klient i server musejí naloženým objektům rozumět, takže musejí být napsány v Pythonu nebo musejí být schopny přistupovat k Pythonu, třeba pomocí knihovny Jython v Javě nebo Boost.Python v C++. A samozřejmě i zde se na použití naložených objektů vztahují obvyklé bezpečnostní pokyny.

Naším příkladem bude program pro registraci vozidel. Server uchovává podrobnosti o registraci vozidla (poznávací značka, počet míst, počet najetých kilometrů a vlastník). Klient se používá k získávání detailů o vozech, ke změně počtu najetých kilometrů či vlastníka nebo k vytvoření nové registrace vozu. Je možné použít libovolný počet klientů, kteří se navzájem nebudou blokovat, a to i tehdy, pokud by ve stejný okamžik přistupovalo k serveru více klientů. To je dáno tím, že server zpracová-

* Stroje se mohou připojovat také pomocí zjišťování služeb, například pomocí rozhraní API Bonjour. Vhodné moduly jsou k dispozici v seznamu balíčků pro jazyk Python – *pypi.python.org/pypi*.

vá požadavek každého klienta v samostatném vláknu. (Kromě toho uvidíme, že stejně snadno by se daly použít i samostatné procesy.)

Pro účely tohoto příkladu budeme server i klienty spouštět na stejném stroji. To znamená, že můžeme jako IP adresu použít „localhost" (pokud by ale server byl na jiném stroji, pak bychom klientovi mohli předat IP adresu na příkazovém řádku, což by perfektně fungovalo, pokud by ovšem na cestě nebyl žádný firewall). Zvolili jsme nahodilé číslo portu 9653. Číslo portu by mělo být vyšší než 1023, běžně se používají čísla mezi 5 001 a 32 767, ačkoliv platná čísla jsou až po 65 535.

Server může přijímat pět požadavků: `GET_CAR_DETAILS` (vezmi detaily vozu), `CHANGE_MILEAGE` (změň počet najetých kilometrů), `CHANGE_OWNER` (změň vlastníka), `NEW_REGISTRATION` (nová registrace) a `SHUTDOWN` (zastav činnost). Na každý z těchto požadavků reaguje příslušná odpověď, která je tvořena požadovanými daty, potvrzením o provedení požadované akce nebo oznámením chyby.

Tvorba klienta TCP

Klientským programem je `car_registration.py`. Zde je příklad interakce (server již běží):

```
(V)ůz  Upravit K(i)lometry  Upravit V(l)astníka  (N)ový vůz  (Z)astavit server
(K)onec [v]:
SPZ: 3m7 1236
SPZ: 3M7 1236
Počet sedadel: 2
Počet najetých km: 97543
Vlastník: Josef Loudal
(V)ůz  Upravit K(i)lometry  Upravit V(l)astníka  (N)ový vůz  (Z)astavit server
(K)onec [v]: m
SPZ [3M7 1236]:
Počet najetých km [97543]: 103491
Údaj o počtu najetých km byl úspěšně změněn
```

Data zadávaná uživatelem jsou zvýrazněna – řádky bez viditelného vstupu znamenají, že uživatel stiskem klávesy Enter přijal výchozí hodnotu. V tomto příkladu uživatel požádal o detaily určitého vozu a poté změnil počet jeho najetých kilometrů.

Spouštět lze tolik klientů, kolik jen chceme, a když pak uživatel svého klienta ukončí, server tím nebude nijak dotčen. Je-li však zastaven server, pak se ukončí i klient, v němž byl zastaven, a všichni ostatní klienti obdrží při následném pokusu o přístup k serveru chybu „Spojení odmítnuto" a budou ukončeni. V sofistikovanější aplikaci by možnost zastavit server byla dostupná pouze určitým uživatelům, třeba jen na některých strojích, my jsme ji ale umístili do klienta, abychom si v prvé řadě ukázali, jak se provádí.

Nyní se podíváme na kód. Začneme funkcí `main()` a obsluhou uživatelského rozhraní a skončíme samotným síťovým kódem.

```
def main():
    if len(sys.argv) > 1:
        Address[0] = sys.argv[1]
```

```
    call = dict(v=get_car_details, i=change_mileage, l=change_owner,
                n=new_registration, z=stop_server, k=quit)
    menu = ("(V)ůz  Upravit K(i)lometry  Upravit V(l)astníka  (N)ový vůz  "
            "(Z)astavit server  (K)onec")
    valid = frozenset("vilnzk")
    previous_license = None
    while True:
        action = Console.get_menu_choice(menu, valid, "v", True)
        previous_license = call[action](previous_license)
```

Větvení pomocí slovníků ➤ 331

Proměnná `Address` je globální dvouprvkový seznam uchovávající IP adresu a číslo počtu (`["localhost", 9653]`), přičemž IP adresu přepíšeme, je-li zadána na příkazovém řádku. Slovník `call` mapuje volby nabídky na funkce.

Modul `Console` je dodáván se zdrojovými kódy této knihy a obsahuje několik užitečných funkcí pro získání hodnot zadaných uživatelem na příkazovém řádku. Mezi tyto funkce patří například `Console.get_string()` a `Console.get_integer()`, jež jsou podobné funkcím, které jsme vyvinuli v předchozích lekcích a které byly umístěny do modulu, aby byly snadno použitelné v různých programech.

Pro pohodlí uživatelů uchováváme poslední zadanou SPZ, aby ji mohli použít jako výchozí, protože většina příkazů začíná dotazem na SPZ příslušného vozidla. Jakmile uživatel zadá svou volbu, zavoláme odpovídající funkci, které předáme předchozí SPZ, a zároveň očekáváme, že každá funkce vrátí SPZ, kterou použila. Cyklus je nekonečný, a proto musí program ukončit jedna z funkcí, což si za okamžik ukážeme.

```
def get_car_details(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is not None:
        print("SPZ: {0}\nPočet sedadel: {seats}\n"
              "Počet Najetých km: {mileage}\n"
              "Vlastník: {owner}".format(license, **car._asdict()))
    return license
```

Tato funkce se používá pro získání informací o určitém voze. Většina těchto funkcí vyžaduje zadání SPZ uživatelem a často potřebuje pracovat s některými daty souvisejícími s vozidlem, a proto jsme tuto funkčnost vyčlenili do funkce `retrieve_car_details()`. Ta vrací n-tici se dvěma prvky obsahující SPZ zadanou uživatelem a pojmenovanou n-tici `CarTuple`, jež uchovává počet sedadel, počet najetých kilometrů a vlastníka vozidla (nebo předchozí SPZ a `None`, pokud uživatel zadal neznámou SPZ). Zde jen vypíšeme získané informace a vrátíme SPZ, která bude použita jako výchozí pro další zavolanou funkci vyžadující zadání SPZ.

```
def retrieve_car_details(previous_license):
    license = Console.get_string("SPZ", "SPZ",
                                 previous_license)
    if not license:
        return previous_license, None
    license = license.upper()
```

```
    ok, *data = handle_request("GET_CAR_DETAILS", license)
    if not ok:
        print(data[0])
        return previous_license, None
    return license, CarTuple(*data)
```

Jedná se o první funkci využívající síťové prvky. Voláme zde funkci `handle_request()`, kterou si ukážeme později. Funkce `handle_request()` přijímá jako své argumenty jakákoli data, která odešle na server, a poté vrátí cokoliv, co server odpověděl. Tato funkce o datech, která odesílá a vrací, vůbec nic neví a ani se o ně nestará. Jejím jediným úkolem je poskytovat službu propojení v síti.

V případě registrace vozidel máme protokol, v rámci něhož vždy posíláme jako první argument název akce, kterou má server provést, za níž následují příslušné parametry – v tomto případě pouze SPZ. Odpověď vypadá tak, že server vždy vrátí n-tici, jejíž první prvek je příznak ve formě logické hodnoty signalizující úspěch či neúspěch. Má-li tento příznak hodnotu `False`, pak máme n-tici se dvěma prvky, přičemž druhým prvek je chybová zpráva. Má-li příznak hodnotu `True`, pak máme také n-tici se dvěma prvky, přičemž druhý prvek obsahuje potvrzující zprávu nebo n-tici, jejíž druhý prvek a všechny následující uchovávají požadovaná data.

Je-li tedy v tomto případě SPZ neznámá, bude mít proměnná `ok` hodnotu `False`, a proto vypíšeme chybovou zprávu v prvku `data[0]` a vrátíme předchozí SPZ beze změny. V opačném případě vrátíme SPZ (z níž se nyní stane předchozí SPZ) a instanci n-tice `CarTuple` vytvořenou ze seznamu `data` (počet sedadel, počet najetých kilometrů, vlastník).

```
def change_mileage(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is None:
        return previous_license
    mileage = Console.get_integer("Počet najetých km", "počet najetých km",
                                  car.mileage, 0)
    if mileage == 0:
        return license
    ok, *data = handle_request("CHANGE_MILEAGE", license, mileage)
    if not ok:
        print(data[0])
    else:
        print("Údaj o počtu najetých km byl úspěšně změněn")
    return license
```

Tato funkce pracuje ve stejném duchu jako funkce `get_car_details()`, tedy až na to, že jakmile údaje obdržíme, tak některý z nich aktualizujeme. V této funkci jsou ve skutečnosti dvě síťová volání, protože funkce `retrieve_car_details()` volá funkci `handle_request()` pro získání detailů o vozidle, což je nezbytné pro potvrzení platnosti SPZ a pro získání údaje o aktuálního počtu najetých kilometrů, který použijeme jako výchozí hodnotu. Zde je odpověď vždy n-ticí se dvěma prvky, kde druhý prvek je buď chybová zpráva, nebo `None`.

Funkci `change_owner()` si neukážeme, protože je z hlediska struktury stejná jako funkce `change_mileage()`, a neukážeme si ani funkci `new_registration()`, která se liší pouze v tom, že na začátku

nezískává údaje o vozidle (protože se zadává vozidlo nové) a místo změny jednoho žádá uživatele o všechny údaje. Nic z toho pro nás není nové ani to nijak nesouvisí se síťovým programováním.

```
def quit(*ignore):
    sys.exit()

def stop_server(*ignore):
    handle_request("SHUTDOWN", wait_for_reply=False)
    sys.exit()
```

Pokud se uživatel rozhodne ukončit program, pak voláním funkce `sys.exit()` provedeme čisté ukončení. Každá funkce nabídky se volá s předchozí SPZ, avšak v tomto případě nás tento argument nezajímá. Nemůžeme ale napsat `def quit():`, protože tím by se vytvořila funkce, která neočekává žádné argumenty, takže při jejím zavolání s argumentem předchozí SPZ by se vyvolala výjimka `TypeError` oznamující, že nebyly očekávány žádné argumenty, a přesto byl jeden zadán. Proto raději uvádíme parametr `*ignore`, který dokáže přijmout libovolný počet pozičních argumentů. Jeho název (`ignore`) nemá pro Python žádný speciální význam a použili jsme jej pouze kvůli tomu, aby údržbáři věděli, že se argumenty ignorují.

Pokud se uživatel rozhodne zastavit server, pak jej informujeme pomocí funkce `handle_request()`, které řekneme, že žádnou odpověď nechceme. Jakmile se data odešlou, vrátí nám funkce `handle_request()` řízení, aniž by čekala na odpověď, a my pomocí funkce `sys.exit()` provedeme čisté ukončení.

```
def handle_request(*items, wait_for_reply=True):
    SizeStruct = struct.Struct("!I")
    data = pickle.dumps(items, 3)

    try:
        with SocketManager(tuple(Address)) as sock:
            sock.sendall(SizeStruct.pack(len(data)))
            sock.sendall(data)
            if not wait_for_reply:
                return

            size_data = sock.recv(SizeStruct.size)
            size = SizeStruct.unpack(size_data)[0]
            result = bytearray()
            while True:
                data = sock.recv(4000)
                if not data:
                    break
                result.extend(data)
                if len(result) >= size:
                    break
        return pickle.loads(result)
    except socket.error as err:
```

```
        print("{0}: je server spuštěn?".format(err))
        sys.exit(1)
```

Tato funkce obstarává pro klientský program veškerou práci se sítí. Začíná vytvořením struktury `struct.Struct`, která uchovává jedno celé číslo bez znaménka se síťovým pořadím bajtů a poté z předaných prvků vytvoří naložený objekt (pickle). O těchto prvcích funkce neví a ani se o ně nestará. Všimněte si, že jsme explicitně nastavili protokol pro naložené objekty na hodnotu 3. Tím zajistíme, že klienti i server používají stejnou verzi naložených objektů, a to i v situaci, když dojde k přechodu klienta či serveru na odlišnou verzi Pythonu.

Pokud bychom chtěli, aby náš protokol byl odolný vůči budoucím změnám, pak bychom mohli přidat označení verze (tedy stejně jako v případě binárních diskových formátů). To lze provést buď na síťové úrovni, nebo na datové úrovni. Na síťové úrovni můžeme doplnit informaci o verzi tak, že místo jednoho celého čísla bez znaménka předáme dvě, tedy délku a číslo verze protokolu. Na datové úrovni můžeme postupovat podle konvence, že naložený objekt je vždy seznam (nebo vždy slovník), jehož první prvek (či prvek s „verzí“) obsahuje číslo verze. (Doplnění verze do protokolu bude vaším úkolem ve cvičeních.)

Třída `SocketManager` je náš vlastní správce kontextu, která nám poskytuje soket (vrátíme se k ní za okamžik). Metoda `socket.socket.sendall()` odešle všechna zadaná data, přičemž v pozadí provede v případě nutnosti několik volání metody `socket.socket.send()`. Vždy posíláme dva prvky dat: délku naloženého objektu a samotný naložený objekt. Má-li parametr `wait_for_reply` hodnotu `False`, nečekáme na odpověď a okamžitě funkci opustíme – správce kontextu zajistí, že se soket před samotným opuštěním funkce uzavře.

Po odeslání dat (a v případě, že chceme odpověď) zavoláme pro získání odpovědi metodu `socket.socket.recv()`. Tato metoda blokuje, dokud neobdrží data. Při jejím prvním zavolání požadujeme čtyři bajty, což je velikost celého čísla, které uchovává velikost naloženého objektu, který následuje. Tyto bajty rozbalíme do celočíselné proměnné `size` pomocí objektu typu `struct.Struct`. Potom vytvoříme prázdný objekt typu `bytearray` a pokusíme se načíst příchozí naložený objekt v blocích o velikosti 4000 bajtů. Jakmile načteme alespoň tolik bajtů, kolik udává proměnná `size` (nebo pokud předtím dojdou data), pak opustíme cyklus a pomocí funkce `pickle.loads()` (která přijímá objekt typu `bytes` nebo `bytearray`) vytáhneme data a vrátíme je. V tomto případě víme, že data budou mít vždy podobu n-tice, protože takto jsme ustanovili náš protokol se serverem pro registraci vozidel. O to se ale funkce `handle_request()` nestará a ani to neví.

Pokud se v rámci síťového spojení něco nezdaří (např. server neběží nebo spojení z nějakého důvodu selže), vyvolá se výjimka `socket.error`. V takovýchto případech je výjimka zachycena a klientský program vypíše chybovou zprávu a skončí.

```
class SocketManager:

    def __init__(self, address):
        self.address = address

    def __enter__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect(self.address)
```

```
        return self.sock

    def __exit__(self, *ignore):
        self.sock.close()
```

Objekt `address` je n-tice se dvěma prvky (IP adresa, číslo portu) a nastavuje se při vytvoření tohoto správce kontextu. Jakmile správce kontextu použijeme v příkazu `with`, vytvoří soket a pokusí se ustanovit spojení a přitom blokuje, dokud se spojení neustaví nebo dokud nedojde k vyvolání výjimky soketu. Prvním argumentem inicializační funkce `socket.socket()` je rodina adres. Zde jsme použili hodnotu `socket.AF_INET` (protokol IPv4), k dispozici jsou ale i další, například `socket.AF_INET6` (protokol IPv6), `socket.AF_UNIX` a `socket.AF_NETLINK`. Druhým argumentem je buď hodnota `socket.SOCK_STREAM` (protokol TCP), kterou jsme použili my, nebo hodnota `socket.SOCK_DGRAM` (protokol UDP).

Jakmile tok programu opustí oblast platnosti příkazu `with`, zavolá se metoda kontextového objektu `__exit__()`. Nestaráme se o to, zda byla či nebyla vyvolána výjimka (proto argumenty výjimky ignorujeme), a jen uzavřeme soket. Tato metoda vrací hodnotu `None` (v logickém kontextu `False`), a proto se případná výjimka propaguje dále. To je v pořádku, protože ve funkci `handle_request()` je umístěn vhodný blok `except`, který zpracuje každou výjimku soketu.

Tvorba serveru TCP

Návrhu kódu pro tvorbu serverů bývá často podobný, a proto lze místo nízkoúrovňového modulu `socket` použít vysokoúrovňový modul `socketserver`, který se postará o všechny základní operace. Nám už jen stačí přidat třídu obsluhující požadavky s metodou `handle()`, která se používá pro čtení požadavků a zápis odpovědí. Modul `socketserver` obstará komunikaci za nás tím, že obslouží požadavek každého připojení, ať už sériově nebo předáním každého požadavku jeho vlastnímu samostatnému vláknu či procesu, a to vše provádí transparentně, takže zůstaneme izolováni od nízkoúrovňových detailů.

Pro tuto aplikaci hraje roli serveru program `car_registration_server.py`.* Tento program obsahuje velice jednoduchou třídu Car, která uchovává ve formě vlastností informace o počtu sedadel, počtu najetých kilometrů a vlastníkovi (první je pouze pro čtení). Tato třída neuchovává SPZ vozů, protože vozidla jsou uložena ve slovníku a jeho klíče jsou příslušné SPZ.

Začneme pohledem na funkci `main()`, poté se stručně podíváme na způsob, jakým se načítají data serveru, pak si ukážeme tvorbu vlastní třídy představující server a nakonec se podíváme na implementaci třídy obsluhující požadavky klientů.

```
def main():
    filename = os.path.join(os.path.dirname(__file__),
                            "car_registrations.dat")
    cars = load(filename)
    print("Načteno {0} registrací vozů".format(len(cars)))
    RequestHandler.Cars = cars
```

* Při prvním spuštění serveru ve Windows se může objevit dialogové okno firewallu s hlášením, že Python je blokován – klepněte na tlačítko pro odblokování – tím umožníte serveru pracovat.

```
    server = None
    try:
        server = CarRegistrationServer(("", 9653), RequestHandler)
        server.serve_forever()
    except Exception as err:
        print("CHYBA", err)
    finally:
        if server is not None:
            server.shutdown()
            save(filename, cars)
            print("Uloženo {0} registací vozů".format(len(cars)))
```

Registrační data vozidla jsme uložili do stejného adresáře jako program. Objekt `cars` je nastaven na slovník, jehož klíči jsou řetězce SPZ a hodnotami objekty typu `Car`. Servery obvykle nic nevypisují, protože bývají spouštěny a zastavovány automaticky a běží na pozadí, a proto svůj stav většinou hlásí zápisem do souboru protokolu (např. pomocí modulu `logging`). My jsme se zde rozhodli vypisovat zprávu při spouštění a ukončování, abychom si usnadnili testování a experimentování.

Naše třída obsluhující požadavky musí mít možnost přistupovat do slovníku `cars`, který ale nemůžeme předat instanci, protože server vytváří instance za nás – pro každý požadavek jednu. Tento slovník jsme proto nastavili do proměnné třídy `RequestHandler.Cars`, kde je přístupný všem instancím.

Vytváříme instanci serveru a předáváme jí adresu a port, nad nimiž by měla pracovat, a objekt představující třídu `RequestHandler` (nejedná se o instanci této třídy). Prázdný řetězec místo adresy znamená libovolnou přístupnou adresu protokolu IPv4 (včetně aktuálního stroje, což je localhost). Poté řekneme serveru, aby donekonečna servíroval požadavky. Při ukončování serveru (jak k tomu dojde, si ukážeme za okamžik) uložíme slovník `cars`, protože data mohla být klienty změněna.

```
def load(filename):
    try:
        with contextlib.closing(gzip.open(filename, "rb")) as fh:
            return pickle.load(fh)
    except (EnvironmentError, pickle.UnpicklingError) as err:
        print("server nemůže načíst data: {0}".format(err))
        sys.exit(1)
```

Kód pro načítání je jednoduchý, protože jsme použili správce kontextu z modulu `contextlib` standardní knihovny, který zajistí, že se soubor vždy uzavře bez ohledu na to, zda dojde k nějaké výjimce. Další možností, jak docílit stejného efektu, je použít vlastního správce kontextu. Například:

```
class GzipManager:

    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
```

```
        self.fh = gzip.open(self.filename, self.mode)
        return self.fh

    def __exit__(self, *ignore):
        self.fh.close()
```

Při použití našeho správce kontextu `GzipManager` by příkaz `with` vypadal takto:

```
with GzipManager(filename, "rb") as fh:
```

Tento správce kontextu bude fungovat s kteroukoli verzí Python 3.x. Pokud nás ale zajímá pouze Python 3.1 nebo novější verze, pak můžeme jednoduše napsat `with gzip.open(...) as fh`, protože počínaje Pythonem 3.1 funkce `gzip.open()` podporuje protokol správce kontextu.

3.1

Funkce `save()` (tu jsme si neukázali) je co do struktury stejná jako funkce `load()`. Soubor však otevírá v režimu binárního zápisu, pro uložení dat používá funkci `pickle.dump()` a nevrací žádnou hodnotu.

```
class CarRegistrationServer(socketserver.ThreadingMixIn,
                            socketserver.TCPServer): pass
```

Toto je kompletní třída představující náš vlastní server. Kdybychom chtěli vytvořit server, který by místo vláken používal procesy, pak by jedinou změnou bylo to, že bychom naši třídu neodvodili od třídy `socketserver.ThreadingMixIn`, ale od třídy `socketserver.ForkingMixIn`. Výraz *příměs* (mixin) se v tomto kontextu často používá k popisu tříd, které jsou speciálně navrženy k tomu, aby byly součástí vícenásobného dědění. Třídy modulu `socketserver` lze odvozením od vhodné dvojice bázových tříd využít k tvorbě nejrůznějších serverů včetně serverů UDP a unixových serverů TCP a UDP.

Vícenásobná dědičnost ➤ 375

Všimněte si, že námi použitá příměsová třída modulu `socketserver` musí být děděna jako první. Tím bude zajištěno, že se před metodami druhé třídy přednostně použijí metody příměsové třídy, protože Python hledá metody v bázových třídách v tom pořadí, ve které jsou bázové třídy uvedeny, a použije první vhodnou metodu, kterou najde.

Soketový server vytváří pro obsluhu každého požadavku instanci zadané třídy. Naše třída `RequestHandler` nabízí pro každý druh požadavku, který dokáže zpracovat, jistou metodu plus metodu `handle()`, kterou mít musí, protože se jedná o jedinou metodu používanou soketovým serverem. Než se však podíváme na tyto metody, zastavíme se u deklarace třídy a u jejích proměnných.

```
class RequestHandler(socketserver.StreamRequestHandler):

    CarsLock = threading.Lock()
    CallLock = threading.Lock()
    Call = dict(
            GET_CAR_DETAILS=(
                    lambda self, *args: self.get_car_details(*args)),
            CHANGE_MILEAGE=(
                    lambda self, *args: self.change_mileage(*args)),
            CHANGE_OWNER=(
```

```
            lambda self, *args: self.change_owner(*args)),
    NEW_REGISTRATION=(
            lambda self, *args: self.new_registration(*args)),
    SHUTDOWN=lambda self, *args: self.shutdown(*args))
```

Vytvořili jsme podtřídu třídy `socketserver.StreamRequestHandler`, protože používáme streamovací server (TCP). Pro servery UDP je k dispozici odpovídající třída `socketserver.DatagramRequestHandler` nebo bychom pro nízkoúrovňový přístup mohli svoji třídu odvodit od třídy `socketserver.BaseRequestHandler`.

Slovník `RequestHandler.Cars` je proměnnou třídy, kterou jsme přidali ve funkci `main()` a která uchovává veškerá registrační data. Do objektů (jako jsou třídy a instance – musejí však mít slovník `__dict__`) můžeme přímo přidat další atributy i mimo třídu (v tomto případě ve funkci `main()`), což může být velice pohodlné. Vzhledem k tomu, že víme, že tato třída závisí právě na této proměnné, tak bychom mohli přiřazením `Cars = None` přidat proměnnou třídy, čímž bychom explicitně vyjádřili její existenci.

Téměř každá metoda pro zpracování požadavku potřebuje přístup k datům slovníku `Cars`, a proto musíme zajistit, aby k nim nikdy nepřistupovaly dvě metody současně (z různých vláken). Pokud by k tomu došlo, slovník by se porušil nebo by celý program havaroval. K tomuto účelu máme proměnnou třídy `lock`, pomocí níž zajistíme, že ke slovníku `Cars` bude v jednom okamžiku přistupovat pouze jedno vlákno.* (Vlákna i zámky jsme probírali v lekci 10.)

Slovník `Call` je další proměnnou třídy. Každým klíčem je název akce, kterou může server provést, a každou hodnotou funkce k provedení této akce. Tyto metody nemůžeme použít přímo jako u funkcí ve slovníku s nabídkou klienta, protože na úrovni třídy nemáme žádný objekt `self`. Tento problém jsme vyřešili vytvořením obalových funkcí, které obdrží objekt `self` při svém zavolání a které následně zavolají příslušnou metodu se zadaným objektem `self` a jakýmikoli dalšími argumenty. Další možností je vytvořit slovník `Call` až po všech metodách. To by nám umožnilo vytvářet záznamy ve tvaru `GET_CAR_DETAILS=get_car_details`, přičemž Python by byl schopen nalézt metodu `get_car_details()`, protože slovník by byl vytvořen až po definici příslušné metody. My jsme použili první přístup, poněvadž je explicitnější a není závislý na tom, kde je slovník definován.

Přestože se slovník `Call` čte až po vytvoření třídy, zabezpečili jsme jej skutečně dobře a vytvořili pro něj zámek, který zajistí, že k němu nebudou v jednom okamžiku přistupovat dvě vlákna. (Opět je třeba poznamenat, že díky zámku GIL není tento zámek v interpretu CPython ve skutečnosti potřeba.)

Zámek GIL ➢ 432

```
    def handle(self):
        SizeStruct = struct.Struct("!I")
        size_data = self.rfile.read(SizeStruct.size)
        size = SizeStruct.unpack(size_data)[0]
        data = pickle.loads(self.rfile.read(size))

        try:
            with self.CallLock:
```

* Zámek GIL (Global Interpreter Lock – globální zámek interpretu) sice zajišťuje synchronizovaný přístup k slovníku `Cars`, ale jak jsme si řekli již dříve, této skutečnosti nijak nevyužíváme, protože se jedná o implementační detail interpretu CPython.

Zámek GIL ➢ 432

```
                function = self.Call[data[0]]
            reply = function(self, *data[1:])
        except Finish:
            return
        data = pickle.dumps(reply, 3)
        self.wfile.write(SizeStruct.pack(len(data)))
        self.wfile.write(data)
```

Při každém požadavku klienta se vytvoří nové vlákno s novou instancí třídy `RequestHandler` a poté se zavolá metoda instance `handle()`. Uvnitř této metody lze data přicházející od klienta číst z objektu souboru `self.rfile` a zápisem do objektu `self.wfile` lze data posílat zpět klientovi. Oba tyto objekty poskytuje modul `socketserver` otevřené a připravené k použití.

Objekt typu `struct.Struct` je určen pro celočíselný počet bajtů, což potřebujeme pro formát „délka plus naložený objekt", který používáme pro výměnu dat mezi klienty a serverem.

Začínáme přečtením čtyř bajtů a jejich rozbalením do celého čísla `size`, abychom tak znali velikost naloženého objektu, který nám byl zaslán. Poté přečteme tento počet bajtů a vytáhneme je do proměnné `data`. Čtení bude blokovat, dokud se nenačtou všechna data. V tomto případě víme, že data budou vždy ve formě n-tice, jejíž první prvek je požadovaná akce a ostatní prvky parametry, protože právě takto vypadá protokol, na němž jsme se dohodli s klienty registrace vozidel.

Uvnitř bloku `try` vezmeme lambda funkci, která přísluší požadované akci. Přístup k slovníku `Call` chráníme zámkem, pravděpodobně jsme však až přehnaně opatrní. Jako obvykle provádíme uvnitř oboru platnosti zámku co nejméně – v tomto případě jen získáme ze slovníku odkaz na funkci. Jakmile máme funkci, tak ji zavoláme a předáme jí jako první argument objekt `self` a jako ostatní argumenty zbytek n-tice `data`. Zde provádíme volání funkce, a proto Python žádný objekt `self` nepředává. To ale vůbec nevadí, protože objekt `self` předáváme sami, přičemž uvnitř lambda funkce se zadaný objekt `self` použije pro běžné zavolání metody. Výsledkem je, že se zavolá `self.`*`metoda`*`(*data[1:])`, kde metoda odpovídá akci uvedené v prvku `data[0]`.

Je-li akcí ukončení činnosti, pak v metodě `shutdown()` vyvoláme vlastní výjimku `Finish`. V tomto případě víme, že klient neočekává žádnou odpověď, a proto provedeme jen příkaz `return`. Ovšem v případě jakékoli jiné akce naložíme výsledek volání odpovídající metody (pomocí protokolu pro nakládání objektů verze 3) a zapíšeme velikost naloženého objektu a následně samotná data naloženého objektu.

```
def get_car_details(self, license):
    with self.CarsLock:
        car = copy.copy(self.Cars.get(license, None))
    if car is not None:
        return (True, car.seats, car.mileage, car.owner)
    return (False, "Tato SPZ není registrována")
```

Tato metoda začíná pokusem o získání zámku pro data o vozidle a blokuje, dokud jej nezíská. Potom pomocí metody `dict.get()` s druhým argumentem `None` získáme vozidlo se zadanou SPZ nebo `None`. Vozidlo ihned zkopírujeme a příkaz `with` ukončíme. Tím je zajištěno, že zámek držíme jen po nejkratší možnou dobu. I když čtením se čtená data nijak nezmění, jedná se o měnitelnou kolekci, a tak

může chtít jiná metoda v jiném vlákně změnit slovník ve stejném okamžiku, ve kterém jej čteme, což se díky zámku nestane. Vně oboru platnosti zámku máme kopii objektu `car` (nebo `None`), s níž můžeme pohodlně pracovat, aniž bychom blokovali jiná vlákna.

Jako všechny metody pro zpracování akcí okolo registrace vozidel vracíme i zde n-tici, jejíž první prvek je logická hodnota signalizující úspěch či neúspěch a ostatní prvky se různí. Žádná z takovýchto metod se již dál nemusí starat o způsob vrácení dat klientovi a ani o něm nic neví, protože veškerá síťová interakce je zapouzdřena v metodě `handle()`:

```
def change_mileage(self, license, mileage):
    if mileage < 0:
        return (False, "Nemohu nastavit záporný počet km")
    with self.CarsLock:
        car = self.Cars.get(license, None)
        if car is not None:
            if car.mileage < mileage:
                car.mileage = mileage
                return (True, None)
            return (False, "Nemohu přetočit tachometr zpět")
    return (False, "Tato SPZ není registrována")
```

V této metodě můžeme provést jednu kontrolu, aniž bychom k tomu potřebovali zámek. Pokud je ale parametr `mileage` (počet najetých kilometrů) nezáporný, pak musíme získat zámek a vzít příslušné vozidlo. A pokud toto vozidlo máme (tj. pokud je zadaná SPZ platná), pak musíme zůstat uvnitř oboru platnosti zámku, kde požadovaným způsobem změníme počet najetých kilometrů nebo vrátíme n-tici s chybovou zprávou. Pokud žádné vozidlo nemá zadanou SPZ (proměnná `car` má hodnotu `None`), opustíme příkaz `with` a vrátíme n-tici s chybovou zprávou.

Možná to vypadá, že pokud bychom provedli validace na straně klienta, pak bychom mohli jistý síťový provoz zcela vyloučit a klient by v případě záporného počtu kilometrů mohl vypsat chybovou zprávu (nebo jejich zadání prostě zamezit). Třebaže klient by toto provést měl, musíme i tak na straně serveru provést kontrolu, protože nemůžeme předpokládat, že klient je zcela bez chyb. A přestože klient obdrží výchozí hodnotu pro počet najetých kilometrů vozidla, nemůžeme předpokládat, že tento údaj zadaný uživatelem je platný (a to i tehdy, kdyby byl větší než jeho současná hodnota), protože nějaký další klient by jej mohl mezitím zvýšit. Proto můžeme provést konečnou validaci pouze na serveru a pouze v oboru platnosti zámku.

Metoda `change_owner()` je velice podobná, a proto si ji zde rozebírat nebudeme.

```
def new_registration(self, license, seats, mileage, owner):
    if not license:
        return (False, "Nemohu nastavit prázdnou SPZ")
    if seats not in {2, 4, 5, 6, 7, 8, 9}:
        return (False, "Nemohu zaregistrovat vůz s neplatným počtem sedadel")
    if mileage < 0:
        return (False, "Nemohu nastavit záporný počet km")
    if not owner:
```

```
            return (False, "Nemohu nastavit nevyplněného vlastníka")
        with self.CarsLock:
            if license not in self.Cars:
                self.Cars[license] = Car(seats, mileage, owner)
                return (True, None)
        return (False, "Nemohu zaregistrovat duplicitní SPZ")
```

Opět jsme schopni provést kontroly chyb před přístupem k datům registrace, jsou-li však všechna data platná, pak musíme získat zámek. Pokud SPZ není ve slovníku `RequestHandler.Cars` (a ani by být neměla, protože nová registrace by měla mít nepoužitou SPZ), vytvoříme nový objekt typu `Car` a uložíme jej do slovníku. To vše musíme provést v oboru platnosti stejného zámku. Nesmíme totiž dovolit jiným klientům, aby v době mezi kontrolou existence SPZ ve slovníku `RequestHandler.Cars` a přidáním nového vozidla do tohoto slovníku přidali další vozidlo.

```
    def shutdown(self, *ignore):
        self.server.shutdown()
        raise Finish()
```

Je-li požadovanou akcí ukončení činnosti, pak zavoláme metodu serveru `shutdown()`, která zamezí serveru v přijímání dalších požadavků, ale nechá jej běžet, dokud neobslouží stávající požadavky. Potom vyvoláme vlastní výjimku, která upozorní metodu `handler()`, že jsme skončili. Metoda `handler()` tak vrátí provádění volajícímu, aniž by klientovi odeslala jakoukoli odpověď.

Shrnutí

V této lekci jsme si ukázali, že tvorba síťových klientů a serveru může být v Pythonu díky síťovým modulům standardní knihovny a modulům `struct` a `pickle` docela přímočará.

V první části jsme vyvinuli klientský program a dali jsme mu jedinou funkci `handle_request()`, která odesílá a přijímá libovolná naložená data na server a ze serveru pomocí generického datového formátu „délka plus naložený objekt“. V této části jsme viděli, jak s použitím tříd z modulu `socketserver` vytvořit podtřídu představující server a jak implementovat třídu obsluhující požadavky, která zpracovává požadavky klienta na straně server. Zde jsme srdce síťové interakce nacpali do jediné metody `handle()`, která dokáže od klientů přijímat a klientům odesílat libovolná data, která lze nakládat.

Moduly `socket` a `socketserver` a řada dalších modulů standardní knihovny, například `asyncore`, `asynchat` a `ssl`, poskytují mnohem více funkčnosti, než jsme si zde ukázali. Pokud však možností síťového propojení poskytované standardní knihovnou nejsou dostatečné nebo nejsou dostatečně vysokoúrovňové, pak jako možná alternativa stojí za zvážení síťový rámec třetí strany s názvem Twisted (*www.twistedmatrix.com*).

Cvičení

Cvičení zahrnují úpravu klientského a serverového programu vyvíjeného v této lekci. Tyto modifikace nevyžadují mnoho psaní, pro jejich správné zapracování je ale třeba postupovat s jistou dávkou rozvahy a pečlivosti.

1. Zkopírujte programy `car_registration_server.py` a `car_registration.py` a upravte je tak, aby si vyměňovaly data pomocí protokolu doplněného na síťové vrstvě verzí. To lze provést například tak, že se místo jednoho budou předávat dvě celá čísla ve struktuře (délka, verze protokolu).

   K tomu je potřeba přidat nebo upravit kolem deseti řádků ve funkci `handle_request()` v klientském programu a přidat či upravit kolem šestnácti řádků v metodě `handle()` serverového programu, což zahrnuje kód pro ošetření případů, kdy přečtená verze protokolu neodpovídá očekávané verzi.

   Řešení tohoto a následujících cvičení je k dispozici v souborech `car_registration_ans.py` a `car_registration_server_ans.py`.

2. Zkopírujte program `car_registration_server.py` (nebo použijte jeho verzi z prvního cvičení) a upravte jej tak, aby nabízel novou akci `GET_LICENSES_STARTING_WITH` (načti SPZ začínající zadaným řetězcem). Tato akce by měla přijímat jediný řetězcový parametr. Metoda implementující tuto akci by měla vždy vracet n-tici se dvěma prvky (`True`, seznam SPZ). Neexistuje zde totiž chybový případ (`False`), protože nenalezení odpovídajících SPZ není chyba, ale vede jednoduše k vrácení hodnoty `True` a prázdného seznamu.

   Značky SPZ (klíče slovníku `RequestHandler.Cars`) načítejte v rámci oboru platnosti zámku, veškerou ostatní činnost ale provádějte mimo zámek, abyste minimalizovali blokování. Jeden z efektivních způsobů pro nalezení odpovídajících SPZ spočívá v seřazení klíčů a v následném použití modulu `bisect` pro vyhledání první odpovídající SPZ, od níž pak postupným průchodem najdete další. Další možností je procházet všechny značky SPZ a vybírat ty, které začínají zadaným řetězcem, k čemuž lze využít například seznamovou komprehenzi.

   Kromě dodatečného příkazu `import` bude slovník `Call` potřebovat několik řádků navíc pro novou akci. Metodu implementující tuto akci lze napsat na méně než deset řádků. Není to obtížné, je ale třeba postupovat s rozmyslem. Řešení, které používá modul `bisect`, je k dispozici v souboru `car_registration_server _ans.py`.

3. Zkopírujte program `car_registration.py` (nebo použijte jeho verzi z prvního cvičení) a upravte jej tak, aby využíval výhody nového serveru (`car_registration_server_ans.py`). To zahrnuje změnu funkce `retrieve_car_details()`, která po zadání neplatné SPZ vyzve k zadání začátku SPZ a poté načte seznam, z něhož si uživatel vybere. Zde je ukázková interakce s použitím této nové funkce (server již běží a text zadaný uživatelem je zvýrazněn):

```
(V)ůz Upravit K(i)lometry  Upravit V(l)astníka  (N)ový vůz
(Z)astavit server (K)onec [v]:
SPZ: 3m7 4020
SPZ: 3M7 4020
Počet sedadel: 2
Počet najetých km: 97181
Vlastník: Martin Charouz
(V)ůz  Upravit K(i)lometry  Upravit V(l)astníka  (N)ový vůz
(Z)astavit server (K)onec [v]:
License [3M7 4020]: z
Tato SPZ není registrována
```

```
Začátek SPZ: z
Žádná SPZ nezačíná na Z
Začátek SPZ: 2
(1) 1Z1 5711
(2) 1Z7 4791
(3) 1M2 3035
Zadejte volbu (0 pro zrušení): 3
SPZ: 1M2 3035
Počet sedadel: 5
Počet najetých km: 17719
Vlastník: Antonín Janek
```

Tato změna zahrnuje vymazání jednoho řádku a přidání asi dvaceti dalších řádků. Je trošku záludná, protože uživatel musí mít v každé fázi možnost pokračovat nebo odejít. Nezapomeňte novou funkčnost otestovat pro všechny případy (žádná SPZ nezačíná zadaným řetězcem, jedna SPZ začíná zadaným řetězcem a dvě či více SPZ začínají zadaným řetězcem). Řešení je k dispozici v souboru `car_registration_ans.py`.

LEKCE 12

Programování databází

V této lekci:

- Databáze DBM
- Databáze SQL

Většina softwarových vývojářů si pod pojmem *databáze* obvykle představí *relační databázový systém* (RDBMS – Relational Database Management System). Tyto systémy používají tabulky (mřížky podobné kalkulačním tabulkám) obsahující řádky přirovnávané k záznamům a sloupce přirovnávané k polím. Tyto tabulky a data, jež uchovávají, se vytvářejí a ovládají pomocí příkazů zapsaných v jazyku SQL (Structured Query Language – strukturovaný dotazovací jazyk). Python nabízí rozhraní API (Application Programming Interface – aplikační programovací rozhraní) pro práci s databázemi SQL a běžně se dodává s databází SQLite 3 jako standardem.

Dalším typem databáze je *správce databáze* (DBM – Database Manager), jenž uchovává libovolný počet prvků ve tvaru klíč-hodnota. Standardní knihovna Pythonu se dodává s rozhraními pro několik správců databáze, včetně několika specifických pro Unix. Správce databáze fungují podobně jako slovníky jazyka Python, ovšem s tím, že obvykle nejsou uloženy v paměti, ale na disku a jejich klíče a hodnoty jsou vždy objekty typu `bytes` s případným omezením délky. Modul `shelve`, kterému se budeme věnovat v první části této lekce, nabízí pohodlné rozhraní pro správce databáze, které dovoluje používat řetězcové klíče a objekty (které lze nakládat) jako hodnoty.

Pokud by dostupné správce databáze a databáze SQLite nestačily, pak je zde ještě seznam balíčků pro jazyk Python (viz *pypi.python.org/pypi*), který obsahuje velké množství balíčků pro práci s databázemi, mezi něž patří správce databáze `bsddb` („Berkeley DB“) a rozhraní k oblíbeným databázím typu klient-server, jako jsou například databáze DB2, Informix, Ingres, MySQL, ODBC a PostgreSQL.

Použití databází SQL vyžaduje znalost jazyka SQL a práci s řetězci příkazů jazyka SQL. To je vhodné pro někoho, kdo má zkušenosti s jazykem SQL, stylu jazyka Python se to však příliš nepodobá. Existuje ještě jeden způsob, jak pracovat s databázemi SQL – pomocí mapovače ORM (Object Relational Mapper – mapovač objektových a relačních dat). V Pythonu jsou k dispozici dva z nejoblíbenějších mapovačů ORM jako knihovny třetích stran – jedná se o knihovny SQLAlchemy (*www.sqlalchemy.org*) a SQLObject (*www.sqlobject.org*). Díky mapovači ORM nemusíme používat příkazy jazyka SQL, ale známou syntaxi jazyka Python (vytváření objektů a volání metod).

V této lekci implementujeme dvě verze programu, který udržuje seznam nosičů DVD a u každého sleduje název, rok vydání, délku v minutách a režiséra. První verze používá pro ukládání dat správce databáze (přes modul `shelve`) a druhá používá databázi SQLite. Oba programy dokážou též načítat a ukládat jednoduchý formát XML, díky čemuž lze například data o nosičích DVD z jednoho programu exportovat a importovat je do druhého. Verze založená na databázi SQL nabízí ve srovnání s verzí postavenou na správci databáze o něco bohatší funkčnosti a obsahuje trošku čistší datový návrh.

Databáze DBM

Typ `bytes` ➤ 286

Modul `shelve` zabaluje správce databáze do jakéhosi obalu, díky němuž s ním můžeme pracovat, jako by se jednalo o slovník, ovšem za předpokladu, že používáme pouze řetězcové klíče a hodnoty, které lze nakládat (pomocí modulu `pickle`). Modul `shelve` pak v pozadí převádí klíče a hodnoty na objekty typu `bytes` a zpět.

Modul `shelve` použije nejlepšího aktuálně dostupného správce databáze. Je tedy možné, že soubor uložený správcem databáze na jednom stroji nebude možné na jiném stroji načíst, pokud druhý stroj nemá stejný správce databáze. Řešením je doplnit pro soubory, které musejí být přenositelné mezi jednotlivými stroji, import a export pro jazyk XML, což jsme provedli i v programu `dvds-dbm.py`.

Pro klíče používáme názvy nosičů DVD a pro hodnoty n-tice uchovávající režiséra, rok a délku. Díky modulu `shelve` nemusíme provádět žádné převody dat a můžeme s objektem správce databáze pracovat jako s obyčejným slovníkem.

Struktura programu je podobná interaktivním, nabídkou řízeným programům, s nimiž jsme se již setkali, a proto se zaměříme pouze na ty aspekty, které přímo souvisejí s programování správce databáze. Níže je uveden úryvek z funkce `main()` našeho programu bez kódu pro obsluhu nabídky.

```
db = None
try:
    db = shelve.open(filename, protocol=pickle.HIGHEST_PROTOCOL)
    ...
finally:
    if db is not None:
        db.close()
```

Zde otevíráme (nebo vytváříme, pokud dosud neexistuje) zadaný soubor správce databáze pro čtení i zápis. Hodnota každého prvku se ukládá jako naložený objekt pomocí uvedeného protokolu. Stávající prvky lze číst i tehdy, pokud byly uloženy s použitím nižšího protokolu, protože Python dokáže zjistit správný protokol pro čtení naložených objektů. Na konci správce databáze zavíráme, čímž se vymaže jeho interní mezipaměť, zajistí se, aby diskový soubor odrážel jakékoli provedené změny, a tento soubor se uzavře.

Program nabízí volby pro přidání, úpravu, výpis, odstranění, import a export dat o nosičích DVD. Importování dat z formátu XML a jejich exportování do formátu XML přeskočíme, poněvadž postup je velice podobný tomu, který jsme probírali v lekci 7. Kromě přidávání vynecháme většinu kódu souvisejícího s uživatelským rozhraním – důvodem je opět to, že jsme se s ním již v jiných situacích setkali.

```
def add_dvd(db):
    title = Console.get_string("Název", "název")
    if not title:
        return
    director = Console.get_string("Režisér", "režisér")
    if not director:
        return
    year = Console.get_integer("Rok", "rok", minimum=1896,
                               maximum=datetime.date.today().year)
    duration = Console.get_integer("Délka (v minutách)", "minuty",
                                   minimum=0, maximum=60*48)
    db[title] = (director, year, duration)
    db.sync()
```

Tato funkce, stejně jako všechny ostatní funkce volané nabídkou programu, přijímá jako svůj jediný parametr objekt (`db`) představující správce databáze. Ve větší části funkce se zabýváme získáváním údajů o nosiči DVD a na předposledním řádku ukládáme prvek klíč-hodnota do souboru správce

databáze, přičemž klíčem je název DVD a hodnotou režisér, rok a délka (společně naložené modulem `shelve`).

Kvůli souladu s obvyklým používáním Pythonu nabízí správce databáze stejné rozhraní API jako slovníky, a tudíž se nemusíme učit žádnou novou syntaxi, tedy až na funkci `shelve.open()`, kterou jsme viděli již dříve, a na metodu `shelve.Shelf.sync()`, která se používá pro vymazání interní mezipaměti a synchronizaci dat souboru s provedenými změnami – v tomto případě se jedná o přidání nového prvku.

```
def edit_dvd(db):
    old_title = find_dvd(db, "edit")
    if old_title is None:
        return
    title = Console.get_string("Název", "název", old_title)
    if not title:
        return
    director, year, duration = db[old_title]
    ...
    db[title] = (director, year, duration)
    if title != old_title:
        del db[old_title]
    db.sync()
```

Pro úpravu údaje o nosiči DVD musí uživatel nejdříve zvolit nosič DVD, s nímž chce pracovat. Stačí zadat jen název, protože názvy se používají jako klíče, přičemž hodnoty uchovávají další údaje. Tato funkčnost je potřebná i na jiných místech programu (např. při odstraňování nosiče DVD), a proto jsme ji vyčlenili do samostatné funkce `find_dvd()`, na kterou se podíváme vzápětí. Je-li nosič DVD nalezen, musí uživatel zadat požadované změny jako výchozí se použijí stávající hodnoty, čímž se celá interakce značně urychlí. (Většinu kódu této funkce pro uživatelské rozhraní jsme vypustili, protože je téměř stejný jako při přidávání nosiče DVD.) Na konci ukládáme data stejně, jako při přidávání. Pokud název nebyl změněn, přepíšeme příslušnou hodnotu, a pokud se název liší, tak vytvoříme nový prvek klíč-hodnota a původní vymažeme.

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    while True:
        matches = []
        start = Console.get_string(message, "název")
        if not start:
            return None
        for title in db:
            if title.lower().startswith(start.lower()):
                matches.append(title)
        if len(matches) == 0:
            print("Nemohu najít žádné nosiče DVD začínající na", start)
            continue
        elif len(matches) == 1:
```

```
            return matches[0]
        elif len(matches) > DISPLAY_LIMIT:
            print("Na {0} začíná příliš mnoho nosičů DVD; zkuste zadat "
                  "větší část názvů".format(start))
            continue
        else:
            matches = sorted(matches, key=str.lower)
            for i, match in enumerate(matches):
                print("{0}: {1}".format(i + 1, match))
            which = Console.get_integer("Číslo (nebo 0 pro zrušení)",
                            "číslo", minimum=1, maximum=len(matches))
            return matches[which - 1] if which != 0 else None
```

Aby bylo vyhledání nosiče DVD co nejrychlejší a nejjednodušší, vyžadujeme od uživatele zadání pouze jednoho nebo několika počátečních znaků názvu. Jakmile máme začátek názvu, procházíme správcem databáze a vytváříme seznam shodujících se prvků. Najdeme-li jeden prvek, tak jej vrátíme, a pokud jich najdeme více (ale méně než `DISPLAY_LIMIT`, což je celé číslo definované v jiné části programu), tak je všechny seřadíme bez ohledu na velikost písmen a zobrazíme s čísly, aby si uživatel mohl vybrat název zadáním příslušného čísla. (Funkce `Console.get_integer()` přijímá hodnotu 0 i tehdy, je-li minimum větší než nula, a proto můžeme hodnotu 0 použít pro zrušení výběru. Toto chování může být vypnuto zadáním argumentu `allow_zero=False`. Pro zrušení výběru nemůžeme použít samotnou klávesu Enter (tj. bez zadání hodnoty), protože nezadání hodnoty znamená přijetí hodnoty výchozí.)

```
def list_dvds(db):
    start = ""
    if len(db) > DISPLAY_LIMIT:
        start = Console.get_string("Vypsat ty, které začínají na "
                                   "[Enter=vše]", "start")
    print()
    for title in sorted(db, key=str.lower):
        if not start or title.lower().startswith(start.lower()):
            director, year, duration = db[title]
            print("{title} ({year}) {duration} minut, režie "
                  "{director}".format(**locals()))
```

Výpis všech nosičů DVD (nebo těch, které začínají určitým podřetězcem) je jen záležitostí průchodu před prvky správce databáze.

```
def remove_dvd(db):
    title = find_dvd(db, "remove")
    if title is None:
        return
    ans = Console.get_bool("Odstranit {0}?".format(title), "ne")
    if ans:
        del db[title]
        db.sync()
```

K odstranění nosiče DVD stačí najít uživatelem požadovaný záznam a v případě kladného potvrzení od uživatele jej ze správce databáze vymazat.

Nyní již tedy víme, jak pomocí modulu `shelve` otevřít (nebo vytvořit) soubor správce databáze, jak do něj přidat prvky a jak jeho prvky upravit, procházet a odstranit.

Naneštěstí má náš datový návrh jednu vadu. Názvy režisérů jsou duplikované, což by mohlo snadno vést k nekonzistencím. Například režisér Daniel DeVito může být u jednoho filmu zadán jako „Danny De Vito" a u jiného jako „Danny deVito". Řešením by bylo vytvořit dva soubory správce databáze: hlavní soubor nosičů DVD s klíči ve formě názvů a hodnotami ve tvaru (rok, délka, identifikace režiséra) a soubor režisérů s klíči ve formě identifikací režisérů (tj. celých čísel) a hodnotami obsahujícími jména režisérů. Tento nedostatek vyřešíme v následující části, kde vytvoříme verzi našeho programu pro databázi SQL, která bude používat dvě tabulky, jednu pro nosiče DVD a druhou pro režiséry.

Databáze SQL

Rozhraní k většině oblíbených databází SQL jsou dostupná jako moduly třetích stran. Samotný Python se dodává s modulem `sqlite3` (a s databází SQLite 3), takže lze okamžitě začít s programováním databáze. SQLite je odlehčená databáze SQL postrádající řadu funkcí, které má například databáze PostgreSQL. Velmi se ale hodí k prototypování a pro spoustu situací může být naprosto dostatečná.

Pro maximální usnadnění přechodu mezi jednotlivými databázemi nabízí dokument PEP 249 (Python Database API Specification v2.0 – specifikace rozhraní API Pythonu pro databáze verze 2.0) specifikaci rozhraní API s názvem DB-API 2.0, kterou musejí databázová rozhraní dodržovat. Tuto specifikaci splňuje například modul `sqlite3`, což ale nelze říci o všech modulech třetích stran. Toto rozhraní API specifikuje dva hlavní objekty, objekt `connection` a objekt `cursor`, přičemž rozhraní API, která musejí podporovat, jsou uvedena v tabulkách 12.1 a 12.2. Objekty `connection` a `cursor` modulu `sqlite3` poskytují kromě atributů a metod vyžadovaných specifikací DB-API 2.0 ještě mnoho dalších.

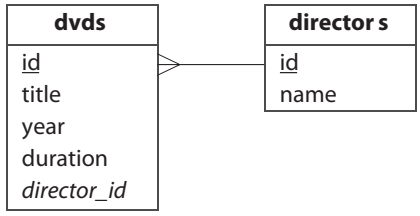
Tabulka 12.1: Metody objektu connection dle specifikace DB-API 2.0

Syntaxe	Popis
db.close()	Uzavře připojení k databázi (reprezentované objektem `db`, který získáme zavoláním funkce `connect()`).
db.commit()	Zapíše jakékoli nevyřízené transakce do databáze. U databází, které nepodporují transakce, nic neprovede.
db.cursor()	Vrátí objekt představující databázový kurzor, skrze něhož lze provádět dotazy.
db.rollback()	Vrátí jakoukoli nevyřízenou transakci do stavu, který existoval před jejím začátkem. U databází, které nepodporují transakce, neprovede nic.

Tabulka 12.2: Atributy a metody objektu cursor dle specifikace DB-API 2.0

Syntaxe	Popis
c.arraysize	Počet řádků (lze číst i zapisovat), které vrátí metoda `fetchmany()`, není-li zadána velikost.
c.close()	Uzavře kurzor `c`, což se provede automaticky, když opustí svůj obor platnosti.
c.description	Posloupnost určená pouze pro čtení, jejíž prvky jsou n-tice se sedmi prvky `(name, type_code, display_size, internal_size, precision, scale, null_ok)` popisující jednotlivé sloupce kurzoru `c`.
c.execute(*`sql`*, *`parametry`*)	Provede dotaz jazyka SQL obsažený v řetězci *`sql`* a přitom nahradí každý zástupný symbol odpovídajícím parametrem z posloupnosti či mapování *`parametry`*, je-li zadáno.
c.executemany(*`sql`*, *`posl_parametrů`*)	Provede dotaz jazyka SQL pro každý prvek v posloupnosti *`posl_parametrů`* obsahující posloupnosti či mapování. Tato metoda by se neměla používat pro operace, které vytvářejí výsledné sady (jako jsou např. příkazy `SELECT`).
c.fetchall()	Vrátí posloupnost všech řádků, které dosud nebyly načteny (může se jednat o všechny řádky).
c.fetchmany(*`velikost`*)	Vrátí posloupnost řádků (kde samotné řádky jsou posloupnosti). Výchozí hodnota parametru *`velikost`* je `c.arraysize`.
c.fetchone()	Vrátí následující řádek výsledné sady dotazu jako posloupnost nebo hodnotu `None`, jsou-li výsledky vyčerpány. Vyvolá výjimku, pokud žádná výsledná sada neexistuje.
c.rowcount	Počet řádků (pouze pro čtení) pro poslední operaci (např. `SELECT`, `INSERT`, `UPDATE` nebo `DELETE`) nebo -1, není-li tento údaj k dispozici nebo použitelný.

Verze našeho programu pro nosiče DVD pracující s databází SQL nese název `dvds-sql.py`. Program uchovává režiséry a údaje o nosičích DVD odděleně, aby se neuchovávaly duplicitní údaje, a dále nabízí jednu volbu nabídky navíc, která umožňuje uživateli vypsat režiséry. Tyto dvě tabulky jsou znázorněné na obrázku 12.1. Program má necelých 300 řádků, zatímco program `dvds-dbm.py` z předchozí verze měl bezmála 200 řádků. Největší rozdíl spočívá v tom, že místo jednoduchých operací se slovníkem musíme používat dotazy jazyka SQL a že při prvním spuštění programu musíme vytvořit databázové tabulky.



Obrázek 12.1: Návrh databáze programu pro nosiče DVD

Funkce `main()` je podobná svému protějšku v předchozí verzi programu, tentokráte však voláme pro připojení vlastní funkci `connect()`.

```
def connect(filename):
    create = not os.path.exists(filename)
    db = sqlite3.connect(filename)
    if create:
        cursor = db.cursor()
        cursor.execute("CREATE TABLE directors ("
            "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
            "name TEXT UNIQUE NOT NULL)")
        cursor.execute("CREATE TABLE dvds ("
            "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
            "title TEXT NOT NULL, "
            "year INTEGER NOT NULL, "
            "duration INTEGER NOT NULL, "
            "director_id INTEGER NOT NULL, "
            "FOREIGN KEY (director_id) REFERENCES directors)")
        db.commit()
    return db
```

Funkce `sqlite3.connect()` otevře zadaný databázový soubor, který v případě neexistence vytvoří, a vrací objekt reprezentující tuto databázi. Vzhledem k tomu jsme si před zavoláním této funkce poznamenali, zda se databáze bude vytvářet zcela od začátku, protože je-li tomu tak, pak musíme vytvořit tabulky, o něž se náš program opírá. Veškeré dotazy se provádějí prostřednictvím databázového kurzoru dostupného z metody `cursor()` objektu, který představuje databázi.

Všimněte si, že obě tabulky jsme vytvořili s identifikačním polem, které má omezení AUTOINCREMENT, což znamená, že databázový systém SQLite automaticky naplní tato identifikační pole jedinečnými čísly, takže je nemusíme při vkládání nových záznamů vyplňovat sami.

Databázový systém SQLite podporuje omezený soubor datových typů (v podstatě jen logické hodnoty, čísla a řetězce), které lze ale dále rozšířit pomocí datových „adaptérů“, které jsou buď předdefinované (např. pro kalendářní data a časy), nebo naše vlastní, jejichž prostřednictvím můžeme reprezentovat libovolné datové typy. Program pro nosiče DVD tyto možnosti nepotřebuje, pokud by je však vyžadoval, pak stačí nahlédnout do dokumentace k modulu `sqlite3`. Námi použitá syntaxe pro cizí klíče nemusí být stejná jako v případě jiných databází. V každém případě má pouze informativní

charakter, protože databázový systém SQLite na rozdíl od mnoha jiných nevynucuje relační integritu. (Nicméně databázový systém SQLite nabízí řešení na bázi příkazu `.genfkey` modulu `sqlite3`.) Dalším charakteristickým znakem modulu sqlite3 je to, že jeho výchozím chováním je podpora implicitních transakcí, a proto zde není žádná metoda pro explicitní „spuštění transakce“.

```
def add_dvd(db):
    title = Console.get_string("Název", "název")
    if not title:
        return
    director = Console.get_string("Režisér", "režisér")
    if not director:
        return
    year = Console.get_integer("Rok", "rok", minimum=1896,
                               maximum=datetime.date.today().year)
    duration = Console.get_integer("Délka (v minutách)", "minuty",
                                   minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor = db.cursor()
    cursor.execute("INSERT INTO dvds "
                   "(title, year, duration, director_id) "
                   "VALUES (?, ?, ?, ?)",
                   (title, year, duration, director_id))
    db.commit()
```

Tato funkce začíná stejným kódem jako její protějšek z programu `dvds-dbm.py`, po shromáždění údajů od uživatele je však zcela jiná. Režisér zadaný uživatelem může, ale také nemusí být obsažen v tabulce `directors`, a proto máme k dispozici funkci `get_and_set_director()`, která vloží režiséra, není-li ještě v databázi, a v každém případě vrátí identifikační číslo režiséra připravené pro vložení do tabulky `dvds`. Se všemi připravenými daty spustíme příkaz `INSERT` jazyka SQL. Identifikační číslo záznamu uvádět nemusíme, protože databázový systém SQLite jej automaticky přidá za nás.

V dotazu jsme pro zástupné symboly použili otazníky. Každý otazník je nahrazen odpovídající hodnotou v posloupnosti, která následuje za řetězcem obsahujícím příkaz jazyka SQL. Můžeme použít také pojmenované zástupné symboly, o čemž se přesvědčíme, až se budeme věnovat úpravě záznamů. Přestože zástupné symboly lze zcela vynechat a umístit data přímo do řetězce s kódem jazyka SQL, my doporučujeme zástupné symboly vždy používat, a přenechat tak obtíže spojené se správným zakódováním a náležitou přípravou datových prvků databázovému modulu. Další výhoda zástupných symbolů spočívá v tom, že zvyšují bezpečnost, protože zamezují zlovolnému vložení libovolného kódu jazyka SQL do dotazu.

```
def get_and_set_director(db, director):
    director_id = get_director_id(db, director)
    if director_id is not None:
        return director_id
    cursor = db.cursor()
    cursor.execute("INSERT INTO directors (name) VALUES (?)",
                   (director,))
    db.commit()
```

```
    return get_director_id(db, director)
```

Tato funkce vrací identifikační číslo zadaného režiséra a v případě potřeby jej vloží do databáze. Je-li záznam v databázi, pak jeho identifikační číslo získáme pomocí funkce `get_director_id()`, kterou zkoušíme zavolat jako první možnost.

```
def get_director_id(db, director):
    cursor = db.cursor()
    cursor.execute("SELECT id FROM directors WHERE name=?",
                   (director,))
    fields = cursor.fetchone()
    return fields[0] if fields is not None else None
```

Funkce `get_director_id()` vrací identifikační číslo zadaného režiséra nebo hodnotu `None`, pokud v databázi neexistuje. Používáme metodu `fetchone()`, protože nalezený záznam je buď jeden, nebo žádný. (Víme, že duplicitní režiséry nemáme, protože pole `name` tabulky `directors` má omezení `UNIQUE` a navíc před přidáním nového režiséra vždy kontrolujeme, zda se již v databázi nenachází.) Metody ve tvaru `fetch*()` vracejí vždy posloupnost polí (nebo `None`, nejsou-li k dispozici žádné další záznamy), a to i tehdy, pokud (jako zde) požádáme o načtení jediného pole.

```
def edit_dvd(db):
    title, identity = find_dvd(db, "edit")
    if title is None:
        return
    title = Console.get_string("Název", "název", title)
    if not title:
        return
    cursor = db.cursor()
    cursor.execute("SELECT dvds.year, dvds.duration, directors.name "
                   "FROM dvds, directors "
                   "WHERE dvds.director_id = directors.id AND "
                   "dvds.id=:id", dict(id=identity))
    year, duration, director = cursor.fetchone()
    director = Console.get_string("Režisér", "režisér", director)
    if not director:
        return
    year = Console.get_integer("Rok", "rok", year, 1896,
                               datetime.date.today().year)
    duration = Console.get_integer("Délka (v minutách)", "minuty",
                                   duration, minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor.execute("UPDATE dvds SET title=:title, year=:year, "
                   "duration=:duration, director_id=:director_id "
                   "WHERE id=:identity", locals())
    db.commit()
```

Pro úpravu záznamu o nosiči DVD musíme nejdříve najít záznam, s nímž chce uživatel pracovat. Je-li záznam nalezen, pak začneme tím, že dáme uživateli možnost změnit název. Pak získáme ostatní pole, abychom mohli nabídnout jejich stávající hodnoty jako výchozí, čímž minimalizujeme množství textu, který musí uživatel zadávat, protože k přijetí výchozí hodnoty stačí, když stiskne klávesu Enter. Zde jsme použili pojmenované zástupné symboly (ve tvaru `:název`), a proto musíme doplnit odpovídající hodnoty pomocí mapování. Pro příkaz `SELECT` jsme použili čerstvě vytvořený slovník a pro příkaz `UPDATE` jsme použili slovník vrácený funkcí `locas()`.

Pro oba příkazy bychom mohli použít čerstvý slovník, takže bychom příkazu `UPDATE` místo `locals()` předali `dict(title=title, year=year, duration=duration, director_id=director_id` a `id=identity))`.

Jakmile máme všechna pole a uživatel zadal jakékoli požadované změny, vezmeme identifikační číslo odpovídajícího režiséra (nový režisér se vloží do databáze, je-li to nutné) a poté aktualizujeme databázi novými daty. Vydali jsme se po nejjednodušší cestě a aktualizujeme všechna pole záznamu, a ne jen ta, které byly skutečně změněna.

Při práci se souborem správce databáze jsme jako klíč použili název nosiče DVD, takže pokud se název změnil, vytvořili jsme nový prvek klíč-hodnota a původní vymazali. Avšak zde má každý záznam o nosiči DVD jedinečné identifikační číslo, které se nastavuje při prvním vložení záznamu, a proto můžeme volně měnit hodnotu libovolného pole bez nutnosti provádět cokoliv dalšího.

```
def find_dvd(db, message):
    message = "(Začátek) názvu " + message
    cursor = db.cursor()
    while True:
        start = Console.get_string(message, "název")
        if not start:
            return (None, None)
        cursor.execute("SELECT title, id FROM dvds "
                       "WHERE title LIKE ? ORDER BY title",
                       (start + "%",))
        records = cursor.fetchall()
        if len(records) == 0:
            print("Nemohu najít žádné nosiče DVD začínající na", start)
            continue
        elif len(records) == 1:
            return records[0]
        elif len(records) > DISPLAY_LIMIT:
            print("Na {0} začíná příliš mnoho nosičů DVD ({1}); zkuste zadat "
                  "větší část názvu".format(start, len(records)))
            continue
        else:
            for i, record in enumerate(records):
                print("{0}: {1}".format(i + 1, record[0]))
            which = Console.get_integer("Číslo (nebo 0 pro zrušení)",
                                        "číslo", minimum=1, maximum=len(matches))
            return records[which - 1] if which != 0 else (None, None)
```

Tato funkce provádí stejnou službu jako funkce `find_dvd()` v programu `dvdsdbm.py` a vrací n-tici se dvěma prvky (název, identifikační číslo nosiče DVD) nebo (`None`, `None`) podle toho, zda byl záznam nalezen. Místo procházení všech dat jsme použili operátor jazyka SQL pro zástupné symboly (`%`), takže se načtou pouze relevantní záznamy.

A protože očekáváme, že počet nalezených záznamů bude malý, načteme je všechny naráz do posloupnosti tvořené posloupnostmi. Pokud bylo nalezeno více odpovídajících záznamů, jejichž počet však nepřekračuje stanovený limit pro zobrazení, pak je vypíšeme společně s čísly u každého z nich, aby si uživatel mohl jeden vybrat podobně jako v programu `dvds-dbm.py`.

```
def list_dvds(db):
    cursor = db.cursor()
    sql = ("SELECT dvds.title, dvds.year, dvds.duration, "
           "directors.name FROM dvds, directors "
           "WHERE dvds.director_id = directors.id")
    start = None
    if dvd_count(db) > DISPLAY_LIMIT:
        start = Console.get_string("Vypsat ty, které začínají na "
                                   "[Enter=vše]", "start")
        sql += " AND dvds.title LIKE ?"
    sql += " ORDER BY dvds.title"
    print()
    if start is None:
        cursor.execute(sql)
    else:
        cursor.execute(sql, (start + "%",))
    for record in cursor:
        print("{0[0]} ({0[1]}) {0[2]} minut, režie {0[3]}".format(
              record))
```

Pro výpis údajů o každém nosiči DVD provedeme dotaz `SELECT`, který spojí dvě tabulky, a ke klauzuli `WHERE` přidáme druhý element, je-li v databázi více záznamů (což zjistíme pomocí funkce `dvd_count()`), než kolik stanoví limit pro jejich zobrazení. Potom dotaz provedeme a projdeme výsledky. Každý záznam je posloupnost, jejíž pole odpovídají polím v dotazu `SELECT`.

```
def dvd_count(db):
    cursor = db.cursor()
    cursor.execute("SELECT COUNT(*) FROM dvds")
    return cursor.fetchone()[0]
```

Tyto řádky jsme vyčlenili do samostatné funkce, protože je potřebujeme v několika různých funkcích.

Kód pro funkci `list_directors()` jsme vynechali, protože je svou strukturou velice podobný funkci `list_dvds()`, je ale jednodušší, neboť vypisuje pouze jedno pole (jméno).

```
def remove_dvd(db):
    title, identity = find_dvd(db, "remove")
    if title is None:
        return
    ans = Console.get_bool("Odstranit {0}?".format(title), "ne")
    if ans:
        cursor = db.cursor()
        cursor.execute("DELETE FROM dvds WHERE id=?", (identity,))
        db.commit()
```

Tato funkce se volá v okamžiku, kdy uživatel požádá o vymazání záznamu. Je velice podobná svému protějšku v programu `dvds-dbm.py`.

Prohlídka programu `dvds-sql.py` je u konce. Viděli jsme, jak vytvořit databázové tabulky, jak vybírat záznamy, procházet vybrané záznamy a jak záznamy vkládat, aktualizovat a mazat. Pomocí metody `execute()` můžeme provést libovolný příkaz jazyka SQL, který podporuje použitá databáze.

Databázový systém SQLite nabízí mnohem více funkčních prvků, než kolik jsme jich využili v našem programu. Patří mezi ně režim automatického potvrzování transakci (a další druhy řízení transakcí) a možnost vytvářet funkce, které lze provádět uvnitř dotazů jazyka SQL. Dále je možné vytvořit tovární funkce, které řídí, co má být pro každý načtený záznam vráceno (např. slovník nebo vlastní typ místo posloupnosti polí). Kromě toho lze zadáním „:memory:“ jako názvu souboru vytvářet databáze SQLite umístěné v paměti.

Shrnutí

V lekci 7 jsme viděli několik různých způsobů ukládání a načítání dat z disku a v této lekci jsme si ukázali, jak pracovat s datovými typy, jež neuchovávají svá data v paměti, ale na disku.

Pro soubory správce databáze se nejlépe hodí modul `shelve`, poněvadž uchovává prvky ve tvaru řetězec-objekt. Chceme-li kompletní kontrolu, pak samozřejmě můžeme sáhnout přímo po libovolném správci databáze. Pěknou vlastností modulu `shelve` a správců databáze obecně je to, že používají rozhraní API slovníku, díky čemuž je získávání, přidávání, upravování a odstraňování prvků pro správu databáze velice snadné, stejně jako úprava programů, které používají slovník. Nevýhoda správců databáze tkví v tom, že v případě relačních dat musíme pro každou tabulku klíč-hodnota použít samostatný soubor správce databáze, kdežto databázový systém SQLite ukládá všechna data do jediného souboru.

V rámci databází SQL se databázový systém SQLite nejlépe hodí pro prototypování a jeho výhodou je to, že je dodáván společně s Pythonem jako standard. Viděli jsme, jak pomocí funkce `connect()` získat objekt představující databázi a jak pomocí metody `execute()` databázového kurzoru provádět příkazy jazyka SQL (např. `CREATE TABLE`, `SELECT`, `INSERT`, `UPDATE` a `DELETE`).

Python nabízí ucelený soubor datových úložišť pro disk i paměť počínaje binárními soubory, textovými soubory, soubory XML a naloženými objekty až po správce databáze a databáze SQL. Díky tomu je možné v každé situaci zvolit nejvhodnější přístup.

Cvičení

Napište interaktivní konzolový program pro správu seznamu záložek. Pro každou záložku uchovávejte dva údaje: adresu URL a název. Zde je ukázkový běh programu:

```
Záložky (bookmarks.dbm)
(1) Programming in Python 3........ http://www.qtrac.eu/py3book.html
(2) PyQt........................... http://www.riverbankcomputing.com
(3) Python......................... http://www.python.org
(4) Qtrac Ltd...................... http://www.qtrac.eu
(5) Scientific Tools for Python.... http://www.scipy.org

(P)řidat (U)pravit (V)ypsat (O)dstranit (K)onec [v]: u
Počet záložek pro úpravu: 2
URL [http://www.riverbankcomputing.com]:
Name [PyQt]: PyQt (Python bindings for GUI library)
```

Program by měl uživateli umožnit přidávat, upravovat, vypisovat a odstraňovat záložky. Aby byla identifikace záložek pro úpravu a odstraňování co nejjednodušší, vypisujte záložky s čísly a požádejte uživatele o zadání čísla záložky, kterou chce upravit či odstranit. Data ukládejte do souboru správce databáze pomocí modulu `shelve` – pro klíče použijte názvy a pro hodnoty adresy URL. Z hlediska struktury se tento program velice podobá programu `dvds-dbm.py`. Výjimkou je funkce `find_bookmark()`, která je mnohem jednodušší než funkce `find_dvd()`, protože jí stačí získat od uživatele celé číslo, které použije pro vyhledání názvu odpovídající záložky.

Dále doplňte pomůcku pro uživatele, kteří zapomenout uvést protokol. V takovém případě přidejte před přidávanou či upravovanou adresu URL „http://“.

Celý program lze napsat na méně než 100 řádků (použijeme-li funkce `Console.get_string()` a podobné z modulu `Console`). Řešení je k dispozici v souboru `bookmarks.py`.

LEKCE 13

Regulární výrazy

V této lekci:

- Jazyk Pythonu pro regulární výrazy
- Modul pro regulární výrazy

Regulární výraz je kompaktní notace pro reprezentaci kolekce řetězců. Co jej ale činí tak mocným, je skutečnost, že jediný regulární výraz dokáže reprezentovat neomezené množství řetězců, které splňují požadavky tohoto regulárních výrazu. Regulární výrazy se definují pomocí minijazyka, který je zcela jiný než jazyk Python. Nicméně Python nabízí modul `re`, jehož prostřednictvím můžeme regulární výrazy bez problémů vytvářet a používat.*

Regulární výrazy se používají především v následujících pěti oblastech:

- **Analýza textu**: identifikování a extrahování částí textu, jež odpovídají určitým kritériím. Regulární výrazy se používají pro vytváření ad hoc analyzátorů a využívají je též tradiční nástroje pro analýzu textu.
- **Vyhledávání**: lokalizace podřetězců, které mohou mít více než jednu formu (např. chceme vyhledat libovolný podřetězec „pet.png“, „pet.jpg“, „pet.jpeg“ nebo „pet.svg“, přitom ale nechceme podřetězce „carpet.png“ a podobné).
- **Vyhledávání s nahrazováním**: nahrazení v celém textu, kdy regulární výraz vyhledává určitý řetězec (např. chceme vyhledat „jízdní kolo“ nebo „lidmi poháněné vozidlo“ a nahradit je řetězcem „kolo“).
- **Rozdělování řetězců**: rozdělení řetězce na každém místě vyhledaném regulárním výrazem (např. rozdělení na každém místě, kde se nachází dvojtečka nebo rovnítko („:“ nebo „=“)).
- **Validace**: kontrola, zda část textu splňuje určitá kritéria (např. zda za číslicemi obsahuje symbol měny).

Analyzování souborů XML ➢ 303

Regulární výrazy používané pro vyhledávání, rozdělování a validaci jsou často poměrně malé a srozumitelné, díky čemuž jsou právě pro tyto účely naprosto ideální. Nicméně i když se regulární výrazy často a úspěšně používají ke tvorbě analyzátorů, mají v této oblasti jistá omezení. Dokážou totiž pracovat pouze s takovým rekurzivně strukturovaným textem, jehož maximální úroveň rekurze je známa. Rovněž rozsáhlé a složité regulární výrazy mohou být obtížně čitelné a udržovatelné. Pro analyzování je tedy kromě jednoduchých případů nejvhodnější sáhnout po nástrojích navržených pro tento účel (například pro kód jazyka XML použijeme specializovaný analyzátor kódu jazyka XML). Není-li takový analyzátor k dispozici, pak můžeme kromě regulárních výrazů použít generický nástroj pro analýzu, což je přístup, jemuž se budeme věnovat v lekci 14.

Ve své nejjednodušší formě je regulární výraz výrazem (např. znakovým literálem), za nímž volitelně následuje kvantifikátor. Složitější regulární výrazy sestávají z libovolného počtu kvantifikovaných výrazů, mohou obsahovat aserce a mohou být ovlivněny příznaky.

V první části této lekce si představíme a vysvětlíme všechny klíčové pojmy regulárních výrazů a ukážeme si čistou syntaxi regulárního výrazu (což má se samotným Pythonem jen pramálo společného). Potom si ve druhé části ukážeme, jak použít regulární výrazy při programování v jazyku Python. Budeme vycházet z látky probrané v předchozích částech. Čtenáři znalí regulárních výrazů, kteří se chtějí jen dozvědět, jak je použít v jazyku Python, mohou přeskočit do druhé části (na stranu 479). V této lekci se budeme věnovat celému jazyku regulárních výrazů, který poskytuje modul

* Kvalitní knihou o regulárních výrazech je kniha „*Mastering Regular Expressions*“, jejímž autorem je Jeffrey E. F. Friedl. Nenajdete v ní sice zmínku o Pythonu, ale modul Pythonu s názvem re nabízí velmi podobnou funkčnost jako subsystém Perlu pro regulární výrazy, kterému se kniha věnuje.

re, včetně všech asercí a příznaků. Regulární výrazy v textu zvýrazníme **tučným písmem**, nalezené shody budou podtržené a zachycené skupiny zvýrazníme šedým podkladem.

Jazyk Pythonu pro regulární výrazy

V této části se ve čtyřech oddílech podíváme na jazyk regulárních výrazů. V prvním oddílu si ukážeme, jak hledat shody s jednotlivými znaky nebo se skupinou znaků (např. shoda s „a“, shoda s „b“ nebo shoda s „a“ nebo „b“). Ve druhém oddílu si ukážeme, jak shody kvantifikovat (např. právě jedna shoda, nejméně jedna shoda nebo maximálně možný počet shodných nálezů). Ve třetím oddílu si ukážeme, jak seskupovat podvýrazy a jak zachytávat nalezený text, a v posledním oddílu si ukážeme, jak pomocí asercí a příznaků tohoto jazyka ovlivnit způsob, jakým regulární výrazy pracují.

Znaky a třídy znaků

Nejjednoduššími výrazy jsou znakové literály (např. `a` nebo `5`), a pokud není explicitně zadán žádný kvantifikátor, pak představují „shodu s jedním výskytem“. Kupříkladu regulární výraz `pila` sestává ze čtyř výrazů, z nichž každý je implicitně kvantifikovaný pro jednu shodu, takže se shoduje s jedním „p“ následovaným jedním „i“ následovaným jedním „l“ následovaným jedním „a“, a proto odpovídá řetězcům `pila` a `nepila`.

Přestože většinu znaků lze použít jako literály, některé znaky jsou „speciální“. Jedná se o symboly v jazyku regulárních výrazů, a proto je nutné při jejich použití jako literálu před nimi uvádět zpětné lomítko (`\`). Těmito speciálními znaky jsou `\.^$?+*{}[]()|`. V regulárních výrazech lze též použít většinu speciálních řetězcových posloupností jazyka Python (např. `\n` pro nový řádek a `\t` pro tabulátor) včetně speciálních symbolů pro šestnáctkové hodnoty ve tvaru `\x`*HH*, `\u`*HHHH* a `\U`*HHHHHHHH*.

Speciální řetězcové posloupnosti ➤ 71

V řadě případů nám nestačí shoda s jedním konkrétním znakem, ale chceme shodu s libovolným znakem z určité množiny znaků. Toho lze docílit pomocí *tříd znaků* (character class), což je jeden či více znaků uzavřených do hranatých závorek. (Nemá to nic společného s třídami jazyka Python. Jedná se o prostý pojem ze světa regulárních výrazů pro „množinu znaků“.) Třída znaků je výraz, který se stejně jako ostatní výrazy bez explicitní kvantifikace shoduje s jedním znakem (což může být libovolný ze znaků v dané třídě znaků). Například regulární výraz `m[oa]p` se shoduje s `mop` a `map`a, ale ne s `moap`. Podobně pro nalezení shody s jedinou číslicí můžeme použít regulární výraz `[0123456789]`. Rozsah znaků můžeme stanovit pomocí pomlčky, takže s číslicí se shoduje také regulární výraz `[0-9]`. Význam třídy znaků můžeme také negovat, k čemuž stačí za otevírající hranatou závorku umístit stříšku, takže `[^0-9]` se shoduje s libovolným znakem, který není číslicí.

Všimněte si, že uvnitř třídy znaků ztrácejí speciální znaky kromě zpětného lomítka svůj význam. Na druhou stranu stříška (`^`) zde dostává nový význam (negace), je-li prvním znakem ve třídě znaků, protože na jiném místě se jedná o obyčejný literál. Dále pomlčka značí rozsah znaků, pokud ovšem není na prvním místě. Pak totiž znamená literál pomlčky.

Některé množiny znaků se používají velice často, a proto je k dispozici několik zkratek, které uvádí tabulka 13.1. S jedinou výjimkou lze tyto zkratky použít uvnitř tříd znaků, takže kupříkladu regulární výraz `[\dA-Fa-f]` odpovídá libovolné šestnáctkové číslici. Výjimkou je tečka (`.`), což je zkratka mimo třídu znaků, ale ve třídě znaků odpovídá literálu tečky.

Význam příznaků ➤ 476

Tabulka 13.1: Zkraty pro třídy znaků

Symbol	Význam
.	Odpovídá libovolnému znaku kromě znaku nového řádku nebo s příznakem `re.DOTALL` jakémukoliv znaku bez výjimky. Uvnitř třídy znaků odpovídá literálu tečky (`.`).
\d	Odpovídá číslici ze znakové sady Unicode.
\D	Odpovídá znaku ze znakové sady Unicode, který není číslicí.
\s	Odpovídá znaku, který ve znakové sadě Unicode představuje bílé místo.
\S	Odpovídá znaku, který ve znakové sadě Unicode nepředstavuje bílé místo.
\w	Odpovídá znaku, který ve znakové sadě Unicode představuje „slovo“.
\W	Odpovídá znaku, který ve znakové sadě Unicode nepředstavuje „slovo“.

Kvantifikátory

Kvantifikátor má tvar `{`*m*`,`*n*`}`, kde *m* a *n* je minimum a maximum pro počet shod s výrazem, na něhož se daný kvantifikátor aplikuje. Například oba regulární výrazy `e{1,1}e{1,1}` a `e{2,2}` se shodují s `fee`l, ale neshodují se s `felt`.

Psaní kvantifikátoru za každým výrazem by bylo únavné a jistě ne moc dobře čitelné. Naštěstí jazyk regulárních výrazů podporuje několik pohodlných zkratek. Je-li v kvantifikátoru uvedeno pouze jedno číslo, pak se bere jako minimum i maximum, takže `e{2}` je totéž, jako `e{2,2}`. A jak jsme si řekli v předchozí části, není-li zadán žádný kvantifikátor, pak se má za to, že jeho hodnota je jedna (tj. `{1,1}` nebo `{1}`). Proto `ee` je totéž jako `e{1,1}e{1,1}` a `e{1}e{1}`, takže regulární výrazy `e{2}` a `ee` se shodují s `fee`l, ale ne s `felt`.

Často se používá odlišná hodnota pro minimum a maximum. Například pro nalezení shody s `travelled` a `traveled` můžeme použít buď `travel{1,2}ed`, nebo `travell{0,1}ed`. Kvantifikace `{0,1}` se používá tak často, že má svoji vlastní zkratku (`?`), takže tento regulární výraz můžeme napsat také jako `travell?ed` (tuto variantu bychom nejspíše použili v praxi).

K dispozici jsou ještě další dvě kvantifikační zkratky: `+` znamená `{1,`*n*`}` („alespoň jeden výskyt“) a `*` znamená `{0,`*n*`}` („libovolný počet výskytů“). V obou případech je *n* maximální možný počet povolený pro kvantifikátor, což je obvykle 32 767. Všechny kvantifikátory uvádí tabulka 13.2.

Tabulka 13.2: Kvantifikátory regulárních výrazů

Syntaxe	Význam
e`?` nebo *e*`{0,1}`	Hladově se shoduje s žádným nebo jedním výskytem výrazu *e*.
e`??` nebo *e*`{0,1}?`	Nehladově se shoduje s žádným nebo jedním výskytem výrazu *e*.
e`+` nebo *e*`{1,}`	Hladově se shoduje s jedním nebo více výskyty výrazu *e*.
e`+?` nebo *e*`{1,}?`	Nehladově se shoduje s jedním nebo více výskyty výrazu *e*.
e`*` nebo *e*`{0,}`	Hladově se shoduje s žádným nebo více výskyty výrazu *e*.
e`*?` nebo *e*`{0,}?`	Nehladově se shoduje s žádným nebo více výskyty výrazu *e*.

Syntaxe	Význam
e{*m*}	Shoduje se přesně s *m* výskyty výrazu *e*.
e{*m*,}	Hladově se shoduje s alespoň *m* výskyty výrazu *e*.
e{*m*,}?	Nehladově se shoduje s alespoň *m* výskyty výrazu *e*.
e{,*n*}	Hladově se shoduje s nejvýše *n* výskyty výrazu *e*.
e{,*n*}?	Nehladově se shoduje s nejvýše *n* výskyty výrazu *e*.
e{*m*,*n*}	Hladově se shoduje s alespoň *m* a s nejvýše *n* výskyty výrazu *e*.
e{*m*,*n*}?	Nehladově se shoduje s alespoň *m* a s nejvýše *n* výskyty výrazu *e*.

Kvantifikátor + je velice užitečný. Například pro nalezení shody s celými čísly můžeme použít `\d+`, což odpovídá jedné nebo více číslicím. Tento regulární výraz by mohl v řetězci `4588.91` nalézt shodu na dvou místech, například `4588`.`91` a `4588`.`91`. Někdy dochází k překlepům, které jsou výsledkem příliš dlouhého přidržení nějaké klávesy. Pomocí regulárního výrazu `bevel+ed` bychom mohli nalézt shodu pro `beveled` a `bevelled`, ale i pro `bevellled`. Pokud bychom chtěli nalézt shodu pouze pro výskyty s jedním a více písmeny „l“, pak bychom použili regulární výraz `bevell+ed`.

Kvantifikátor * je o něco méně užitečný, což je dáno prostě tím, že může často vést k neočekávaným výsledkům. Představte si, že chceme v souborech Pythonu najít řádky, jež obsahují komentáře, takže se můžeme pokusit hledat `#*`. Avšak tento regulární výraz se shoduje naprosto s libovolným řádkem, včetně prázdných řádků, protože ve skutečnosti znamená „najdi shodu s libovolným počtem znaků #“, což tedy zahrnuje i řádky bez znaku #. Nováčci ve světě regulárních výrazů by kvantifikátor * neměli vůbec používat, a pokud jej přece jen použijí (nebo pokud použijí ?), pak se musejí ujistit, že v regulárním výrazu je minimálně ještě jeden výraz s nenulovým kvantifikátorem (tedy alespoň jeden kvantifikátor jiný než * nebo ?), protože oba kvantifikátory * a ? mohou odpovídat nulovému počtu výskytů.

Kvantifikátor * lze často převést na + a naopak. Například pro shodu se slovem „tasselled“ s alespoň jedním „l“ můžeme použít `tassell*ed` nebo `tassel+ed`, a pro shodu se dvěma a více „l“ použijeme `tasselll*ed` nebo `tassell+ed`.

Pokud použijeme regulární výraz `\d+`, pak obdržíme shodu s řetězcem `136`. Proč ale dojde ke shodě se všemi číslicemi, a ne jen s první číslicí? Všechny kvantifikátory jsou *hladové* (greedy), takže se shodují s tolika znaky, s kolika mohou. Jakýkoliv kvantifikátor lze převést na nehladový (označovaný též jako *minimální*) tím, že za něj doplníme symbol ?. (Otazník má tedy dva odlišné významy – sám o sobě je zkratkou pro kvantifikátor `{0,1}` a za kvantifikátorem říká kvantifikátoru, aby nebyl hladový.) Například regulární výraz `\d+?` se může shodovat s řetězcem `136` na třech různých místech: `1`36, `13`6 a `136`. Zde je další příklad: regulární výraz `\d??` se neshoduje s žádnou nebo se shoduje s jedinou číslicí, preferuje však shodu s nulovým počtem číslic, poněvadž není hladový – sám o sobě trpí stejným problémem jako kvantifikátor *, protože se může shodovat s ničím, což znamená s naprosto žádným textem.

Nehladové kvantifikátory mohou být užitečné pro rychlé a ne úplně čisté analyzování kódu jazyků XML a HTML. Například pro shodu se všemi obrázkovými značkami nebude zápis `<img.*>` (shoda s jedním „<“, poté s jedním „m“, pak s jedním „g“, pak s žádným či více libovolnými znaky kromě znaku nového řádku, pak s jedním „>“) fungovat, protože část `.*` je hladová a bude se shodovat se

vším včetně uzavíracího znaku značky (`>`) a bude pokračovat dál, dokud nenarazí na poslední znak `>` v celém textu.

Můžeme uvažovat o třech řešeních (kromě řádného analyzátoru). Jedním je regulární výraz `<img[^>]*>` (shoda s řetězcem `<img`, poté s libovolnými znaky kromě znaku `>` a pak s uzavíracím znakem `>`), druhým je `<img.*?>` (shoda s řetězcem `<img`, poté s libovolným počtem znaků, ovšem nehladově, takže se zastaví těsně před uzavíracím znakem `>` a pak s uzavíracím znakem `>`) a třetí řešení kombinuje obě předchozí jako `<img[^>]*?>`. Přesto není žádné z nich správné, protože všechna se mohou shodovat s řetězcem `<img>`, což nechceme. Vzhledem k tomu, že víme, že značka `<image>` musí mít atribut `src`, můžeme napsat přesnější regulární výraz `<img\s+[^>]*?src=\w+[^>]*?>`. Ten se shoduje s literály znaků `<img`, poté s jedním či více znaky představujícími bílé místo, poté nehladově s nula či více libovolnými znaky kromě `>` (pro přeskočení ostatních atributů, jako je `alt`), poté s atributem `src` (literály znaků `src=` následované alespoň jedním znakem představujícím „slovo“) a poté s libovolnými znaky kromě `>` (včetně žádných znaků), což zahrnuje případné další atributy, a nakonec s uzavíracím znakem `>`.

Seskupování a zachytávání

V praktických aplikacích často potřebujeme regulární výrazy, které se shodují s kteroukoliv ze dvou či více alternativ, a často potřebujeme tuto shodu nebo její část zachytit pro další zpracování. Někdy také chceme, aby se kvantifikátor aplikoval na několik výrazů. To vše lze realizovat seskupováním pomocí `()` a v případě alternativ alternací pomocí `|`.

Alternace je zvláště užitečná v situaci, kdy chceme shodu s kteroukoliv z několika zcela odlišných alternativ. Například regulární výraz `letadlo|letoun|tryskáč` se bude shodovat s kterýmkoliv textem, jenž obsahuje „letadlo“ nebo „letoun“ nebo „tryskáč“. Stejného výsledku lze docílit pomocí regulárního výrazu `let(adlo|oun)|tryskáč`. Závorky zde používáme pro seskupení výrazů, takže zde máme dva vnější výrazy `let(adlo|oun)` a `tryskáč`. První z nich obsahuje vnitřní výraz `adlo|oun`, a protože je před ním umístěn výraz `let`, shoduje se první vnější výraz pouze s řetězcem „letadlo“ nebo „letoun“.

Závorky slouží dvěma různým cílům. Můžeme je použít pro seskupení výrazů a také pro zachycení textu, který odpovídá danému výrazu. Pojem *skupina* budeme používat pro označení seskupeného výrazu nehledě na to, zda je zachytáván či nikoliv, a pomocí pojmů *zachycení* a *zachycovaná skupina* budeme označovat zachycenou skupinu. Regulární výraz `(letadlo|letoun|tryskáč)` se nejen shoduje s kterýmkoli ze tří výrazů, ale též zachytí pro pozdější zpracování, se kterým z nich došlo k shodě. Podívejme se pro srovnání na regulární výraz `(let(adlo|oun)|tryskáč)`, který v případě shody prvního výrazu obsahuje dvě zachycení („letadlo“ nebo „letoun“ jako první zachycení a „adlo“ nebo „out“ jako druhé zachycení) a v případě shody druhého výrazu jedno zachycení („tryskáč“). Zachycování můžeme vypnout umístěním znaků `?:` za otevírací závorku, takže například regulární výraz `(let(?:adlo|oun)|tryskáč)` bude mít v případě shody pouze jedno zachycení („letadlo“ nebo „letoun“ nebo „tryskáč“).

Seskupený výraz je sám o sobě výrazem, a proto jej lze kvantifikovat. Podobně jako u jakéhokoliv jiného výrazu je i zde v případě nezadání explicitní kvantity předpokládán jeden výskyt. Pokud jsme například přečetli textový soubor s řádky ve formě *`klíč=hodnota`*, kde klíč je alfanumerický, pak se regulární výraz `(\w+)=(.+)` bude shodovat s každým řádkem, který má neprázdný klíč a neprázd-

nou hodnotu. (Vzpomeňte si, že `.` odpovídá čemukoliv kromě znaků nového řádku.) A pro každý odpovídající řádek se provedou dvě zachycení, z nichž první je klíč a druhé hodnota.

Tento regulární výraz se bude například shodovat s celým řádkem `topic= physical geography` (obě zachycení jsou zvýrazněna). Všimněte si, že druhé zachycení obsahuje nějaké bílé místo a že bílé místo před rovnítkem není přijato. Regulární výraz bychom tedy mohli upravit tak, aby byl při přijímání bílého místa flexibilnější a nechtěné bílé místo ořízl:

```
[ \t]*(\w+)[ \t]*=[ \t]*(.+)
```

Tento regulární výraz se shoduje se stejným řádkem jako jeho předchozí verze a také s řádky, které mají kolem rovnítka bílé místo, ovšem s tím, že první zachycení nemá na začátku ani na konci bílé místo a druhé zachycení nemá bílé místo na začátku. Například: `topic = physical geography`. Dbali jsme na to, aby se části shodující se s bílým místem nacházely mimo zachycovací závorky a aby byly povolené také řádky, jež nemají vůbec žádné bílé místo. Pro shodu s bílým místem jsme nepoužili `\s`, protože tím by docházelo ke shodě se znaky nového řádku (`\n`), což by mohlo vést k nesprávným shodám zasahujícím do více řádků (např. při použití příznaku `re.MULTILINE`). A pro hodnotu jsme nepoužili `\S` pro shodu s nebílým místem, protože chceme povolit hodnoty, jež obsahují bílé místo (např. anglické nebo české věty). Aby druhé zachycení nemělo ukončující bílé místo, museli bychom použít sofistikovanější regulární výraz. Ukážeme si jej v následujícím oddílu.

Příznaky regulárních výrazů ➤ 480

Na zachycení se můžeme odkazovat pomocí *zpětných odkazů* (backreferences), které odkazují zpět na předchozí zachycované skupiny.* Jedna syntaxe pro zpětné odkazy uvnitř samotných regulárních výrazů má tvar `\`*`i`*, kde *i* je číslo zachycení. Zachycení jsou číslována od jedné a zleva doprava se u každé nové levé (zachycovací) závorky toto číslo o jedničku zvyšuje. Například pro zjednodušené shody duplicitních slov můžeme použít regulární výraz `(\w+)\s+\1`, který se shoduje se „slovem“, pak s alespoň jednou mezerou a poté se stejným slovem, které bylo zachyceno. (Číslo zachycení 0 se vytváří automaticky bez nutnosti přidávat závorky. Uchovává celou shodu, tj. to, co si ukazujeme jako podtržené.) Sofistikovanější způsob pro nalezení duplicitních slov si ukážeme později.

V dlouhých nebo komplikovaných regulárních výrazech je často pohodlnější použít pro zachycení místo čísel názvy. Tímto způsobem lze též usnadnit údržbu, poněvadž přidávání či odstraňování zachycovacích závorek může změnit čísla, na jména to však nemá žádný vliv. Pro pojmenování zachycení je nutné za otevírací závorku zapsat `?P<`*`název`*`>`. Například regulární výraz `(?P<key>\w+)=(?P<value>.+)` obsahuje dvě zachycení s názvem `"key"` a `"value"`. Syntaxe pro zpětné odkazy na pojmenovaná zachycení uvnitř regulárního výrazu má tvar `(?P=`*`název`*`)`. Kupříkladu regulární výraz `(?P<word>\w+)\s+(?P=word)` se shoduje s duplicitními slovy a používá zachycení s názvem `"word"`.

Aserce a příznaky

Jeden problém, jenž ovlivňuje spoustu regulárních výrazů, které jsme si zatím ukázali, spočívá v tom, že se mohou shodovat s více texty nebo s odlišným textem, než o jakém jsme uvažovali. Například regulární výraz `letadlo|letoun|tryskáč` se bude shodovat s texty „madlo“, „okoun“ i „tryskáč“. Tento problém můžeme vyřešit pomocí asercí. Aserce se neshoduje s žádným textem, ale místo toho říká něco o textu v místě, kde se daná aserce v regulárním výrazu se vyskytuje.

* Je třeba poznamenat, že uvnitř tříd znaků, tj. uvnitř `[]`, lze použít pouze číslované zpětné odkazy.

Jednou z asercí je `\b` (hranice slova), která tvrdí, že znak, který ji předchází, musí být „slovo“ (`\w`) a znak, který následuje za ní, nemusí být slovo (`\W`), nebo naopak. Například regulární výraz `let` se může v textu `let malým letadlem` shodovat dvakrát (`let` `malým` `let`adlem), ale regulární výraz `\bletb` se bude shodovat pouze jednou (`let` `malým letadlem`). Původní regulární výraz bychom tedy mohli zapsat buď jako `\bletadlo\b|\bletoun\b|\btryskáč\b`, nebo srozumitelněji jako `\b(?:letadlo|letoun|tryskáč)\b` (hranice slova, nezachycovaný výraz, hranice slova).

Tabulka 13.3: Aserce regulárních výrazů

Příznaky regulárních výrazů ➢ 480

Symbol	Význam
`^`	Dává shodu na začátku a také po každém znaku nového řádku v případě nastaveného příznaku `re.MULTILINE`.
`$`	Dává shodu na konci a také před každým znakem nového řádku v případě nastaveného příznaku `re.MULTILINE`.
`\A`	Dává shodu na začátku.
`\b`	Dává shodu na hranici „slova“. Tato aserce je ovlivněna příznakem `re.ASCII`. Uvnitř třídy znaků se jedná o symbol pro znak zpětného lomítka.
`\B`	Dává shodu na hranici „neslova“. Tato aserce je ovlivněna příznakem `re.ASCII`.
`\Z`	Dává shodu na konci.
`(?=`*e*`)`	Dává shodu, pokud se výraz *e* shoduje v místě této aserce, ale nepostupuje přes něj – označuje se jako *pohled dopředu* (lookahead) nebo *pozitivní pohled dopředu* (positive lookahead).
`(?!`*e*`)`	Dává shodu, pokud se výraz *e* neshoduje v místě této aserce, ale nepostupuje přes něj – označuje se jako *negativní pohled dopředu* (negative lookahead).
`(?<=`*e*`)`	Dává shodu, pokud se výraz *e* shoduje bezprostředně před touto asercí – označuje se jako *pozitivní pohled dozadu* (positive lookbehind).
`(?<!`*e*`)`	Dává shodu, pokud se výraz *e* neshoduje bezprostředně před touto asercí – označuje se jako *negativní pohled dozadu* (negative lookbehind).

Podporována je řada dalších asercí, jejichž přehled uvádí tabulka 13.3. Aserce můžeme použít pro zlepšení srozumitelnosti regulárního výrazu pro řádky *klíč=hodnota*, který můžeme změnit například na `^(\w+)=([^\n]+)` a nastavit příkaz `re.MULTILINE`, který zajistí, že každá dvojice *klíč=hodnota* se vezme z jediného řádku bez možnosti přesahu na více řádků – ovšem za předpokladu, že žádná část regulárního výrazu se neshoduje se znakem nového řádku, takže nemůžeme použít třeba `\s`. (Příznaky jsou uvedeny v tabulce 13.5 – viz stranu 480 – jejich syntaxe je popsána na konci tohoto oddílu a příklady jsou uvedeny v následující části.) A pokud chceme odříznout bílé místo z konců a použít pojmenovaná zachycení, bude regulární výraz vypadat takto:

```
^[ \t]*(?P<key>\w+)[ \t]*=[ \t]*(?P<value>[^\n]+)(?<![ \t])
```

Příznaky regulárních výrazů ➢ 480

Přestože je tento regulární výraz navržen pro poměrně jednoduchý úkol, vypadá docela komplikovaně. Můžeme jej ale doplnit o komentáře, aby se lépe udržoval. Toho lze docílit přidáním vložených komentářů pomocí syntaxe `(?#`*komentář*`)`, ovšem v praxi může být kvůli takovýmto komentářům celý regulární výraz ještě hůře čitelný. Mnohem lepším řešením je použít příznak `re.VERBOSE`, díky kterému můžeme v regulárním výrazu volně používat znaky představující bílé místo a běžné komentáře jazyka

Python. Jediným omezením je, že pokud potřebujeme shodu s bílým místem, pak musíme použít buď `\s`, nebo třídu znaků, jako je `[ ]`. Zde je regulární výraz pro řádky *klíč=hodnota* s komentáři:

```
^[ \t]*               # začátek řádku a případné úvodní bílé místo
(?P<key>\w+)          # text klíče
[ \t]*=[ \t]*         # rovnítko s případným obklopujícím bílým místem
(?P<value>[^\n]+)     # text hodnoty
(?<![ \t])            # negativní pohled dozatu kvůli koncovému bílému místu
```

V programech napsaných v jazyku Python bychom normálně napsali takovýto regulární výraz do holého řetězce uzavřeného do trojitých uvozovek. Díky holému řetězci totiž nemusí zdvojovat zpětná lomítka a díky trojitým uvozovkám jej můžeme rozložit na více řádků.

Holé řetězce ➤ 72

Kromě asercí, které jsme dosud probírali, existují ještě další aserce, které se dívají na text za (nebo před) asercí, zda se shoduje (či neshoduje) se zadaným výrazem. Výrazy, které lze použít v asercích s pohledem dozadu, musejí mít fixní délku (nemůžeme tedy použít kvantifikátory `?`, `+` a `*` a číselné kvantifikátory musejí mít fixní velikost, například `{3}`).

V případě regulárního výrazu pro řádky klíč=hodnota znamená aserce s negativním pohledem dozadu, že v místě jejího výskytu nesmí být předchozím znakem mezera nebo tabulátor. To nám zajistí, že posledním znakem zachyceným do zachycované skupiny `"value"` nebude mezera ani tabulátor (přesto tím ale nezabráníme, aby se mezery či tabulátory objevily uvnitř zachyceného textu).

Podívejme se na další příklad. Představte si, že čteme víceřádkový text, jenž obsahuje jména „Helena Patricie Sharman“, „Mario Sharman“, „Sharman Josef“, „Helena Králová“ a tak dále, a že chceme hledat shodu se jménem „Helena Sharman“, ale jen pokud odkazuje na jméno „Helena Patricie Sharman“. Nejjednodušší je použít regulární výraz `\b(Helena\s+Patricie)\s+Sharman\b`. Stejného výsledku bychom mohli docílit také pomocí aserce s pohledem dopředu, například `\b(Helena\s+Patricie)(?=\s+Sharman\b)`. Tento regulární výraz se bude shodovat s textem „Helena Patricie“ pouze tehdy, pokud je před ním hranice slova a za ním bílé místo a text „Sharman“ končící na hranici slova.

Pro zachycení určitých variací křestních jmen („Helena“, „Helen P.“ nebo „Helena Patricie“) můžeme vytvořit sofistikovanější regulární výraz, například `\b(Helena(?:\s+(?:P\.|Patricie))?)\s+(?=Sharman\b)`. Tento regulární výraz se shoduje s hranicí slova následovanou jednou z forem křestního jména, ovšem jen tehdy, je-li následováno bílým místem, textem „Sharman“ a hranicí slova.

Všimněte si, že zachytávání provádějí pouze dvě syntaxe: `(`*e*`)` a `(?P<`*název*`>`*e*`)`. Žádná jiná forma závorek nezpůsobí zachycení. To perfektně koresponduje s asercemi s pohledem dopředu i dozadu, protože tyto aserce uvádějí pouze určité tvrzení o tom, co je následuje nebo předchází, přičemž samy nejsou součástí shody, ale spíše ovlivňují, zda k nějaké shodě vůbec dojde. Krásně sem zapadají také poslední dvě formy závorek, kterým se nyní budeme věnovat.

Již dříve jsme viděli, jak se můžeme číslem (např. `\1`) nebo názvem (např. `(?P=`*název*`)`) odkazovat na zachycení uvnitř regulárního výrazu. Ke shodě může docházet také podmíněně podle toho, zda došlo k nějaké dřívější shodě. V takovém případě se používají syntaxe `(?(`*id*`)`*výraz_ano*`)` a `(?(`*id*`)`*výraz_ano*`|`*výraz_ne*`)`. Část *id* představuje název nebo číslo dřívějšího zachycení, na které se odkazujeme.

Je-li zachycení úspěšné, použije se shoda s výrazem *výraz_ano*. Pokud se zachycení neuskuteční, tak se použije shoda s výrazem *výraz_ne*, je-li zadán.

Podívejme se na další příklad. Představte si, že chceme extrahovat názvy souborů, na které se odkazuje atribut `src` ve značkách `<img>` souborů HTML. Začneme hledáním shody s atributem `src`, ovšem na rozdíl od dřívějšího pokusu budeme počítat se třemi formami, které tento atribut může mít: jednoduché uvozovky, dvojité uvozovky a bez uvozovek. Zde je první pokus: `src=(["'])([^"'>]+)\1`. Část `([^"'>]+)` zachytí hladovou shodu s nejméně jedním znakem, který není uvozovkou ani `>`. Tento regulární výraz funguje dobře v případě názvů souborů uzavřených do uvozovek a díky odkazu `\1` dochází ke shodě pouze tehdy, jsou-li otevírací i uzavírací uvozovky stejné. Jenže názvy souborů bez uvozovek nezachytíme. Tento problém napravíme tak, že otevírací uvozovku učiníme volitelnou, takže ke shodě bude docházet jen v případě její přítomnosti.

Zde je revidovaná verze regulárního výrazu: `src=(["'])?([^"'>]+)(?(1)\1)`. Výraz *výraz_ne* jsme neuvedli, protože bez uvozovek nemusíme hledat žádnou jinou shodu. Tahle verze ale naneštěstí nefunguje úplně správně. Pro názvy souborů uzavřené do uvozovek funguje pěkně, ale pro názvy souborů bez uvozovek funguje jen v případě, kdy je atribut `src` posledním atributem ve značce. V opačném případě se totiž nesprávně shoduje s textem následujícího atributu. Řešením je zpracovat tyto dva případy (s uvozovkami a bez nich) samostatně a použít alternaci: `src=((["'])([^\1>]+?)\1|([^"' >]+))`. Podívejme se nyní na tento regulární výraz v kontextu s pojmenovanými skupinami, se závorkami neprovádějícími shodu a s komentáři:

```
<img\s+                     # začátek značky
[^>]*?                      # libovolné atributy před atributem src
src=                        # začátek atributu src
(?:
    (?P<quote>["'])         # otevírací uvozovka
    (?P<qimage>[^\1>]+?)    # název souboru obrázku
    (?P=quote)              # uzavírací uvozovka odpovídající otevírací uvozovce
|                           # ---nebo alternativně---
    (?P<uimage>[^"' >]+)    # název souboru obrázku bez uvozovek
)
[^>]*?                      # libovolné atributy za atributem src
>                           # konec značky
```

Odsazení nemá jiný význam než zlepšení čitelnosti. Nezachycovací závorky používáme pro alternaci. První alternativa se shoduje s uvozovkou (jednoduchou či dvojitou), poté s názvem souboru obrázku (který může obsahovat libovolné znaky kromě nalezené uvozovky nebo >) a nakonec s další uvozovkou, který musí být stejná jako ta předchozí. Pro název souboru jsme museli použít minimální hledání shody (`+?`), abychom zajistili, že shoda nepřekročí první odpovídající uzavírací uvozovku. To znamená, že shoda s názvem souboru `"I'm here!.png"` proběhne správně. Všimněte si také, že jsme pro odkazování na odpovídající uvozovku museli uvnitř třídy znaků místo `(?P=quote)` použít číslovaný zpětný odkaz (`\1`), protože uvnitř třídy znaků fungují pouze číslované zpětné odkazy. Druhá alternativa se shoduje s názvem souboru bez uvozovek, což je řetězec znaků, které neobsahují uvozovky, mezery ani `>`. Kvůli alternaci se název souboru zachytí do zachycení `"qimage"` (číslo zachycení 2) nebo `"uimage"` (číslo zachycení 3, protože `(?P=quote)` sice provede shodu, ale ne zachycení), a proto musíme zkontrolovat obě.

Finální částí syntaxe regulárních výrazů, kterou nabízí subsystém Pythonu pro regulární výrazy, je prostředek pro nastavování příznaků. Příznaky se obvykle nastavují jejich předáním jakožto dodatečných parametrů při volání funkce `re.compile()`, někdy je ale pohodlnější nastavit je jako součást samotného regulárního výrazu. K tomu se používá syntaxe `(?`*příznaky*`)`, kde *příznaky* představují jedno či více písmen `a` (stejné jako předání hodnoty `re.ASCII`), `i` (`re.IGNORECASE`), `m` (`re.MULTILINE`), `s` (`re.DOTALL`) a `x` (`re.VERBOSE`).* Jsou-li příznaky nastaveny tímto způsobem, pak by měly být umístěny na začátku regulárního výrazu. Neshodují se s ničím, takže jejich vliv na regulární výraz spočívá jen v nastavení příslušných příznaků.

Příznaky regulárních výrazů ➤ 480

Modul pro regulární výrazy

Modul `re` nabízí dva způsoby práce s regulárními výrazy. Můžeme použít funkce uvedené v tabulce 13.4 a každé z nich předáme regulární výraz jako první argument. Každá funkce převede regulární výraz do interního formátu (proces označovaný jako *kompilování*) a poté provede svoji činnost. To je velice pohodlné pro jednorázové použití, pokud ale potřebujeme použít tentýž regulární výraz opakovaně, pak se můžeme vyhnout opětovné kompilaci (která něco stojí) a zkompilovat jej jen jednou pomocí funkce `re.compile()`. Potom můžeme volat metody na zkompilovaném objektu představujícím regulární výraz tolikrát, kolikrát chceme. Atributy a metody takového objektu najdete v tabulce 13.6 (strana 480).

Tabulka 13.4: Funkce modulu pro práci s regulárními výrazy

Syntaxe	Popis
re.compile(r, *f*)	Vrátí zkompilovaný regulární výraz `r` s příznaky nastavený na *f*, jsou-li uvedeny. (Příznaky popisuje tabulka 13.5.)
re.escape(s)	Vrátí řetězec s alfanumerickými znaky zakódovanými pomocí zpětného lomítka. Vrácený řetězec tudíž neobsahuje žádný speciální znak regulárního výrazu.
re.findall(r, s, *f*)	Vrátí všechny nepřekrývající se shody regulárního výrazu `r` v řetězci `s` (ovlivněné příznaky *f*, jsou-li zadány). Má-li regulární výraz zachycení, pak se každá shoda vrátí jako n-tice se zachyceními.
re.finditer(r, s, *f*)	Vrátí objekt představující nalezenou shodu pro každou nepřekrývající se shodu regulárního výrazu `r` v řetězci `s` (ovlivněnou příznaky *f*, jsou-li zadány).
re.match(r, s, *f*)	Vrátí objekt představující nalezenou shodu (ovlivněnou příznaky *f*, jsou-li zadány), pokud se regulární výraz `r` shoduje se začátkem řetězce `s`. V opačném případě vrátí hodnotu `None`.
re.search(r, s, *f*)	Vrátí objekt představující nalezenou shodu (ovlivněnou příznaky *f*, jsou-li zadány), pokud se regulární výraz `r` shoduje s kteroukoli částí řetězce `s`. V opačném případě vrátí hodnotu `None`.

* Písmena používaná pro příznaky jsou stejná, jaká používá subsystém pro regulární výrazy v jazyku Perl, a proto se pro příznak `re.DOTALL` používá písmeno `s` a pro příznak `re.VERBOSE` písmeno `x`.

3.x

Syntaxe	Popis
re.split(r, s, m, *f*)	Vrátí seznam řetězců, které jsou výsledkem rozdělení řetězce `s` podle každého výskytu regulárního výrazu `r` (hledání shod je v Pythonu 3.1 ovlivněno příznaky *f*, jsou-li zadány), přičemž se provede nejvýše `m` rozdělení (nebo tolik, kolik jich lze provést, není-li *m* zadáno). Má-li regulární výraz zachycení, pak se tyto začlení do seznamu mezi části, které rozdělují.
re.sub(r, x, s, *m*, *f*)	Vrátí kopii řetězce `s`, v němž je každá shoda (nebo nejvýše *m* shod) regulárního výrazu `r` (v Pythonu 3.1 ovlivněna příznaky `f`, jsou-li zadány) nahrazená `x`, což může být řetězec nebo funkce (viz níže uvedený text).
re.subn(r, x, s *m*, *f*)	Tato funkce pracuje stejně jako funkce `re.sub()`, vrací však n-tici se dvěma prvky, které představují výsledný řetězec a počet provedených substitucí.

Tabulka 13.5: Příznaky modulu pro práci s regulárními výrazy

Příznak	Význam
`re.A` nebo `re.ASCII`	Aserce a třídy znaků `\b`, `\B`, `\s`, `\S`, `\w` a `\W` předpokládají, že řetězec je v kódování ASCII. Ve výchozím stavu pracují podle specifikace znakové sady Unicode.
`re.I` nebo `re.IGNORECASE`	Hledání shod s regulárním výrazem bude probíhat nezávisle na velikosti písmen.
`re.M` nebo `re.MULTILINE`	Aserce `^` se shoduje se začátkem a po každém znaku nového řádku a aserce `$` se shoduje před každým znakem nového řádku a s koncem.
`re.S` nebo `re.DOTALL`	Třída znaků `.` se shoduje s každým znakem včetně znaků nového řádku.
`re.X` nebo `re.VERBOSE`	Povoluje začlenění bílého místa a komentářů.

Tabulka 13.6: Atributy a metody objektu představujícího regulární výraz

Syntaxe	Popis
rx.findall(s, *začátek*, *konec*)	Vrátí všechny nepřekrývající se shody regulárního výrazu v řetězci `s` (nebo v řezu *začátek*`:`*konec* řetězce `s`). Má-li regulární výraz zachycení, pak se každá shoda vrátí jako n-tice se zachyceními.
rx.finditer(s, *začátek*, *konec*)	Vrátí objekt představující nalezenou shodu pro každou nepřekrývající se shodu regulárního výrazu v řetězci `s` (nebo v řezu *začátek*`:`*konec* řetězce `s`).
rx.flags	Příznaky, které byly nastaveny při kompilování regulárního výrazu.
rx.groupindex	Slovník, jehož klíči jsou názvy zachycovaných skupin a hodnotami čísla skupin. Pokud se názvy nepoužívají, je prázdný.
rx.match(s, *začátek*, *konec*)	Vrátí objekt představující nalezenou shodu, pokud se regulární výraz shoduje se začátkem řetězce `s` (nebo se začátkem řezu *začátek*`:`*konec* řetězce `s`). V opačném případě vrátí hodnotu `None`.

Syntaxe	Popis
rx.pattern	Řetězec, z něhož byl regulární výraz zkompilován.
rx.search(s, *začátek*, *konec*)	Vrátí objekt představující nalezenou shodu, pokud se regulární výraz shoduje s kteroukoli částí řetězce `s` (nebo řezu *začátek*:*konec* řetězce `s`). V opačném případě vrátí hodnotu `None`.
rx.split(s, m)	Vrátí seznam řetězců, které jsou výsledkem rozdělení řetězce `s` podle každého výskytu regulárního výrazu, přičemž se provede nejvýše `m` rozdělení (nebo tolik, kolik jich lze provést, není-li `m` zadáno). Má-li regulární výraz zachycení, pak se tato začlení do seznamu mezi části, které rozdělují.
rx.sub(x, s, m)	Vrátí kopii řetězce `s`, v němž je každá shoda (nebo nejvýše `m` shod) nahrazená `x`, což může být řetězec nebo funkce (viz níže uvedený text).
rx.subn(x, s m)	Tato metoda pracuje stejně jako metoda `rx.sub()`, vrací však n-tici se dvěma prvky, které představují výsledný řetězec a počet provedených substitucí.

```
match = re.search(r"#[\dA-Fa-f]{6}\b", text)
```

Tento úryvek kódu ukazuje použití funkce z modulu `re`. Regulární výraz se shoduje s barvami ve stylu HTML (např. `#C0C0AB`). Je-li nalezena shoda, pak funkce `re.search()` vrátí objekt představující tuto shodu. V opačném případě vrátí hodnotu `None`. Metody poskytované objekty, jež představují shodu, uvádí tabulka 13.7.

Tabulka 13.7: Atributy a metody objektu představujícího nalezenou shodu

Syntaxe	Popis
m.end(g)	Vrátí koncovou pozici shody v textu pro skupinu `g`, je-li zadána (nebo pro skupinu 0, tedy pro celou shodu). Vrátí hodnotu -1, pokud se této shody skupina neúčastní.
m.endpos	Koncová pozice hledání (konec textu nebo konec zadaný metodě `match()` nebo `search()`).
m.expand(s)	Vrátí řetězec `s`, který má značky zachycení (`\1`, `\2`, `\g<`*název*`>` a podobně) nahrazené odpovídajícími zachyceními.
m.group(g, ...)	Vrátí očíslovanou nebo pojmenovanou zachycovanou skupinu `g`. Je-li zadána více než jedna, vrátí n-tici s odpovídajícími zachycovanými skupinami (celá shoda je skupina 0).
m.groupdict(*výchozí*)	Vrátí slovník se všemi pojmenovanými zachycovanými skupinami s názvy jako klíči a zachyceními jako hodnotami. Je-li zadán parametr *výchozí*, pak se použije jako hodnota pro zachycované skupiny, které se na této shodě neúčastní.
m.groups(*výchozí*)	Vrátí n-tici se všemi zachycovanými skupinami počínaje číslem 1. Je-li zadán parametr *výchozí*, pak se použije jako hodnota pro zachycované skupiny, které se na této shodě neúčastní.

Syntaxe	Popis
m.lastgroup	Název zachycované skupiny s nejvyšším číslem, která se podílí na této shodě, nebo `None`, pokud taková neexistuje nebo nepoužívá názvy.
m.lastindex	Číslo nejvyšší zachycované skupiny, která se podílí na této shodě, nebo `None`, pokud taková neexistuje.
m.pos	Počáteční pozice pro hledání (začátek textu nebo začátek zadaný metodě `match()` nebo `search()`).
m.re	Objekt představující regulární výraz, který vytvořil tento objekt představující shodu.
m.span(g)	Vrátí počáteční a koncovou pozici této shody v textu pro skupinu `g`, je-li zadána (nebo pro skupinu 0, což je celá shoda). Vrátí n-tici (-1, -1), pokud se této shody skupina neúčastní.
m.start(g)	Vrátí počáteční pozici této shody v textu pro skupinu `g`, je-li zadána (nebo pro skupinu 0, což je celá shoda). Vrátí n-tici (-1, -1), pokud se této shody skupina neúčastní.
m.string	Řetězec, který byl předán metodě `match()` nebo `search()`.

Pokud bychom chtěli použít tento regulární výraz opakovaně, pak bychom jej mohli zkompilovat jednou, a poté použít zkompilovaný regulární výraz, kdykoli bychom jej potřebovali:

```
color_re = re.compile(r"#[\dA-Fa-f]{6}\b")
match = color_re.search(text)
```

Jak jsme si řekli již dříve, používáme holé řetězce, abychom nemuseli zdvojovat zpětná lomítka. Další možnost, jak zapsat tento regulární výraz, spočívá v použití třídy znaků `[\dA-F]` a předání příznaku `re.IGNORECASE` jako posledního argumentu funkci `re.compile()` nebo v použití regulárního výrazu `(?i)#[\dA-F]{6}\b`, který začíná příznakem pro ignorování velikosti písmen.

Pokud potřebujeme více než jeden příznak, můžeme je zkombinovat pomocí operátoru OR (`|`), například `re.MULTILINE|re.DOTALL`, nebo je přímo v regulárním výrazu umístit za sebe (např. `(?ms)`).

Tuto části uzavřeme prostudováním několika příkladů. Začneme regulárními výrazy, které jsme si ukázali již dříve, abychom si demonstrovali nejčastěji používanou funkčnost, nabízenou modulem `re`. Podívejme se tedy na regulární výraz pro hledání duplicitních slov:

```
double_word_re = re.compile(r"\b(?P<word>\w+)\s+(?P=word)(?!\w)",
                            re.IGNORECASE)
for match in double_word_re.finditer(text):
    print("{0} je duplikováno".format(match.group("word")))
```

Tento regulární výraz je malinko sofistikovanější než naše předchozí verze. Začíná hranicí slova (pro zajištění, aby každá shoda začínala na začátku slova), poté se hladově shoduje s jedním či více znaky představujícími „slovo“, poté s jedním či více bílými znaky a potom opět se stejným slovem – ovšem jen tehdy, není-li druhý výskyt tohoto slova následovaný znakem představujícím slovo.

Je-li vstupním textem „bota ta nota“, pak bychom *bez* první aserce obdrželi jednu shodu a jedno zachycení: `bota ta nota`. Dvě shody neobdržíme, protože zatímco výraz `(?P<word>)` provede shodu i zachycení, část `\s+` a `(?P=word)` provede pouze shodu. Použitím aserce představující hranici slova zajistíme, že prvním nalezeným slovem bude celé slovo, takže nakonec nebudeme mít žádnou shodu ani zachycení, protože v textu není žádné celé slovo duplikované. Podobně pokud by vstupním textem bylo „jedna to to dvě, pes sní cop“, pak bychom bez poslední aserce obdrželi dvě shody a dvě zachycení: `jedna to to dvě, pes sní cop`. Použití aserce s pohledem dopředu znamená, že druhé nalezené slovo je celým slovem, takže nakonec budeme mít jednu shodu a jedno zachycení: `jedna to to dvě, pes sní cop`.

Cyklus `for` prochází objekt představující shodu vrácený metodou `finditer()`, přičemž text zachycené skupiny získáme pomocí metody `group()` tohoto objektu. Stejně snadno (ovšem ne tak dobře z hlediska udržovatelnosti) bychom mohli použít volání `group(1)` – nemuseli bychom totiž zachycovanou skupinu pojmenovávat, takže by nám stačil regulární výraz `\b(\w+)\s+\1(?!\w)`. Dále je třeba poznamenat, že na konci bychom místo výrazu `(?!\w)` mohli použít hranici slova `\b`.

Dalším příkladem, který jsme si ukázali již dříve, byl regulární výraz pro hledání názvů souborů ve značkách `<img>` souborů HTML. Níže je uveden postup, jak tento regulární výraz zkompilovat. Přidáváme příznaky, aby se nerozlišovala velikost písmen a abychom mohli přidávat komentáře:

```
image_re = re.compile(r"""
    <img\s+                   # začátek značky
    [^>]*?                    # atributy jiné než src
    src=                      # start of src attribute
    (?:
        (?P<quote>["'])       # otevírací uvozovky
        (?P<qimage>[^\1>]+?)  # název souboru s obrázkem
        (?P=quote)            # uzavírací uvozovky
    |                         # ---nebo alternativně---
        (?P<uimage>[^"' >]+)  # název souboru s obrázkem bez uvozovek
    )
    [^>]*?                    # atributy jiné než src
    >                         # konec značky
    """, re.IGNORECASE|re.VERBOSE)
image_files = []
for match in image_re.finditer(text):
    image_files.append(match.group("qimage") or match.group("uimage"))
```

Opět používáme metodu `finditer()` pro získání každé shody a funkci `group()` objektu představující tuto shodu pro získání zachyceného textu. Při každé shodě nevíme, u které ze skupin (`"qimage"` nebo `"uimage"`) k této shodě došlo, což ale krásně vyřešíme operátorem `or`. Nerozlišování velikosti písmen se vztahuje jen na výrazy `img` a `src`, a proto bychom mohli zahodit příznak `re.IGNORECASE` a použít místo něj `[Ii][Mm][Gg]` a `[Ss][Rr][Cc]`. Regulární výraz by byl sice hůře čitelný, ale mohl by pracovat rychleji, protože by nevyžadoval, aby se zpracovávaný text převedl na velká (či malá) písmena – ovšem rozdíl v rychlosti by byl znát jen při použití regulárního výrazu na skutečně rozsáhlý text.

Běžnou činností je vzít text HTML a vypsat jen holý text, který obsahuje. To bychom mohli přirozeně provést prostřednictvím některého z analyzátorů Pythonu, avšak pomocí regulárních výrazů si můžeme vytvořit jednoduchý nástroj. Ve skutečnosti musíme provést tři kroky: vymazat jakékoli značky, nahradit entity znaky, které reprezentují, a vložit prázdné řádky pro oddělení odstavců. Zde je funkce (z programu `html2text.py`), která tyto kroky provádí:

```
def html2text(html_text):
    def char_from_entity(match):
        code = html.entities.name2codepoint.get(match.group(1), 0xFFFD)
        return chr(code)

    text = re.sub(r"<!--(?:.|\n)*?-->", "", html_text)          #1
    text = re.sub(r"<[Pp][^>]*?>", "\n\n", text)                #2
    text = re.sub(r"<[^>]*?>", "", text)                        #3
    text = re.sub(r"&#(\d+);", lambda m: chr(int(m.group(1))), text)
    text = re.sub(r"&([A-Za-z]+);", char_from_entity, text)     #5
    text = re.sub(r"\n(?:[ \xA0\t]+\n)+", "\n", text)           #6
    return re.sub(r"\n\n+", "\n\n", text.strip())               #7
```

První regulární výraz, `<!--(?:.|\n)*?-->`, se shoduje s komentáři jazyka HTML, a to včetně těch, které v sobě mají vnořené další značky HTML. Funkce `re.sub()` nahradí tolik shod, kolik nalezne. Je-li uvedenou náhradou prázdný řetězec (jako v našem příkladu), pak tyto shody vymaže. (Zadáním dalšího celého čísla můžeme stanovit maximální počet nahrazovaných shod.)

Dbáme na to, abychom používali nehladové (minimální) hledání shody, které zajistí, že při každé shodě vymažeme jen jeden komentář. Pokud bychom takto nepostupovali, vymazali bychom vše od začátku prvního komentáře po konec posledního komentáře.

3.0 V Pythonu 3.0 nepřijímá funkce `re.sub()` příznaky jako argumenty, a protože `.` znamená „jakýkoliv znak kromě znaku nového řádku“, tak musíme hledat `.` nebo `\n`. Tyto shody navíc nesmíme hledat pomocí třídy znaků, ale pomocí alternace, poněvadž uvnitř třídy znaků má `.` (tj. znak „tečka“) svůj literálový význam. Další možností by bylo začít regulární výraz s vloženým příznakem, například `(?s)<!--.*?-->`, nebo bychom objekt představující regulární výraz mohli zkompilovat s příznakem `re.DOTALL`, takže bychom mohli použít jen `<!--.*?-->`.

3.1 Počínaje Pythonem 3.1 mohou funkce `re.split()`, `re.sub()` a `re.subn()` přijímat příznaky jako argument, takže bychom mohli použít `<!--.*?-->` a předat příznak `re.DOTALL`.

Druhý regulární výraz, `<[Pp][^>]*?>`, se shoduje s otevíracími značkami pro odstavec (jako je např. `<P>` nebo `<p align="center">`). Shoduje se s otevřením `<p` (nebo `<P`), poté s libovolnými atributy (s použitím nehladového hledání) a nakonec s uzavíracím znakem `>`. Druhé volání funkce `re.sub()` používá tento regulární výraz pro nahrazení otevíracích značek pro odstavec znaky nového řádku (což je standardní způsob oddělování odstavců v holém textovém souboru).

Třetí regulární výraz, `<[^>]*?>`, se shoduje s libovolnou značkou a používáme jej ve třetím volání funkce re.sub() pro vymazání všech zbývajících značek.

Entity jazyka HTML představují jistý způsob specifikování znaků nespadajících do kódování ASCII pomocí znaků z kódování ASCII. Mohou mít dvě formy: `&`*`název`*`;`, kde *`název`* je název znaku (např. `&copy;` pro ©) a `&#`*`číslice`*`;`, kde *`číslice`* jsou desítkové číslice identifikující kódový bod znakové sady Unicode (např. `&#165;` pro ¥). Čtvrté volání funkce `re.sub()` používá regulární výraz `&#(\d+);`, který se shoduje s číslicovou formou a zachycuje číslice do zachycované skupiny 1. Pro nahrazení nepoužíváme textový literál, ale lambda funkci. Když funkci `re.sub()` předáme funkci, pak se tato funkce zavolá pro každou nalezenou shodu a obdrží jako jediný argument objekt představující tuto shodu. Uvnitř lambda funkce získáme číslice (jako řetězec), které vestavěnou funkcí `int()` převedeme na celé číslo, a poté pomocí vestavěné funkce `chr()` získáme pro zadaný kódový bod znak ze znakové sady Unicode. Návratová hodnota funkce (nebo v tomto případě lambda výrazu, takže se jedná o výsledek výrazu) se použije jako nahrazovací text.

Páté volání funkce `re.sub()` používá regulární výraz `&([A-Za-z]+);` pro zachycení pojmenovaných entit. Modul `html.entities` ze standardní knihovny obsahuje slovníky entit včetně slovníku name2codepoiint, jehož klíč jsou názvy entit a hodnotami celočíselné kódové body. Funkce `re.sub()` zavolá při každé nalezené shodě místní funkci `char_from_entity()`. Ta používá slovník `dict.get()` s výchozím argumentem `0xFFFD` (kódový bod pro standardní nahrazovací znak znakové sady Unicode – často vyobrazovaný jako ❓). Díky tomu bude kódový bod vždy k dispozici a ve spojení s funkcí `chr()` tak získáme vhodný znak pro nahrazení pojmenované entity; v případě neplatného názvu entity se použije nahrazovací znak znakové sady Unicode.

Regulární výraz `\n(?:[ \xA0\t]+\n)+` šestého volání funkce `re.sub()` se používá pro vymazání řádků, které obsahují pouze bílé místo. Námi použitá třída znaků obsahuje mezeru, nezrušitelnou mezeru (která v předchozím regulárním výrazu nahradila entity ` `) a tabulátor. Tento regulární výraz se shoduje se znakem nového řádku (jedná se o znak na konci řádku, který je umístěn před jedním či více řádky obsahujícími pouze bílé místo) a poté s nejméně jedním řádkem (a s nejvíce tolika, kolik jich je), který obsahuje pouze bílé místo. Tato shoda obsahuje znak nového řádku, a proto ji musíme na řádku, který je umístěn před řádky obsahující pouze bílé místo, nahradit jediným znakem nového řádku. Jinak bychom totiž nevymazali jen řádky obsahující bílé místo, ale také znak nového řádku na řádku, který je umístěn před nimi.

Výsledek sedmého a posledního volání funkce `re.sub()` vracíme volajícímu. Regulární výraz \n\n+ používáme pro nahrazení posloupností dvou a více znaků nového řádku přesně dvěma znaky nového řádku, abychom tak zajistili, že každý odstavec bude oddělen jediným prázdným řádkem.

V příkladu s HTML jsme žádný z nahrazujících řetězců nebrali přímo z nalezené shody (použili jsme názvy entit jazyka HTML a čísla), avšak v některých situacích může být nezbytné, aby náhrada obsahovala celý text nalezené shody nebo některou jeho část. Máme-li kupříkladu seznam názvů, každý ve formě *`Křestní_jméno Prostřední_jméno1 ... Prostřední_jménoN Příjmení`*, kde může být libovolný počet prostředních jmen (nebo žádné), a my chceme vytvořit novou verzi seznamu s každým prvkem ve formě *`Příjmení,Křestní_jménoProstřední_jméno1 ... Prostřední_jménoN`*, pak můžeme použít následující regulární výraz:

```
new_names = []
for name in names:
    name = re.sub(r"(\w+(?:\s+\w+)*)\s+(\w+)", r"\2, \1", name)
    new_names.append(name)
```

V první části `(\w+(?:\s+\w+)*)` regulárního výrazu se s křestním jménem shoduje první výraz `\w+` a s nula či více prostředními jmény výraz `(?:\s+\w+)*`. Výraz pro prostřední jména se shoduje s nula či více výskyty bílého místa následovaného slovem. Druhá část `\s+(\w+)` regulárního výrazu se shoduje s bílým místem, které následuje za křestním jménem (a prostředními jmény), a s příjmením.

Pokud regulární výraz vypadá tak trochu jako šum na lince, pak můžeme pro zlepšení čitelnosti a udržovatelnosti použít pojmenované zachycované skupiny:

```
name = re.sub(r"(?P<forenames>\w+(?:\s+\w+)*)"
              r"\s+(?P<surname>\w+)",
              r"\g<surname>, \g<forenames>", name)
```

Na zachycený text se lze odkazovat ve funkci či metodě `sub()` nebo `subn()` pomocí syntaxe `\i` nebo `\g<`*`id`*`>`, kde *`i`* je číslo zachycované skupiny a *`id`* je název nebo číslo zachycované skupiny. Takže `\1` je totéž jako `\g<1>`, což je v tomto příkladu totéž jako `\g<forenames>`. Tuto syntaxi lze použít také v řetězci, který předáváme metodě `expand()` objektu, jenž představuje nalezenou shodu.

Proč nezabere první část regulárního výrazu celé jméno? Koneckonců používá přece hladové hledání shody! Ve skutečnosti zabere, ale poté hledání shody selže, protože i když se část pro prostřední jména může shodovat nulakrát či vícekrát, část pro příjmení se musí shodovat právě jednou, avšak hladová část pro prostřední jména zabrala vše. Po neúspěchu se subsystém pro regulární výrazy vydá zpět, vzdá se posledního „prostředního jména“, čímž umožní provedení shody s příjmením. Přestože se hladové hledání shody shoduje s maximálním možným počtem nálezů, zastaví se, pokud by další shody znamenaly neúspěch.

Máme-li například jméno „John le Carré“, najde náš regulární výraz nejdříve shodu s celým názvem, to znamená `John le Carré`. Tím je splněna první část regulárního výrazu, pro část s příjmením ale nezůstalo nic, a protože je příjmení povinné (má implicitní kvantifikátor s hodnotou 1), regulární výraz selhal. Část s prostředními jmény je kvantifikována `*`, může se tedy shodovat nulakrát či vícekrát (v tomto případě dvakrát: „le“ a „Carré“), a proto ji subsystém pro regulární výrazy může nechat, aby se vzdala některých nalezených shod, aniž by to vedlo k selhání. Z tohoto důvodu se regulární výraz vydá zpět, vzdá se poslední shody `\s+\w+` (tj. „Carré“), takže shoda bude mít podobu `John le Carré`, bude splňovat celý regulární výraz a bude obsahovat dvě nalezené skupiny se správnými texty.

Tento regulárním výraz má ve své nynější podobě jednu slabinu: nezvládá totiž správně křestní jména, která jsou zapsána jako iniciály (např. „James W. Loewen“ nebo „J. R. R. Tolkien“). To je dáno tím, že část `\w` se shoduje se znaky slov a ty nezahrnují tečku. Očividným, i když nesprávným řešením je změnit na obou místech část regulárního výrazu pro křesní jména `\w+` na výraz `[\w.]+`. Tečka ve třídě znaků znamená literál tečky a zkratky pro třídy znaků si ve třídách znaků zachovávají svůj význam, takže nový výraz se shoduje se znaky slov nebo s tečkami. Toto řešení by ale povolovalo jména, jako jsou „.“, „..“, „.A“, „.A.“ a tak dále. Je tedy zapotřebí sáhnout po důvtipnějším řešení.

```
name = re.sub(r"(?P<forenames>\w+\.?(?:\s+\w+\.?)*)"
              r"\s+(?P<surname>\w+)",
              r"\g<surname>, \g<forenames>", name)
```

Zde jsme změnili část regulárního výrazu pro křestní jména (první řádek). První část regulárního výrazu pro křestní jména se shoduje s jedním nebo více znaky slov, za nimiž může volitelně následovat tečka. Druhá část se shoduje s nejméně jedním znakem bílého místa, poté s jedním či více znaky slov volitelně následovanými tečkou, přičemž celá tato druhá část se může shodovat nulakrát či vícekrát.

Když použijeme alternaci (|) se dvěma či více alternativami zachycení, pak nevíme, ve které z nich došlo ke shodě, a proto ani nevíme, ze které zachycené skupiny máme načíst zachycený text. Můžeme samozřejmě projít všechny skupiny a najít tu, která je neprázdná, ovšem častěji můžeme v takovéto situaci využít atribut `lastindex` objektu nalezené shody, který nám poskytne požadované číslo skupiny. Podíváme se na poslední příklad, na němž si ukážeme tuto techniku a získáme další zkušenosti s prací s regulárními výrazy.

Představte si, že chceme zjistit, jaké kódování používá soubor HTML, XML nebo Pythonu. Mohli bychom jej otevřít v binárním režimu a načíst řekněme prvních 1000 bajtů do objektu typu `bytes`. Potom bychom soubor mohli zavřít, najít v objektu typu `bytes` kódování a soubor znovu otevřít v textovém režimu s použitím nalezeného kódování nebo s použitím záložního kódování (např. UTF-8). Subsystém pro regulární výrazy očekává, že regulární výrazy budou zadávány jako řetězce, ale text, na který regulární výraz aplikujeme, může být objektem typu `str`, `bytes` nebo `bytearray`. Při použití objektu typu `bytes` nebo `bytearray` vracejí všechny funkce a metody místo řetězců bajty a příznak `re.ASCII` je implicitně zapnutý.

Pro soubory HTML je kódování obvykle uvedeno ve značce `<meta>` (je-li vůbec uvedeno), například `<meta http-equiv='Content-Type' content='text/html; charset=ISO-8859-1'/>`. Soubory XML mají výchozí kódování UTF-8, což lze ale změnit, například `<?xml version="1.0" encoding="Shift_JIS"?>`. Soubory Pythonu 3 mají taktéž výchozí kódování UTF-8, což lze ale též změnit doplněním řádku jako je například `# encoding: latin1` nebo `# -*- coding: latin1 -*`, bezprostředně za řádek shebang.

Níže je uveden způsob, jakým bychom mohli najít použité kódování. Předpokládáme, že proměnná `binary` je objekt typu `bytes` obsahující prvních 1000 bajtů souboru s kódem jazyka HTML, XML nebo Python:

```
match = re.search(r"""(?<![-\w])                    #1
                      (?:(?:en)?coding|charset)     #2
                      (?:=(["']?)([-\w]+)(?(1)\1)   #3
                      |:\s*([-\w]+))""".encode("utf8"),
                    binary, re.IGNORECASE|re.VERBOSE)
encoding = match.group(match.lastindex) if match else b"utf8"
```

K prohledání objektu typu `bytes` musíme uvést vzor, který je též objektem typu `bytes`. V tomto případě chceme použít holý řetězec, který pak musíme převést na objekt typu `bytes` a předat jako první argument funkci `re.search()`.

První částí samotného regulárního výrazu je aserce s pohledem dozadu, která říká, že shodě nemůže předcházet pomlčka nebo znak slova. Druhá část se shoduje s „encoding“, „coding“ nebo „charset“ a mohli bychom ji zapsat také jako `(?:encoding|coding|charset)`. Třetí část jsme rozprostřeli na dva řádky, abychom zdůraznili skutečnost, že obsahuje dvě alternativní části, `=(["']?)([-\w]+)(?(1)\1)` a `:\s*([-\w]+)`, přičemž shoda může nastat pouze u jedné z nich. První se shoduje s rovnítkem následovaným jedním či dvěma znaky slova nebo pomlček (volitelně uzavřenými do odpovídajících uvozovek pomocí podmíněné shody), přičemž druhý se shoduje s dvojtečkou, a potom s volitelným bílým místem následovaným jedním či více znaky slova nebo pomlček. (Vzpomeňte si, že pomlčka uvnitř třídy znaků se považuje za literál, je-li prvním znakem. V opačném případě představuje rozsah znaků, například `[0-9]`.)

Podmíněné hledání shody ➢ 477

Místo nutnosti psát `(?:(?:[Ee][Nn])? [Cc][Oo][Dd][Ii][Nn][Gg]|[Cc][Hh][Aa][Rr][Ss][Ee][Tt])` jsme použili příznak `re.IGNORECASE` a kvůli možnosti pěkně rozložit regulární výraz a doplnit komentáře (v tomto případě jde jen o čísla, abychom se mohli v textu snadno odkazovat na jednotlivé části) jsme použili příznak `re.VERBOSE`.

Máme zde tři zachycované skupiny, všechny ve třetí části: `(["'])?` zachycuje volitelnou otevírací uvozovku, `([-\w]+)` zachycuje kódování za rovnítkem a druhý výraz `([-\w]+)` (na následujícím řádku) zachycuje kódování za dvojtečkou. Nás zajímá pouze kódování, a proto chceme získat buď druhou, nebo třetí zachycovanou skupinu, z nichž se může shodovat pouze jediná, poněvadž se jedná o alternativy. Atribut `lastindex` uchovává index poslední zachycované skupiny (v tomto příkladu 2 nebo 3, dojde-li ke shodě), takže získáme kteroukoli shodu nebo použijeme výchozí kódování, pokud k žádné shodě nedošlo.

Ukázali jsme si veškerou nejčastěji používanou funkčnost modulu `re` v akci, takže tuto část uzavřeme zmínkou o jedné poslední funkci. Funkce `re.split()` (nebo metoda `split` objektu s nalezenou shodou) dokáže rozdělit řetězce podle regulárního výrazu. Běžným požadavkem je získat seznam slov rozdělením textu podle bílého místa. Toho lze docílit voláním `re.split(r"\s+", text)`, které vrátí seznam slov (nebo přesněji seznam řetězců, z nichž každý se shoduje s výrazem `\S+`). Regulární výrazy jsou velice mocné a užitečné, a jakmile si je osvojíte, snadno uvidíte všechny problémy související s textem jako požadavky na vytvoření určitého regulárního výrazu. Někdy jsou ale metody řetězce dostatečné a také vhodnější. Text můžeme například stejně snadno rozdělit podle bílého místa pomocí volání `text.split()`, protože výchozí chování metody `str.split()` je rozdělení podle výrazu `\s+`.

Shrnutí

Regulární výrazy nabízejí mocný způsob pro hledání řetězců v textu na základě určitého vzoru a pro nahrazení takovýchto řetězců jinými řetězci, které samy závisejí na tom, co bylo nalezeno.

V této lekci jsme viděli, že shoda s většinou znaků probíhá na základě jejich literálů, které jsou implicitně kvantifikovány výrazem `{1}`. Dozvěděli jsme se, jak specifikovat třídy znaků (množiny znaků, které se mají hledat) a jak takovéto množiny negovat a začleňovat do nich rozsahy znaků bez nutnosti vypisovat jednotlivé znaky.

Naučili jsme se, jak kvantifikovat výrazy, aby odpovídaly určitému počtu nalezených shod nebo počtu, který spadá do zadaného minima a maxima, a jak používat hladové a nehladové hledání shody. Dále jsme se dozvěděli, jak seskupovat jeden nebo více výrazů dohromady, aby je bylo možné kvantifikovat (a volitelně zachycovat) jako jeden celek.

V této lekci jsme si také ukázali, jak lze to, co se nalezne jako shoda, ovlivnit pomocí nejrůznějších asercí, jako je pozitivní a negativní pohled dopředu a dozadu, a pomocí nejrůznějších příznaků, které slouží například pro řízení interpretace tečky nebo pro stanovení, zda se má rozlišovat velikost písmen.

V poslední části jsme si ukázali, jak používat regulární výrazy v programech napsaných v jazyku Python. V této části jsme se naučili, jak používat funkce poskytované modulem `re` a metody dostupné ze zkompilovaných regulárních výrazů a z objektů představujících nalezenou shodu. Dále jsme se dozvěděli, jak nahrazovat nalezené shody řetězcovými literály, jež obsahují zpětné odkazy, a výsledky

volání funkcí nebo lambda výrazů a jak pomocí pojmenovaných zachycení a komentářů psát lépe udržovatelné regulární výrazy.

Cvičení

1. V řadě případů (např. v některých webových formulářích) musejí uživatelé zadávat telefonní číslo, přičemž některé dráždí uživatele tím, že přijímají pouze určitý formát. Napište program, který čte americká telefonní čísla s třícifernou volbou oblasti a sedmiciferným místním kódem. Čísla mohou mít tvar deseti číslic, mohou být rozdělená do bloků pomocí pomlčky nebo mezer a kód oblasti může být volitelně uzavřen do závorek. Kupříkladu všechna tato čísla jsou platná: 555-123-1234, (555) 1234567, (555) 123 1234 a 5551234567. Čtěte telefonní čísla ze vstupu `sys.stdin` a pro každé proveďte výpis čísla ve formě „(999) 999 9999“ nebo nahlaste chybu, je-li číslo neplatné nebo nemá-li přesně deset číslic.

   Regulární výraz shodující se s takovýmito telefonními čísly má kolem deseti řádků (při použití příznaku `re.VERBOSE`) a je docela přímočarý. Řešení je k dispozici v souboru `phone.py` a má kolem dvaceti pěti řádků.

2. Napište malý program, který čte soubor s kódem jazyka XML nebo HTML zadaný na příkazovém řádku a pro každou značku, která obsahuje atributy, vypíše název značky a pod něj její atributy. Zde je výňatek z výstupu programu při zadání souboru `index.html` náležícího k dokumentaci jazyka Python:

```
html
    xmlns = http://www.w3.org/1999/xhtml
meta
    http-equiv = Content-Type
    content = text/html; charset=utf-8
li
    class = right
    style = margin-right: 10px
```

   První možností je použít dva regulární výrazy, jeden pro zachycení značek s jejich atributy a druhý pro extrahování názvu a hodnoty každého atributu. Hodnoty atributů mohou být uzavřeny do jednoduchých či dvojitých uvozovek (v takovém případě mohou obsahovat bílé místo a uvozovky, v nichž nejsou uzavřeny) nebo mohou být bez uvozovek (v takovém případě nemohou obsahovat bílé místo ani uvozovky). Nejjednodušší je asi začít vytvořením samostatných regulárních výrazů pro obsluhu hodnot s uvozovkami a bez nich a poté tyto dva regulární výrazy sloužit do jediného regulárního výrazů řešícího oba případy. Nejlepší je použít pojmenované skupiny, aby byl regulární výraz lépe čitelný. Není to snadné, a to zejména proto, že uvnitř třídy znaků lze použít pouze číslované zpětné odkazy.

   Řešení najdete v souboru `extract_tags.py`, který má méně než 35 řádků. Regulární výraz pro značku a atributy je na jediném řádku. Regulární výraz pro názvy a hodnoty atributu má půl tuctu řádků a používá alternaci, podmíněné hledání shody (dvakrát, jedno vnořené ve druhém) a hladové i nehladové kvantifikátory.

LEKCE 14

Úvod do syntaktické analýzy

V této lekci:

- Terminologie formy BNF a syntaktické analýzy
- Ruční tvorba analyzátorů
- Syntaktická analýza ve stylu jazyka Python pomocí nástroje PyParsing
- Syntaktická analýza s nástrojem PLY podle nástrojů Lex a Yacc

Syntaktická analýza je v mnoha programech nezbytnou činností a kromě nejtriviálnějších případů se jedná o náročné téma. Syntaktická analýza se často provádí v situaci, kdy potřebujeme číst data, která jsou uložena v určitém formátu, abychom je mohli zpracovat nebo nad nimi provést nějaké dotazy. Někdy je též nezbytné analyzovat jazyk DSL (Domain-Specific Language – doménově specifický jazyk). Jedná se o minijazyky určené pro specifické oblasti, jejichž popularita je zdá se na vzestupu. Ať už potřebujeme číst data v určitém formátu nebo kód zapsaný pomocí jazyka DSL, budeme muset vytvořit vhodný analyzátor. Ten můžeme vytvořit buď ručně, nebo využít některý z modulů Pythonu pro generickou analýzu.

V Pythonu můžeme psát analyzátory s použitím kterékoli standardní techniky z oblasti informatiky, jako jsou regulární výrazy, konečné automaty, rekurzivně sestupné analyzátory a další. Všechny tyto techniky mohou fungovat docela dobře, ale pro složitá data či jazyky DSL (např. rekurzivně strukturovaný jazyk DSL s operátory s různou precedencí a asociativitou) může být obtížné provést vše správně. Pokud navíc potřebujeme analyzovat spoustu odlišných datových formátů nebo jazyků DSL, může ruční psaní všech analyzátorů spotřebovat množství času a bude obtížné všechny je udržovat.

Pro některé datové formáty však analyzátory naštěstí vůbec psát nemusíme. Když například chceme analyzovat kód jazyka XML, můžeme sáhnout po standardní knihovně Pythonu, která se dodává s analyzátory pro model DOM, rozhraní SAX a strom elementů, přičemž další analyzátory XML jsou k dispozici jako doplňky třetích stran.

Souborové formáty ➢ 215

Python má ve skutečnosti vestavěnou podporu pro čtení a zapisování širokého spektra datových formátů, mezi něž patří data s jednoduchými oddělovači (modul `csv`), soubory ve stylu konfiguračních souborů `.ini` ve Windows (modul `configparser`), data ve formátu JSON (modul `json`) a také několik dalších, o nichž jsme se zmínili v lekci 5. Python neposkytuje žádnou vestavěnou podporu pro analýzu jiných jazyků, nabízí však modul `shlex`, který lze použít pro vytvoření lexikálního analyzátoru pro jazyky DSL podobné unixovému shellu, a modul `tokenize`, který poskytuje lexikální analyzátor pro zdrojový kód napsaný v jazyku Python. A pak jsou tu samozřejmě vestavěné funkce `eval()` a `exec()`, které dokážou spouštět kód jazyka Python.

Dynamické provádění kódu ➢ 334

Obecně lze říci, že pokud Python již obsahuje vhodný analyzátor ve standardní knihovně nebo jako doplněk třetí strany, pak je obvykle nejlepší použít raději tento analyzátor a nepouštět se do tvorby vlastního.

Chceme-li se pustit do syntaktické analýzy datových formátů či jazyků DSL, pro něž není k dispozici žádný analyzátor, můžeme místo ruční tvorby analyzátoru použít některý z modulů Pythonu třetích stran pro univerzální syntaktickou analýzu. V této lekci se seznámíme se dvěma nejpopulárnějšími analyzátory třetích stran. Jedním z nich je nástroj PyParsing, jehož autorem je Paul McGuire. Tento nástroj nabízí jedinečný a pro Python typický přístup. Dalším je nástroj PLY (Python Lex Yacc), jehož autorem je David Beazley. Nástroj PLY je věrně vymodelován podle klasických unixových nástrojů lex a yacc a značným způsobem využívá regulární výrazy. K dispozici je řada dalších analyzátorů, z nichž mnoho je uvedeno na stránce *www.dabeaz.com/ply* (ve spodní části) a samozřejmě na stránce se seznamem balíčků pro jazyk Python (*pypi.python.org/pypi*).

První část této lekce nabízí stručné seznámení se syntaxí standardní formy BNF (Backus-Naur Form – Backusova-Naurova forma) používané pro popis gramatik datových formátů a jazyků DSL. V této části si též vysvětlíme základní terminologii. Ve všech zbývajících částech se budeme věnovat samotné syntaktické analýze, přičemž ve druhé části se podíváme na ruční tvorbu analyzátorů pomocí regu-

lárních výrazů a rekurzivního sestupu, čímž plynule navážeme na výklad regulárních výrazů z předchozí lekce. Ve třetí části si představíme nástroj PyParsing. První příklady budou stejné jako ty, pro něž jsme ve druhé části ručně vytvořili analyzátory, což nám pomůže při osvojení přístupu nástroje PyParsing k syntaktické analýze a dále tak získáme možnost tyto přístupy porovnat. Poslední příklad této části obsahuje ambicióznější gramatiku, s níž se zde setkáme poprvé. V poslední části se seznámíme s nástrojem PLY a ukážeme si stejné příklady, které jsme použili v části o nástroji PyParsing, díky čemuž si opět snadněji osvojíme práci s tímto modulem a budeme jej moci porovnat s předchozími přístupy.

Je třeba poznamenat, že až na jedinou výjimku si popis, gramatiku ve formě BNF a ukázku dat či jazyka DSL každého datového formátu a jazyka DSL uvedeme pouze v části o ručně psaných analyzátorech, přičemž v ostatních částech budou na příslušných místech k dispozici pouze odpovídající zpětné odkazy. Výjimkou je analyzátor logiky prvního řádu, jehož podrobnosti si uvedeme v části o nástroji PyParsing a příslušné zpětné odkazy v části o nástroji PLY.

Terminologie formy BNF a syntaktické analýzy

Syntaktická analýza představuje prostředek pro transformaci dat, která se nacházejí v nějakém strukturovaném formátu (bez ohledu na to, zda jde o skutečná data, o příkazy v nějakém programovacím jazyku nebo o směsici obou), do reprezentace, která odráží strukturu těchto dat a kterou lze použít k odvození významu, jejž tato data reprezentují. Proces analýzy obvykle tvoří dvě fáze: lexikální analýza (označovaná též jako tokenizace nebo skenování) a řádná analýza (označovaná též obecnějším pojmem jako syntaktická analýza).

Máme-li například českou větu „malý pes štěkal“, můžeme ji transformovat do posloupnosti n-tic (slovní druh, slovo): `((ADJECTIVE, "malý"), (NOUN, "pes"), (VERB, "štěkal"))`. Nyní můžeme provést syntaktickou analýzu, abychom zjistili, jedná-li se o platnou českou větu. V tomto případě tomu tak je, ale náš analyzátor by měl umět odmítnout třeba větu „malý štěkal pes“.*

Lexikální fáze se používá pro převod dat do proudu tokenů. V typickém případě uchovává každý token nejméně dva údaje: typ tokenu (reprezentovaný druh dat nebo jazykový konstrukt) a hodnotu tokenu (která může být prázdná, stačí-li samotný typ – například klíčové slovo v programovacím jazyku).

Ve fázi analýzy čte analyzátor každý token a provádí určité sémantické akce. Analyzátor pracuje podle předem definované sady pravidel gramatiky, jež definují syntaxi, kterou by měla data dodržovat. (Pokud data tato syntaktická pravidla nedodržují, pak analyzátor správně neuspěje.) Ve vícefázových analyzátorech je sémantická akce tvořena sestavováním interní reprezentace vstupní paměti (označované jako ATS, Abstract Syntax Tree – abstraktní syntaktický strom), která slouží jako vstup do následující fáze. Jakmile je abstraktní syntaktický strom zkonstruován, můžeme jej procházet a dotazovat se na data, zapsat je do jiného formátu nebo provádět výpočty, která odpovídají významu zakódovanému v těchto datech.

Datové formáty a jazyky DSL (a programovací jazyky obecně) lze popsat pomocí *gramatiky*, což je množina syntaktických pravidel, jež definují platnou syntaxi pro data či jazyk. Samozřejmě nelze říci, že syntakticky platná věta musí dávat nějaký smysl. Například „kočka snědla demokracii“ je platná česká

* V praxi je syntaktická analýza českého jazyka a dalších přirozených jazyků velice obtížný problém. Více informací najdete například na webových stránkách projektu Natural Language Toolkit (*www.nltk.org*).

věta, která ale nedává vůbec žádný smysl. Nicméně možnost definování gramatiky je velice užitečná, neboť pro gramatiky existuje obecně používaná syntaxe označovaná jako forma BNF (Backus-Naur Form – Backusova-Naurova forma). Vytvoření formy BNF je prvním krokem k vytvoření analyzátoru, a přestože není formálně nezbytná, je až na nejtriviálnější gramatiky naprosto zásadní.

Zde si popíšeme velmi jednoduchou podmnožinu syntaxe BNF, která je pro naše potřeby dostatečná.

Ve formě BNF jsou dva druhy prvků: terminály a neterminály. Terminál je prvek, který je již ve své konečné podobě (např. číselný nebo řetězcový literál). Neterminál je prvek, který je definován nulovým nebo větším počtem jiných prvků (které samy mohou být terminály či neterminály). Každý neterminál musí být vposledku definován žádným či několika terminály. Obrázek 14.1 znázorňuje příklad formy BNF, jež definuje syntaxi souboru s „atributy“.

```
ATTRIBUTE_FILE ::= (ATTRIBUTE '\n')+
ATTRIBUTE      ::= NAME '=' VALUE
NAME           ::= [a-zA-Z]\w*
VALUE          ::= 'true' | 'false' | \d+ | [a-zA-Z]\w*
```

Obrázek 14.1: Forma BNF pro soubor atributů

Symbol `::=` znamená *je definován jako*. Neterminály se píší velkými, zkosenými písmeny (např. `VALUE`). Terminály jsou buď řetězcové literály uzavřené do uvozovek (např. `'='` nebo `'true'`), nebo regulární výrazy (např. `\d+`). Definice (na pravé straně `::=`) jsou tvořeny jedním či více terminály nebo neterminály – ty se pak musejí pro splnění definice vyskytovat v zadané posloupnosti. Pro alternativy se používá svislá čára (`|`), takže pro splnění definice není nutná shoda s celou posloupností, ale s libovolnou alternativou. Terminály a neterminály lze kvantifikovat pomocí `?` (žádný nebo jeden výskyt, tj. volitelné), `+` (jeden nebo více výskytů) nebo `*` (žádný nebo více výskytů). Bez explicitně uvedeného kvantifikátoru jsou kvantifikovány na shodu s právě jedním výskytem. Pro seskupování dvou či více terminálů či neterminálů, s nimiž chceme pracovat jako s jedním celkem (např. pro seskupení alternativ nebo pro kvantifikaci), můžeme použít závorky.

Forma BNF má vždy nějaký „počáteční symbol“, což je neterminál, s nímž se musí shodovat celý vstup. Budeme se řídit konvencí, že první neterminál je vždy počátečním symbolem.

V tomto příkladu máme čtyři neterminály: `ATTRIBUTE_FILE` (počáteční symbol), `ATTRIBUTE`, `NAME` a `VALUE`. Neterminál `ATTRIBUTE_FILE` je definován jako jeden či více neterminálů `ATTRIBUTE` následovaných novým řádkem. Neterminál `ATTRIBUTE` je definován jako neterminál `NAME` následovaný literálem `=` (tj. terminálem), za nímž následuje neterminál `VALUE`. Části `NAME` a `VALUE` jsou neterminály, a proto musejí být také definovány. Neterminál `NAME` je definován regulárním výrazem (tj. terminálem). Neterminál `VALUE` je definován libovolnou ze čtyř alternativ tvořených dvěma literály a dvěma regulárními výrazy (vše jsou terminály). Všechny neterminály jsou již definované pomocí terminálů (nebo pomocí neterminálů, které jsou samy nakonec definovány pomocí terminálů), a tudíž je naše forma BNF kompletní.

Forma BNF se obecně dá zapsat více než jedním způsobem. Obrázek 14.2 znázorňuje alternativní verzi formy BNF `ATTRIBUTE_FILE`.

```
ATTRIBUTE_FILE ::= ATTRIBUTE+
ATTRIBUTE      ::= NAME '=' VALUE '\n'
NAME           ::= [a-zA-Z]\w*
VALUE          ::= 'true' | 'false' | \d+ | NAME
```

Obrázek 14.2: Alternativní forma BNF pro soubory atributů

Zde jsme znak nového řádku přesunuli na konec neterminálu *ATTRIBUTE*, čímž jsme zjednodušili definici neterminálu *ATTRIBUTE_FILE*. Dále jsme v definici neterminálu *VALUE* opětovně použili neterminál *NAME*, ačkoliv se jedná o pochybnou změnu, protože je jen shodou okolností, že oba neterminály používají stejný regulární výraz. Tato verze formy BNF by měla hledat shodu s naprosto stejným textem jako verze předchozí.

Jakmile máme formu BNF, můžeme ji v mysli nebo na papíře „otestovat". Máme-li například text „depth = 37\n", můžeme postupovat podle formy BNF a zjistit, zda se tento text shoduje, přičemž začneme první neterminálem *ATTRIBUTE_FILE*. Tento neterminál začíná shodou s jiným neterminálem *ATTRIBUTE*. A neterminál *ATTRIBUTE* začíná shodou s dalším neterminálem *NAME*, který se zase musí shodovat s terminálovým regulárním výrazem `[a-zA-Z]\w*`. Tento regulární výraz se skutečně shoduje se začátkem textu, s částí „depth". Další věcí, s níž se musí neterminál *ATTRIBUTE* shodovat, je literál =. Jenže na tomto místě shoda selže, protože za částí „depth" následuje mezera. V tomto místě by měl analyzátor nahlásit, že zadaný text neodpovídá gramatice. V tomto případě musíme buď opravit data odstraněním mezery před a za rovnítkem, nebo změnit gramatiku – například změnou (první) definice neterminálu *ATTRIBUTE* na *NAME* `\s* = \s*` *VALUE*. Po několika testech na papíře a doladění gramatiky bychom měli mít mnohem jasnější představu o tom, s čím se naše forma BNF bude shodovat a s čím se shodovat nebude.

Forma BNF musí být kompletní, jinak není platná, avšak platná forma BNF není nutně správná. Jeden z problémů souvisí s nejednoznačností. Ve výše uvedeném příkladu se literál `true` shoduje nejen s první alternativou neterminálu *VALUE* (`'true'`), ale také s jeho poslední alternativou (`[a-zA-Z]\w*`). Forma BNF je i přesto platná, musí s tím ale počítat analyzátor implementující tuto formu BNF. A jak si ukážeme v pozdější části této lekce, formy BNF mohou být docela záludné, protože někdy definujeme věci pomocí nich samotných. To pak může být dalším zdrojem nejednoznačnosti a vyústit v neanalyzovatelné gramatiky.

Pro stanovení pořadí, ve kterém by se měly aplikovat operátory ve výrazech, jež neobsahují závorky, se používá precedence a asociativita. Precedence se používá v situaci s odlišnými operátory a asociativita v případě stejných operátorů.

Jako příklad precedence můžeme uvést výraz jazyka Python `3 + 4 * 5`, který se vyhodnotí na `23`. To znamená, že operátor `*` má v jazyku Python vyšší precedenci než operátor +, protože tento výraz se chová tak, jako by byl zapsaný ve tvaru `3 + (4 * 5)`. Tuto skutečnost můžeme říci také tak, že „v jazyku Python váže operátor * silněji než operátor +".

Jako příklad precedence vezměme výraz `12 / 3 / 2`, který se vyhodnotí na `2`. To znamená, že operátor `/` je zleva asociativní. Když tedy výraz obsahuje dva nebo více operátorů `/`, vyhodnotí se zleva doprava. Zde se nejdříve část `12 / 3` vyhodnotila na výsledek `4` a poté se část `4 / 2` vyhodnotila na `2`. Na druhou stranu operátor = je zprava asociativní, takže můžeme psát `x = y = 5`. Máme-li tedy

dva nebo více operátorů =, vyhodnotí se zprava doleva, takže nejdříve se vyhodnotí část `y = 5`, čímž se do proměnné `y` přiřadí hodnota `5`, a poté se vyhodnotí část `x = y`, která přiřadí do proměnné `x` stejnou hodnotu. Pokud by operátor = nebyl zprava asociativní, tento výraz by selhal (za předpokladu, že proměnná `y` dosud neexistuje), protože nejdříve by došlo k pokusu o přiřazení hodnoty neexistující proměnné `y` do proměnné `x`.

Precedence a asociativita mohou někdy pracovat společně. Pokud například dva odlišné operátory mají stejnou precedenci (což je běžný případ s operátory + a -), pak bez použití závorek zbývá k určení pořadí vyhodnocení pouze jejich asociativita.

Ve formě BNF lze Precedenci a asociativitu vyjádřit skládáním činitelů do termů a termů do výrazů. Kupříkladu forma BNF na obrázku 14.3 definuje čtyři základní aritmetické operace nad celými čísly i s uzávorkovanými podvýrazy a to vše se správnými precedencemi a asociativitami (zleva doprava).

```
INTEGER        ::= \d+
ADD_OPERATOR   ::= '+' | '-'
SCALE_OPERATOR ::= '*' | '/'
EXPRESSION     ::= TERM (ADD_OPERATOR TERM)*
TERM           ::= FACTOR (SCALE_OPERATOR FACTOR)*
FACTOR         ::= '-'? (INTEGER | '(' EXPRESSION ')')
```

Obrázek 14.3: Forma BNF pro aritmetické operace

Vztah mezi precedencemi je ustaven způsobem, jakým kombinujeme výrazy, termy a činitele, zatímco asociativity ustavujeme strukturou definic neterminálů každého výrazu, termu a činitele.

Potřebujeme-li asociativitu zleva doprava, můžeme použít následující strukturu:

BNF

```
POWER_EXPRESSION ::= FACTOR ('**' POWER_EXPRESSION)
```

Rekurzivní povaha neterminálu *POWER_EXPRESSION* nutí analyzátor postupovat zprava doleva.

Řešení precedence a asociativity se můžeme zcela vyhnout. Stačí, když jednoduše uvedeme, že data nebo jazyk DSL používají závorky, díky nimž budou všechny vztahy vyjádřeny explicitně. To lze sice provést velice snadno, pro uživatele našeho datového formátu nebo jazyka DSL to ale nebude nic příjemného, a proto budeme všude tam, kde to bude vhodné, precedenci a asociativitu používat.*

Se syntaktickou analýzou toho souvisí ještě mnohem více, než co jsme si zde řekli. Nicméně informace v této lekci by měly být pro začátek dostatečné, třebaže doplňková četba bude pro ty, kteří plánují tvorbu komplexních a sofistikovaných analyzátorů, jistě vhodná.

Syntaxi formy BNF a něco z terminologie používané v oblasti syntaktické analýzy nyní již známe, a proto si napíšeme několik analyzátorů, přičemž začneme jejich ruční tvorbou.

* Další možností, jak se vyhnout precedenci a asociativitě (a přitom nevyžadovat závorky), je použít prefixovou nebo postfixovou notaci (viz *http://cs.wikipedia.org/wiki/Prefixová_notace*).

Ruční tvorba analyzátorů

V této části budeme vyvíjet tři ručně psané analyzátory. První je jen o malinko víc než rozšíření regulárního výrazu pro data ve tvaru klíč-hodnota z předchozí lekce, je na něm ale dobře vidět infrastruktura nezbytná pro použití takového regulárního výrazu. Druhý analyzátor je také založen na regulárním výrazu, ve skutečnosti se ale jedná o konečný automat, protože má dva stavy. Oba tyto příklady představují analyzátory dat. Třetím příkladem je analyzátor pro jazyk DSL. V něm používáme rekurzivní sestup, poněvadž analyzovaný jazyk DSL umožňuje vnořování výrazů. V pozdějších částech vyvineme nové verze těchto analyzátorů pomocí nástrojů PyParsing a PLY a zvláště u našeho jazyka DSL uvidíme, o kolik snazší je použití generického generátoru analyzátorů ve srovnání s jejich ruční tvorbou.

Analyzování jednoduchých dat ve tvaru klíč-hodnota

Příklady ke knize obsahují program s názvem `playlists.py`. Tento program čte seznam skladem ve formátu `.m3u` (rozšířená adresa URL pro MPEG Audio Layer 3) a vypisuje ekvivalentní seznam skladem ve formátu `.pls` (Play List 2) a obráceně. V tomto oddílu napíšeme analyzátor pro formát `.pls` a v následujícím oddílu napíšeme analyzátor pro formát `.m3u`. Oba analyzátory jsou psané ručně a používají regulární výrazy.

Analyzátor dat ve tvaru klíč-hodnota na bázi nástroje PyParsing ➤ 515

Formát `.pls` je v podstatě stejný jako formát `.ini` systému Windows, takže pro jeho analýzu bychom správně měli použít modul `configparser` ze standardní knihovny. Nicméně formát `.pls` je ideální pro vytvoření našeho prvního datového analyzátoru, protože se díky jeho jednoduchosti můžeme zaměřit na aspekty analýzy, a proto v tomto případě modul `configparser` nepoužijeme.

Analyzátor dat ve tvaru klíč-hodnota na bázi nástroje PLY ➤ 530

Začneme pohledem na kratičký úryvek ze souboru `.pls`, abychom získali určitou představu o datech, potom vytvoříme formu BNF a poté vytvoříme analyzátor pro čtení těchto dat. Úryvek je zachycen na obrázku 14.4.

```
[playlist]
File1=Blondie\Atomic\01-Atomic.ogg
Title1=Blondie - Atomic
Length1=230
...
File18=Blondie\Atomic\18-I'm Gonna Love You Too.ogg
Title18=Blondie - I'm Gonna Love You Too
Length18=-1
NumberOfEntries=18
Version=2
```

Obrázek 14.4: Úryvek ze souboru .pls

Většinu dat jsme vynechali a uvedli místo nich výpustek (. . .). Máme zde jen jeden řádek s hlavičkou ve stylu souboru `.ini` (`[playlist]`), přičemž všechny ostatní záznamy jsou v jednoduchém formátu *`klíč=hodnota`*. Klíče obvykle vypadají tak, že se jejich názvy opakují, ale kvůli jedinečnosti jsou doplněny čísly. Pro každou skladbu se uchovávají tři údaje: název souboru (v tomto příkladu s oddělova-

či cesty systému Windows), titul a délka v sekundách. V tomto příkladu má první skladba známou délku, ale délka posledního záznamu je neznámá, což je zachyceno záporným číslem.

Forma BNF, kterou jsme vytvořili, dokáže zpracovat soubory `.pls` a je navíc dostatečně generická, aby si poradila i s podobnými formáty ve tvaru klíč-hodnota. Naše forma BNF je zachycena na obrázku 14.5.

```
PLS         ::= (LINE '\n')+
LINE        ::= INI_HEADER | KEY_VALUE | COMMENT | BLANK
INI_HEADER ::= '[' [^]]+ ']'
KEY_VALUE   ::= KEY \s* '=' \s* VALUE?
KEY         ::= \w+
VALUE       ::= .+
COMMENT     ::= #.*
BLANK       ::= ^$
```

Obrázek 14.5: Forma BNF pro souborový formát `.pls`

Forma BNF pro soubor atributů ➤ 494

Tato forma BNF definuje formát `.pls` (neterminál *PLS*) jako jeden či více řádků (neterminálů *LINE*) se znakem nového řádku. Každý řádek může být tvořen hlavičkou (neterminál *INI_HEADER*), dvojicí klíč-hodnota (neterminál *KEY_VALUE*), komentářem (neterminál *COMMENT*) nebo prázdným místem (neterminál *BLANK*). Hlavička je definována jako otevírací hranatá závorka, za níž následuje jeden či více znaků (kromě uzavírací hranaté závorky), za nimiž následuje uzavírací hranatá závorka (tyto řádky budeme přeskakovat). Dvojice klíč-hodnota se od příkladu s formou BNF pro soubor atributů (*ATTRIBUTE_FILE*) uvedeného v předchozí části liší v tom, že hodnota (neterminál *VALUE*) je volitelná. Dále zde povolujeme bílé místo před i po rovnítku (=). To znamená, že v této formě BNF je například řádek „title5=\n“ platný, stejně jako kupříkladu řádek „length=126\n“, jehož platnost bychom očekávali. Klíč (neterminál *KEY*) je posloupnost jednoho či více alfanumerických znaků a hodnota (neterminál *VALUE*) je libovolná posloupnost znaků. Komentáře (neterminál *COMMENTS*) jsou podobné jako v jazyku Python – a všechny je budeme přeskakovat. Prázdné řádky (neterminál *BLANK*) jsou povolené, budeme je ale přeskakovat.

Smyslem našeho analyzátoru je naplnit slovník prvky ve tvaru klíč-hodnota odpovídajícími těm, které jsou uloženy v souboru, ovšem s klíči převedenými na malá písmena. Program `playlists.py` používá tento analyzátor pro získání slovníku se seznamem skladeb, který poté vypíše v požadovaném formátu. Samotný program `playlists.py` si zde rozebírat nebudeme, protože s analýzou jako takovou nijak nesouvisí, a tak jako tak jej lze stáhnout z webové stránky této knihy.

Syntaktická analýza probíhá v jediné funkci, která přijímá objekt představující otevřený soubor (`file`) a logickou hodnotou (`lowercase_keys`) s výchozí hodnotou je `False`. Funkce použije dva regulární výrazy a naplní slovník (`key_values`), který vrátí. Podíváme se na regulární výrazy a poté na kód, který analyzuje řádky soubory a naplňuje slovník.

```
INI_HEADER = re.compile(r"^\[[^]]+\]$")
```

Přestože hlavičky formátu `.ini` chceme ignorovat, musíme je i tak umět identifikovat. Tento regulární výraz nepovoluje žádné úvodní ani koncové bílé místo, což nevadí, protože budeme bílé místo z každého načítaného řádku odřezávat. Samotný regulární výraz se shoduje se začátkem řádku, potom

s otevírací hranatou závorkou, pak s jedním či více znaky (kromě uzavíracích hranatých závorek), poté s uzavírací hranatou závorkou, a nakonec s koncem řádku.

```
KEY_VALUE_RE = re.compile(r"^(?P<key>\w+)\s*=\s*(?P<value>.*)$")
```

Regulární výraz `KEY_VALUE_RE` povoluje bílé místo kolem rovnítka (=), zachytáváme ale pouze skutečný klíč a hodnotu. Hodnota je kvantifikována `*`, může být tedy prázdná. Dále používáme pojmenovaná zachycení, poněvadž jsou čitelnější a snadněji se udržují, což je dáno tím, že je nijak neovlivní přidávané či odstraňované zachycované skupiny, s čímž bychom museli počítat, pokud bychom pro identifikaci zachycovaných skupin použili čísla.

Regulární výraz pro text ve tvaru klíč-hodnota ➤ 475

```
key_values = {}
for lino, line in enumerate(file, start=1):
    line = line.strip()
    if not line or line.startswith("#"):
        continue
    key_value = KEY_VALUE_RE.match(line)
    if key_value:
        key = key_value.group("key")
        if lowercase_keys:
            key = key.lower()
        key_values[key] = key_value.group("value")
    else:
        ini_header = INI_HEADER.match(line)
        if not ini_header:
            print("Chyba při analýze řádku {0}: {1}".format(lino,
                                                            line))
```

Obsah souboru zpracováváme řádek po řádku pomocí vestavěné funkce `enumerate()`, která vrací dvojici tvořenou číslem řádku (začínajícím v rámci textových souborů tradičně od 1) a samotným řádkem. Ořežeme bílé místo, díky čemuž můžeme ihned přeskočit prázdné řádky (a použít jednodušší regulární výrazy). Kromě toho přeskakujeme také řádky s komentářem.

Funkce `enumerate()` ➤ 140

Očekáváme, že většina řádků bude mít tvar *`klíč=hodnota`*, a proto nejdříve zkoušíme shodu s regulárním výrazem `KEY_VALUE_RE`. Je-li úspěšná, extrahujeme klíč a převedeme jej na malá písmena, je-li to nutné. Potom přidáme klíč a hodnotu do slovníku.

Pokud řádek nemá tvar *`klíč=hodnota`*, vyzkoušíme shodu s hlavičkou formátu `.ini`. Obdržíme-li shodu, tak ji jednoduše ignorujeme a pokračujeme na další řádek. V opačném případě vypíšeme chybu. (Bylo by docela přímočaré vytvořit slovník, jehož klíči jsou hlavičky formátu `.ini` a hodnotami slovníky příslušných dvojic klíč-hodnota. Pokud bychom ale chtěli zajít tak daleko, pak bychom už skutečně museli použít `configparser`.)

Regulární výrazy a kód jsou docela jednoduché, vzájemně však na sobě závisejí. Pokud bychom totiž z každého řádku neodřízli bílé místo, museli bychom změnit regulární výrazy, aby povolaly úvodní i koncové bílé místo. Zde je pro nás pohodlnější bílé místo odříznout, mohou být ale situace, v nichž budeme postupovat opačně – neexistuje zde totiž žádný jediný správný přístup.

Na konci (výše uvedená ukázka konec neobsahuje) jednoduše vracíme slovník `key_values`. Nevýhodou použití slovníku v tomto konkrétním případě je, že každá dvojice klíč-hodnota je odlišná, zatímco ve skutečnosti spolu prvky končící stejným číslem souvisejí (např. „title12“, „file12“ a „lenght12”). Program `playlists.py` obsahuje funkci (jedná se o funkci `songs_from_dictionary()`, kterou jsme si zde neukázali, najdete ji však mezi zdrojovými kódy ke knize), která načte slovník s dvojicemi klíč-hodnota ve stejném stylu, jaký vrací výše uvedený kód, a vrátí seznam n-tic se skladbami, což je něco, co budeme v následujícím oddílu provádět přímo.

Analyzování seznamu skladeb

Analyzátor formátu .m3u na bázi nástroje PyParsing ➢ 516

Program `playlists.py` zmíněný v předchozím oddílu dokáže číst a zapisovat soubory ve formátu `.pls`. V tomto oddílu vytvoříme analyzátor, který umí číst soubory ve formátu `.m3u` a který vrací své výsledky ve formě seznamu objektů typu `collections.namedtuple()`, z nichž každý uchovává titul, délku v sekundách a název souboru.

Začneme jako obvykle pohledem na úryvek dat, která chceme analyzovat, poté vytvoříme vhodnou formu BNF a nakonec vytvoříme analyzátor pro analýzu těchto dat. Úryvek dat zachycuje obrázek 14.6.

Analyzátor formátu .m3u na bázi nástroje PLY ➢ 532

```
#EXTM3U
#EXTINF:230,Blondie - Atomic
Blondie\Atomic\01-Atomic.ogg
...
#EXTINF:-1,Blondie - I'm Gonna Love You Too
Blondie\Atomic\18-I'm Gonna Love You Too.ogg
```

Obrázek 14.6: Úryvek ze souboru .m3u

Většinu dat jsme vynechali a uvedli místo nich výpustek (. . .). Soubor musí začínat řádkem `#EXTM3U`. Každý záznam zabírá dva řádky. První řádek záznamu začíná textem `#EXTINF:` a obsahuje délku v sekundách a titul. Druhý řádek záznamu obsahuje název souboru. Podobně jako ve formátu `.pls` záporná délka i zde označuje neznámou dobu trvání.

Forma BNF je znázorněna na obrázku 14.7. Definuje formát `.m3u` (neterminál *M3U*) jako textový literál `#EXTM3U` následovaný novým řádkem a poté jedním či více záznamy (neterminál *ENTRY*). Každý záznam se skládá z informací (neterminál *INFO*), za nimiž následuje nový řádek, poté název souboru (neterminál *FILENAME*) a nakonec nový řádek. Informace začínají textovým literálem `#EXTINF:`, za nímž následuje délka ve formě vteřin (neterminál *SECONDS*), poté čárka a pak titul (neterminál *TITLE*). Vteřiny jsou definovány jako volitelné znaménko minus následované jednou či více číslicemi. Titul a název souboru jsou volně definovány jako posloupnost libovolných znaků kromě znaků nového řádku.

```
M3U      ::= '#EXTM3U\n' ENTRY+
ENTRY    ::= INFO '\n' FILENAME '\n'
INFO     ::= '#EXTINF:' SECONDS ',' TITLE
SECONDS  ::= '-'? \d+
TITLE    ::= [^\n]+
FILENAME ::= [^\n]+
```

Obrázek 14.7: Forma BNF pro formát `.m3u`

Před pohledem na samotný analyzátor se nejdříve podíváme na pojmenovanou n-tici, kterou použijeme pro uložení každého výsledku:

Pojmenovaná n-tice ➤ 113

```
Song = collections.namedtuple("Song", "title seconds filename")
```

Tento přístup je mnohem pohodlnější ve srovnání se slovníkem s klíči ve tvaru „file5“, „title17“ a tak dále, u kterého musíme napsat kód pro spárování všech klíčů, jež končí stejným číslem.

Kód analyzátoru prostudujeme pro snazší výklad ve čtyřech krátkých částech.

```
if fh.readline() != "#EXTM3U\n":
    print("This is not a .m3u file")
    return []
songs = []
INFO_RE = re.compile(r"#EXTINF:(?P<seconds>-?\d+),(?P<title>.+)")
title = seconds = None
WANT_INFO, WANT_FILENAME = range(2)
state = WANT_INFO
```

Objekt představující otevřený soubor je v proměnné `fh`. Pokud soubor nezačíná správným textem pro formát `.m3u`, vypíšeme chybovou zprávu a vrátíme prázdný seznam.

Pojmenované n-tice `Song` budou ukládány do seznamu `songs`. Regulární výraz slouží pro hledání shody s neterminálem *`INFO`* naší formy BNF. Samotný analyzátor se vždy nachází v jednom ze dvou stavů: `WANT_INFO` (počáteční stav) nebo `WANT_FILENAME`. Ve stavu `WANT_INFO` se analyzátor pokouší získat titul a vteřiny a ve stavu `WANT_FILENAME` analyzátor vytváří novou skladbu (n-tici `Song`) a přidává ji do seznamu `songs`.

```
for lino, line in enumerate(fh, start=2):
    line = line.strip()
    if not line:
        continue
```

Procházíme každý řádek zadaného otevřeného souboru podobným způsobem jako v případě analyzátoru pro formát `.pls` v předchozím oddílu, tentokráte ale začínáme počítat řádky od 2, protože řádek 1 jsme zpracovali ještě před vstupem do cyklu. Odřežeme bílé místo a přeskočíme prázdné řádky; další zpracování závisí na aktuálním stavu.

```
    if state == WANT_INFO:
        info = INFO_RE.match(line)
        if info:
            title = info.group("title")
            seconds = int(info.group("seconds"))
            state = WANT_FILENAME
        else:
            print("Chyba při analýze řádku {0}: {1}".format(
                  lino, line))
```

Pokud očekáváme řádek s informacemi, pokusíme se nalézt shodu s regulárním výrazem `INFO_RE` a extrahovat titul a počet vteřin. Poté změníme stav analyzátoru tak, aby očekával na následujícím řádku odpovídající název souboru. Chování převodní funkce `int()` kontrolovat nemusíme (např. pomocí bloku `try ... except`), protože text použitý pro převod odpovídá vždy díky regulárnímu výrazu `-?\d+` platnému číslu.

```
elif state == WANT_FILENAME:
    songs.append(Song(title, seconds, line))
    title = seconds = None
    state = WANT_INFO
```

Pokud očekáváme řádek s názvem souboru, pak jednoduše připojíme novou n-tici typu `Song` s dříve nastaveným titulem a vteřinami a s aktuálním řádkem jako názvem souboru. Potom vrátíme stav analyzátoru do jeho původního stavu, v němž je připraven analyzovat údaje o další skladbě.

Na konci (výše uvedené ukázka konec neobsahuje) jednoduše vracíme slovník `songs` volajícímu. A díky pojmenovaným n-ticím lze k atributům každé skladby pohodlně přistupovat podle názvu (např. `songs[12].title`).

Sledování stavu pomocí proměnné jako v tomto případě funguje v mnoha jednoduchých případech skvěle. Ovšem obecně je tento přístup při práci s daty nebo jazyky DSL, jež mohou obsahovat vnořené výrazy, nedostatečný. V následujícím oddílu si ukážeme, jak udržovat stav tváří v tvář vnořování.

Analýza bloků jakožto doménově specifického jazyka

Analyzátor bloků na bázi nástroje PyParsing ➤ 518

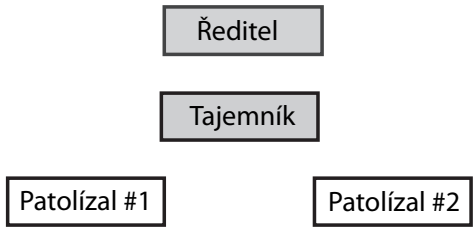
Program `blocks.py` je k dispozici jako jeden z příkladů v rámci zdrojových kódů k této knize. Čte jeden nebo více souborů `.blk` zadaných na příkazovém řádku, které používají náš vlastní textový formát (blokový formát, smyšlený jazyk), a pro každý z nich vytvoří soubor SVG (Scalable Vector Graphics – škálovatelná vektorová grafika) se stejným názvem, ale s příponou změněnou na `.svg`. Vykreslené soubory typu SVG sice nebudou příliš pěkné, poskytují však dobrou vizuální reprezentaci, jež usnadňuje rozpoznání chyb v souborech `.blk` a zároveň ukazuje možnosti nabízené i docela jednoduchým jazykem DSL.

Analyzátor bloků na bázi nástroje PLY ➤ 534

Obrázek 14.8 ukazuje celý soubor `hierarchy.blk` a obrázek 14.9 ukazuje, jak se vykreslí soubor `hierarchy.svg` vytvořený programem `blocks.py`.

```
[] [lightblue: Ředitel]
//
[] [lightgreen: Tajemník]
//
[Patolízal #1] [] [Patolízal #2]
```

Obrázek 14.8: Soubor hierarchy.blk



Obrázek 14.9: Soubor hierarchy.svg

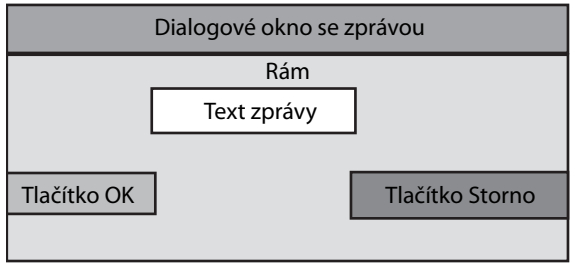
Blokový formát je v podstatě tvořen dvěma prvky: bloky a značkami nového řádku. Bloky jsou uzavřeny do hranatých závorek. Mohou být i prázdné – v takovém případě se použijí jako oddělovače zabírající jednu buňku pomyslné mřížky. Bloky mohou dále obsahovat text a volitelně barvu. Značkami nového řádku jsou lomítka, která označují místo, kde by měl začít nový řádek. Na obrázku 14.8 jsou na obou místech dvě značky nového řádku, díky čemuž vzniknou dva prázdné řádky, které jsou patrné na obrázku 14.9.

Blokový formát dále umožňuje vnořování bloků do sebe. Stačí mezi hranaté závorky nějakého bloku umístit za jeho text další bloky a značky nového řádku.

Obrázek 14.10 ukazuje kompletní soubor `messagebox.blk` obsahující vnořené bloky a obrázek 14.11 ukazuje, jak vypadá vykreslený soubor `messagebox.svg`.

```
[#00CCDE: Dialogové okno se zprávou
[lightgray: Rám
[] [white: Text zprávy]
//
[goldenrod: Tlačítko OK] [] [#ff0505: Tlačítko Storno]
/
[]
]
]
```

Obrázek 14.10: Soubor messagebox.blk



Obrázek 14.11: Soubor messagebox.svg

Barvy lze specifikovat pomocí názvů podporovaných formátem SVG nebo jako šestnáctkové hodnoty (včetně úvodního znaku #). Blokový soubor uvedený na obrázku 14.10 má jeden vnější blok („Dia-

logové okno se zprávou“), vnitřní blok („Rám“) a uvnitř několik bloků společně se značkami nového řádku. Bílé místo používáme čistě pro přehlednost, blokový formát jej zcela ignoruje.

Ukázali jsme si několik blokových souborů, a proto se nyní podíváme na příslušnou formu BNF, abychom formálněji pochopili, z čeho se skládá platný blokový soubor, a zároveň se připravili na analýzu tohoto rekurzivního formátu. Forma BNF je zachycena na obrázku 14.12.

```
BLOCKS  ::= NODES+
NODES   ::= NEW_ROW* \s* NODE+
NODE    ::= '[' \s* (COLOR ':')? \s* NAME? \s* NODES*
\s* ']'
COLOR   ::= '#' [\dA-Fa-f]{6} | [a-zA-Z]\w*
NAME    ::= [^][/]+
NEW_ROW ::= '/'
```

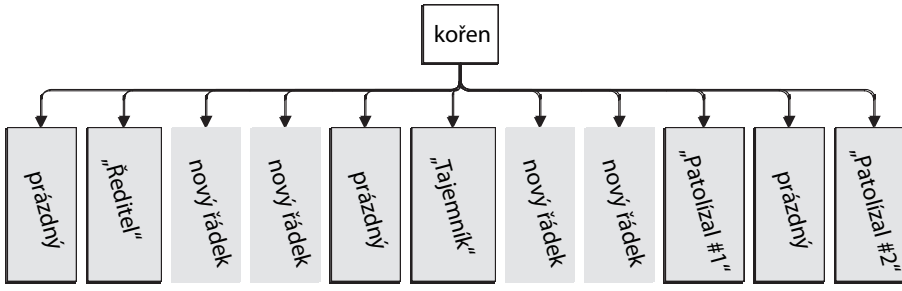
Obrázek 14.12: Forma BNF pro formát .blk

Tato forma BNF definuje blokový soubor (neterminál `BLOCKS`) jako jeden nebo více uzlů (neterminál `NODES`). Uzly se skládají z žádného či více nových řádků (neterminál `NEW_ROW`), za nimiž následuje jeden či více uzlů (neterminál `NODE`). Uzel je levá hranatá závorka následovaná volitelnou barvou (neterminál `COLOR`), za kterou následuje volitelný název (neterminál `NAME`), žádný či více uzlů a pravá hranatá závorka. Barva je tvořena znakem mřížky (`'#'`), za nímž následuje šest šestnáctkových číslic a dvojtečka nebo posloupnost jednoho či více alfanumerických znaků začínajících alfabetickým znakem a dvojtečka. Název je posloupnost libovolných znaků kromě hranatých závorek nebo lomítka. Nový řádek je literál lomítka. Jak je z hojného výskytu regulárního výrazu `\s*` patrné, bílé místo může být všude mezi terminály i neterminály a nemá žádný zvláštní význam.

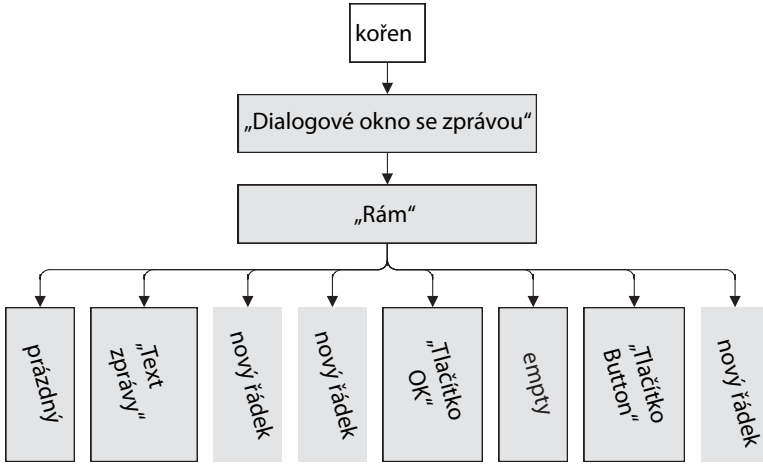
Definice neterminálu `NODE` je rekurzivní, protože obsahuje neterminál `NODES`, který je definován pomocí neterminálu `NODE`. V rekurzivních definicích, jako je tato, snadno vznikají chyby, což může vést k tomu, že analyzátory skončí v nekonečné smyčce. Proto je dobré provést několik testů na papíře a ujistit se, že gramatika je konečná, tj. že při zadání platného neterminálu se nakonec všechny neterminály přepíší na terminály a nedojde k nekonečnému cyklu od jednoho neterminálu k druhému.

Dříve jsme se po vytvoření formy BNF pustili přímo do tvorby analyzátoru a zpracování jsme prováděli v průběhu samotné analýzy. To ale není pro rekurzivní gramatiky kvůli možnosti vnořování jednotlivých prvků vhodné. Musíme totiž nejdříve vytvořit třídu reprezentující každý blok (nebo nový řádek), která bude uchovávat seznam vnořených podřízených bloků, které samy mohou obsahovat potomky a tak dále. Výsledky analyzátoru získáme ve formě seznamu (který může pro reprezentaci vnořených bloků obsahovat seznamy uvnitř seznamů), který můžeme převést na strom s „prázdným“ kořenovým blokem a všemi ostatními bloky jako jeho potomky.

V případě souboru `hierarchy.blk` obsahuje kořenový blok seznam nových řádků a podřízených bloků (včetně prázdných bloků), z nichž žádný nemá další potomky. Tuto situaci zachycuje obrázek 14.13 – soubor `hierarchy.blk` jsme si ukázali již dříve (strana 502). Ukázkový soubor `messagebox.blk` má kořenový blok, jenž obsahuje jeden podřízený blok („Dialogové okno se zprávou“), který sám obsahuje jeden podřízený blok („Rám“), který zase obsahuje seznam nových řádků a podřízených bloků (včetně prázdných bloků) uvnitř bloku „Rám“. Tuto strukturu znázorňuje obrázek 14.14 – soubor `messagebox.blk` jsme si ukázali již dříve (strana 503).



Obrázek 14.13: Bloky analyzovaného souboru hierarchy.blk



Obrázek 14.14: Bloky analyzovaného souboru messagebox.blk a jejich potomci

Všechny blokové analyzátory z této lekce vracejí kořenový blok s podřízenými bloky (viz obrázky 14.13 a 14.14), je-li ovšem analýza úspěšná. Modul `BlockOutput.py`, který používá program `blocks.py`, poskytuje funkci s názvem `save_blocks_as_svg()`, která přijímá kořenový blok a rekurzivním průchodem přes jeho potomky vytváří soubor typu SVG pro vizuální reprezentaci těchto bloků.

Ještě než vytvoříme analyzátor, definujeme si třídu `Block`, která bude reprezentovat blok a libovolné v něm obsažené podřízené bloky. Poté se podíváme na analyzátor a ukážeme si, jak vytváří jediný kořenový objekt typu `Block`, jehož podřízené bloky reprezentují obsah analyzovaného souboru `.blk`.

Instance třídy `Block` mají tři atributy: `name` (název), `color` (barva) a `children` (seznam potomků, který může být prázdný). Kořenový blok nemá žádný název ani barvu a prázdný blok nemá název, jeho barva je však bílá. Seznam `children` obsahuje objekty typu `Block` a hodnoty `None`, které představují značky nového řádku. Nebudeme vyžadovat po uživateli třídy `Block`, aby si všechny tyto konvence pamatoval, a proto doplníme ještě několik metod modulu poskytujících určitou úroveň abstrakce.

```
class Block:

    def __init__(self, name, color="white"):
        self.name = name
        self.color = color
        self.children = []

    def has_children(self):
        return bool(self.children)
```

Třída `Block` je velice jednoduchá. Metoda `has_children()` je doplňkovou metodou pro modul `BlockOutput.py`. Není zde žádné explicitní rozhraní API pro přidávání potomků, poněvadž očekáváme, že klienti budou pracovat přímo s atributem `children`.

```
get_root_block = lambda: Block(None, None)
get_empty_block = lambda: Block("")
get_new_row = lambda: None
is_new_row = lambda x: x is None
```

Lambda funkce ➤ 180

Tyto čtyři kratičké pomocné funkce poskytují abstrakci pro pohodlnější použití třídy `Block`. Programátoři používající modul `Block` si tedy nemusejí pamatovat považované konvence, ale jen tyto funkce. Kromě toho nám poskytují určitý manévrovací prostor, pokud bychom se rozhodli tyto konvence později změnit.

Třídu `Block` a podpůrné funkce máme připraveny (vše v modulu `Block.py` importovaném programem `blocks.py`, jenž obsahuje analyzátor), a proto se můžeme pustit do tvorby analyzátoru souborů `.blk`. Náš analyzátor vytvoří kořenový blok a naplní jej potomky (a potomky potomků a tak dále). Kořenový blok bude reprezentovat soubor `.blk` a lze jej předat funkci `BlockOutput.save_blocks_as_svg()`.

Nyní vytvoříme rekurzivně sestupný analyzátor, což je nezbytné, protože blokový formát může obsahovat vnořené bloky. Analyzátor se skládá ze třídy `Data`, která je inicializována textem analyzovaného souboru a která udržuje aktuální pozici v rámci analýzy a poskytuje metody pro postupování textem. Analyzátor kromě toho obsahuje skupinu analyzujících funkcí, jež pracují na instanci třídy `Data`, postupně zpracovávají data a naplňují zásobník objektů typu `Block`. Některé z těchto funkcí volají rekurzivně ostatní funkce, což odráží rekurzivní povahu dat, což je též reflektováno ve formě BNF.

Začneme pohledem na třídu Data, pak se podíváme na to, jak se tato třída používá a jak se zahajuje analýza, a poté si postupně rozebereme každou analyzující funkci.

```
    class Data:

        def __init__(self, text):
            self.text = text
            self.pos = 0
            self.line = 1
            self.column = 1
            self.brackets = 0
            self.stack = [Block.get_root_block()]
```

Třída Data uchovává text analyzovaného souboru, pozici, na níž se nacházíme (`self.pos`), a řádek (počítaný od 1) společně se sloupcem, které tato pozice reprezentuje. Dále sleduje výskyty hranatých závorek (pro každou otevírací hranatou závorku přičte jedničku a pro každou uzavírací jedničku odečte). Objekt `self.stack` obsahuje seznam objektů typu `Block` a inicializujeme jej prázdným kořenovým blokem, jejž na konci vrátíme. Pokud byla analýza úspěšná, bude tento blok obsahovat podřízené bloky (které mohou mít své vlastní podřízené bloky a tak dále) reprezentující bloková data.

```
def location(self):
    return "řádek {0}, sloupec {1}".format(self.line,
                                           self.column)
```

Tato malička metoda vrací aktuální lokaci jako řetězec obsahující číslo řádku a sloupce.

```
def advance_by(self, amount):
    for x in range(amount):
        self._advance_by_one()
```

Analyzátor musí při analýze textu tento text postupně procházet. Pro usnadnění práce máme k dispozici několik metod nabízejících postup v analyzovaném textu. Tato metoda postoupí o zadaný počet znaků.

```
def _advance_by_one(self):
    self.pos += 1
    if (self.pos < len(self.text) and
        self.text[self.pos] == "\n"):
        self.line += 1
        self.column = 1
    else:
        self.column += 1
```

Všechny metody zajišťující postup v analyzovaném textu používají pro samotný posun pozice v textu tuto soukromou metodu. To znamená, že kód udržující hodnoty řádku a sloupce aktuální je na jednom místě.

```
def advance_to_position(self, position):
    while self.pos < position:
        self._advance_by_one()
```

Tato metoda postoupí na zadanou indexovou pozici v textu, k čemuž se opět použije soukromá metoda `_advance_by_one()`.

```
def advance_up_to(self, characters):
    while (self.pos < len(self.text) and
           self.text[self.pos] not in characters and
           self.text[self.pos].isspace()):
        self._advance_by_one()
    if not self.pos < len(self.text):
        return False
```

```
        if self.text[self.pos] in characters:
            return True
        raise LexError("očekáváno '{0}' obdrženo '{1}'"
                       .format(characters, self.text[self.pos]))
```

Tato metoda postupuje přes bílé znaky, dokud znak na aktuální pozici neodpovídá jednomu z těch, které jsou zadány v řetězci `characters`. Oproti ostatním metodám tohoto typu se liší v tom, že může selhat (poněvadž může narazit na znak, který není bílým místem a který není mezi očekávanými znaky). Tato metoda vrací logickou hodnotu signalizující, zda uspěla.

```
class LexError(Exception): pass
```

Tuto třídu představující výjimku používá interně náš analyzátor. Raději používáme vlastní výjimku než řekněme výjimku `ValueError`, protože tak při ladění snadněji odlišíme naše vlastní výjimky od výjimek Pythonu.

```
data = Data(text)
try:
    parse(data)
except LexError as err:
    raise ValueError("Chyba {{0}}:{0}: {1}".format(
                     data.location(), err))
return data.stack[0]
```

Analýza je z pohledu nejvyšší úrovně docela jednoduchá. Vytvoříme instanci třídy `Data` na základě textu, který chceme analyzovat, a poté zavoláme funkci `parse()` (na ni se podíváme za okamžik), která provede analýzu. Pokud dojde k nějaké chybě, volá se naše vlastní výjimka `LexError`, kterou jednoduše převedeme na výjimky `ValueError`, čímž izolujeme volajícího od interně používaných výjimek. Chybová zpráva obvykle obsahuje zakódovaný název pole pro metodu `str.format()` – od volajícího se očekává, že jej použije pro vložení názvu souboru, což zde nemůžeme, protože máme k dispozici jen text souboru, a ne název souboru nebo objekt představující soubor.

Na konci vrátíme kořenový blok, který by měl mít potomky (a jejich potomky) reprezentující analyzované bloky.

```
def parse(data):
    while data.pos < len(data.text):
        if not data.advance_up_to("[]/"):
            break
        if data.text[data.pos] == "[":
            data.brackets += 1
            parse_block(data)
        elif data.text[data.pos] == "/":
            parse_new_row(data)
        elif data.text[data.pos] == "]":
            data.brackets -= 1
            data.advance_by(1)
        else:
```

```
            raise LexError("očekávám '[', ']', nebo '/'; "
                    "ale obdržel jsem '{0}'".format(data.text[data.pos]))
    if data.brackets:
        raise LexError("při očekávání '{0}' došel text"
                       .format(']' if data.brackets > 0 else '['))
```

Tato funkce je srdcem rekurzivně sestupného analyzátoru. Prochází text a hledá začátek nebo konec bloku nebo značku nového řádku. Narazí-li na začátek bloku, zvýší počet hranatých závorek a zavolá funkci `parse_block()`. Narazí-li na značku nového řádku, zavolá funkci `parse_new_row()`. A narazí-li na konec bloku, sníží počet hranatých závorek a postoupí na další znak. Pokud narazí na jakýkoliv jiný znak, jedná se o chybu, kterou odpovídajícím způsobem nahlásí. Podobně nahlásí chybu i po analýze všech dat, pokud je počet hranatých závorek nenulový.

```
def parse_block(data):
    data.advance_by(1)
    nextBlock = data.text.find("[", data.pos)
    endOfBlock = data.text.find("]", data.pos)
    if nextBlock == -1 or endOfBlock < nextBlock:
        parse_block_data(data, endOfBlock)
    else:
        block = parse_block_data(data, nextBlock)
        data.stack.append(block)
        parse(data)
        data.stack.pop()
```

Tato funkce začíná postoupením o jeden znak (čímž přeskočí otevírací hranatou závorku představující začátek bloku). Poté se podívá po dalším začátku bloku a dalším konci bloku. Pokud nenalezne žádný následující blok nebo pokud je další konec bloku před začátkem jiného bloku, pak tento blok nemá žádné vnořené bloky, a proto můžeme jednoduše zavolat funkci `parse_block_data()` a předat jí koncovou pozici konce tohoto bloku.

Má-li tento blok uvnitř jeden nebo více vnořených bloků, analyzujeme jeho data až po místo, kde začíná jeho první vnořený blok. Potom tento blok umístíme do zásobníku bloků a rekurzivně zavoláme funkci `parse()` pro analýzu vnořeného bloku (nebo bloků – a jejich vnořených bloků atd.). A na konci tento blok vytáhneme ze zásobníku, neboť o celé vnořování se postará rekurzivní volání.

```
def parse_block_data(data, end):
    color = None
    colon = data.text.find(":", data.pos)
    if -1 < colon < end:
        color = data.text[data.pos:colon]
        data.advance_to_position(colon + 1)
    name = data.text[data.pos:end].strip()
    data.advance_to_position(end)
    if not name and color is None:
        block = Block.get_empty_block()
    else:
```

```
            block = Block.Block(name, color)
        data.stack[-1].children.append(block)
        return block
```

Tato funkce se používá pro analýzu dat jednoho bloku (až po zadaný koncový bod v textu) a pro přidání odpovídajícího objektu typu `Block` do zásobníku bloků.

Začneme pokusem o nalezení barvy a v případě úspěchu postoupíme v textu za ni. Dále se pokusíme najít text bloku (jeho název), který ale může být oprávněně prázdný. Máme-li blok bez názvu nebo barvy, vytvoříme prázdný objekt typu `Block`. V opačném případě vytvoříme objekt typu `Block` se zadaným názvem a barvou.

Jakmile byl objekt typu `Block` vytvořen, přidáme jej jako posledního potomka bloku umístěného na vrcholu zásobníku bloků. (Na začátku je na vrcholu kořenový blok, pokud ale máme vnořené bloky, může jít o jiný blok, který byl umístěn na vrchol zásobníku.) Na konci vrátíme blok, který může být umístěn do zásobníku bloků, což provedeme jen tehdy, obsahuje-li další vnořené bloky.

```
def parse_new_row(data):
    data.stack[-1].children.append(Block.get_new_row())
    data.advance_by(1)
```

Jedná se o nejjednodušší z funkcí určených pro analýzu. Jednoduše přidá nový řádek jako posledního potomka bloku na vrcholu zásobníku bloků a postoupí dále přes znak nového řádku.

Dostali jsme se na konec rekurzivně sestupného analyzátoru bloků. Tento analyzátor sice nevyžaduje obrovské množství kódu (jedná se o méně než 100 řádků), přesto se jedná o více než 50 procent více řádků než verze využívající nástroj PyParsing a přibližně o 33 procent více řádků než verze využívající nástroj PLY. A jak uvidíme, použití nástroje PyParsing nebo PLY je mnohem jednodušší než ruční tvorba rekurzivně sestupného analyzátoru. Kromě toho díky těmto modulům získáme analyzátory, které se mnohem snadněji udržují.

Převod do souboru typu SVG pomocí funkce `BlockOutput.save_blocks_as_svg()` je stejný pro všechny blokové analyzátory, protože všechny vytvářejí stejné struktury ve formě kořenového bloku a potomků. Kód této funkce si zde rozebírat nebudeme, poněvadž se syntaktickou analýzou nijak nesouvisí. Nachází se v modulu `BlockOutput.py`, který je součástí zdrojových kódů k této knize.

Dokončili jsme prohlídku ručně vytvářených analyzátorů. V následujících dvou částech si ukážeme jejich verze využívající nástroje PyParsing a PLY. Kromě toho si ukážeme analyzátor pro jazyk DSL, který by v případě ruční tvorby vyžadoval poměrně sofistikovaný rekurzivně sestupný analyzátor a na kterém bude skutečně patrné, že s růstem našich požadavků škáluje generický analyzátor mnohem lépe než ruční řešení.

Syntaktická analýza ve stylu jazyka Python pomocí nástroje PyParsing

Správná tvorba ručně psaných rekurzivně sestupných analyzátorů může být docela obtížná, a pokud potřebujeme vytvořit větší množství analyzátorů, může být jejich psaní a zvláště jejich údržba brzy únavná. Jedním z řešení je použít modul pro generickou syntaktickou analýzu, přičemž ti, kdo mají zkušenosti s gramatikami ve formě BNF nebo s unixovými nástroji lex a yacc budou přirozeně tíhnout k podobným nástrojům. V části za touto částí se budeme věnovat nástroji PLY (Python Lex Yacc), který je příkladem tohoto klasického přístupu. Avšak v této části se podíváme na naprosto odlišný druh nástroje pro syntaktickou analýzu: PyParsing.

Jeho autor popisuje nástroj PyParsing jako „alternativní přístup k tvorbě a provádění jednoduchých gramatik oproti tradičnímu přístupu na bázi nástrojů lex/yacc nebo oproti použití regulárních výrazů". (I když ve skutečnosti lze regulární výrazy použít také s nástrojem PyParsing.) U těch, kteří jsou zvyklí na tradiční přístup, vyžaduje nástroj PyParsing určitou změnu způsobu myšlení. Odměnou jim bude schopnost vyvíjet analyzátory, které nevyžadují množství kódu (což je dáno tím, že nástroj PyParsing nabízí řadu vysokoúrovňových prvků, které se mohou rovnat běžným konstruktům) a které jsou srozumitelné a snadno udržovatelné.

Nástroj PyParsing je dostupný pod licencí open source a lze jej použít v nekomerční i v komerční sféře. Nicméně nástroj PyParsing není součástí standardní knihovny Pythonu, takže je nutné jej stáhnout a nainstalovat zvlášť (i když pro linuxové uživatele je téměř jistě k dispozici prostřednictvím systému pro správu balíčků). Získat jej lze z webu *pyparsing.wikispaces.com* – stačí klepnout na odkaz Download. Pro Windows se dodává ve formě spustitelného instalačního programu a pro unixové systémy (např. Linux a Mac OS X) ve formě zdrojového kódu. Popis instalace je k dispozici na stránce s informacemi o stažení. Nástroj PyParsing je obsažen v jediném modulu (`pyparsing_py3.py`), takže jej lze snadno distribuovat s jakýmkoliv programem, který jej používá.

Stručné seznámení s nástrojem PyParsing

Nástroj PyParsing nečiní žádný rozdíl mezi lexikální a syntaktickou analýzou. Namísto toho poskytuje funkce a třídy pro tvorbu prvků analyzátoru – jeden prvek pro každou věc, s níž se má hledat shoda. Několik předdefinovaných prvků analyzátoru nabízí nástroj PyParsing, ostatní lze vytvořit voláním jeho funkcí nebo tvorbou instancí jeho tříd. Prvky analyzátoru lze též vytvořit zkombinováním jiných prvků dohromady. Můžeme tak například jejich spojením pomocí operátoru + vytvořit posloupnost nebo pomocí operátoru | vytvořit alternaci.

Koneckonců nástroj PyParsing je v podstatě jen kolekcí prvků analyzátoru (které samy mohou být tvořeny prvky analyzátoru a tak dále) složených dohromady.

Chceme-li analyzovaný text zpracovat, můžeme zpracovat výsledky vrácené nástrojem PyParsing nebo můžeme k určitým prvkům analyzátoru přidat akce analyzátoru (úryvky kódu) anebo provedeme kombinaci obou možností.

Nástroj PyParsing poskytuje pestrou paletu prvků analyzátoru, z nichž několik nejčastěji používaných si zde stručně popíšeme. Prvek analyzátoru `Literal()` se shoduje se zadaným textovým literálem a `CaselessLiteral()` dělá totéž, ignoruje však velikost písmen. Pokud nás některé části grama-

tiky nezajímají, můžeme použít prvek `Suppress()`, který se shoduje se zadaným textovým literálem (nebo prvkem analyzátoru), nepřidává jej ale do výsledků.

Prvek `Keyword()` je téměř stejný jako `Literal()`, ale musí být následován znakem, který není slovem. Tím se zamezí shodě v situaci, kdy je klíčové slovo prefixem něčeho jiného. Máme-li například text „filename“, pak se `Literal("file")` bude shodovat s `file`name a `name` zůstane pro další prvek analyzátoru, avšak ke shodě s prvkem `Keyword("file")` vůbec nedojde.

Dalším důležitým prvkem analyzátoru je `Word()`. Tento prvek přijímá řetězec, který považuje za skupinu znaků, a bude hledat shodu s libovolnou posloupností kterýchkoli ze zadaných znaků. Máme-li například text „abakus“, pak se `Word("abk")` bude shodovat s `abakus`. Pokud prvku `Word()` předáme dva řetězce, pak první obsahuje znaky, které jsou platné pro první znak shody, druhý obsahuje znaky, které jsou platné pro zbývající znaky. Tímto způsobem se typicky hledá shoda s identifikátory – například prvek `Word(alphas, alphanums)` se shoduje s textem, který začíná alfabetickým znakem, za nímž následuje žádný nebo více alfanumerických znaků. (Proměnné `alphas` a `alphanums` jsou předem definované řetězce znaků poskytované nástrojem PyParsing.)

Méně často používanou alternativou k prvku `Word()` je prvek `CharsNotIn()`, který přijímá řetězec, jejž považuje za skupinu znaků, přičemž hledá shodu se všemi znaky od aktuální pozice analýzy dále, dokud nenarazí na znak ze zadané skupiny znaků. Nepřeskakuje bílé místo a selže, je-li aktuálně analyzovaný znak v zadané množině, což znamená, není-li možno akumulovat žádný znak. Další dvě alternativy prvku `Word()` se též hojně používají. První je prvek `SkipTo()`, který je podobný prvku `CharsNotIn()`, ovšem až na to, že přeskakuje bílé místo a vždy uspěje, tedy i tehdy, když nic nenaakumuluje (prázdný řetězec). Druhým prvkem je `Regex()`, který se používá pro hledání shody pomocí zadaného regulárního výrazu.

Mezi další prvky nabízené nástrojem PyParsing patří také prvek `restOfLine`, který se shoduje s libovolnými znaky od aktuálního místa analýzy po konec řádku, prvek `pythonStyleComment`, který se shoduje s komentáři ve stylu jazyka Python nebo prvek `quotedString`, který se shoduje s řetězcem uzavřeným do jednoduchých či dvojitých uvozovek (počáteční a koncové uvozovky se musejí shodovat).

K dispozici je též řada pomocných funkcí obstarávajících běžné případy. Například funkce `delimitedList()` vrací prvek analyzátoru, který se shoduje se seznamem prvků se zadaným oddělovačem, a funkce `makeHTMLTags()` vrací dvojici prvků analyzátoru, které se shodují se začátkem a koncem zadané značky HTML, přičemž pro začátek se mohou dále shodovat s libovolnými atributy definovanými v této značce.

Prvky analyzátoru lze kvantifikovat podobně jako regulární výrazy, k čemuž slouží funkce `Optional()`, `ZeroOrMore()`, `OneOrMore()` a několik dalších. Není-li zadán žádný kvantifikátor, použije se výchozí množství 1. Prvky lze pomocí funkce `Group()` seskupovat a pomocí funkce `Combine()` kombinovat (jejich činnost si ukážeme v pozdější části textu).

Jakmile máme specifikované všechny prvky našeho analyzátoru včetně jejich kvantit, můžeme je začít kombinovat a vytvářet tak analyzátor. Prvky analyzátoru, které musejí následovat v posloupnosti za sebou, musíme stanovit vytvořením nového prvku analyzátoru, který spojuje dva nebo více stávajících prvků analyzátoru dohromady. Máme-li například prvky analyzátoru `key` a `value`, můžeme vytvořit prvek analyzátoru `key_value` zápisem `key_value = key + Suppress("=") + value`. Prvky analy-

zátoru, které se shodují s některou ze dvou či více alternativ, můžeme stanovit vytvořením nového prvku analyzátoru, který spojí operátorem | dva či více stávajících prvků analyzátoru dohromady. Máme-li například prvky analyzátoru `true` a `false`, můžeme vytvořit prvek analyzátoru `boolean` zapsáním `boolean = true | false`.

Všimněte si, že u prvku analyzátoru `key_value` jsme nepotřebovali upřesnit nic ohledně bílého místa kolem rovnítka. Nástroj PyParsing standardně přijímá libovolné množství bílého místa (včetně žádného) mezi prvky analyzátoru, takže kupříkladu definici formy BNF *KEY* '=' *VALUE* považuje za stejnou jako `\s*` *KEY* `\s* '=' \s*` *VALUE* `\s*`. (Toto výchozí chování lze samozřejmě vypnout.)

Je třeba poznamenat, že zde i v následujících oddílech budeme importovat každý prvek nástroje PyParsing samostatně. Například:

```
from pyparsing_py3 import (alphanums, alphas, CharsNotIn, Forward,
        Group, hexnums, OneOrMore, Optional, ParseException,
        ParseSyntaxException, Suppress, Word, ZeroOrMore)
```

Díky tomu nemusíme používat syntaxi `import * syntax`, která může znečistit náš obor názvů nechtěnými názvy, zároveň si ale nemusíme vypisovat třeba `pyparsing_py3.alphanums` a `pyparsing_py3.Word()`, ale můžeme pohodlně psát `alphanums` a `Word()`.

Před ukončením tohoto rychlého seznámení s nástrojem PyParsing a než vrhneme se na příklady následujícího oddílu, řekneme si několik důležitých myšlenek souvisejících se způsobem převodu formy BNF na analyzátor nástroje PyParsing.

Nástroj PyParsing obsahuje řadu předdefinovaných prvků, které se mohou shodovat s běžně používanými konstrukty. Vždy bychom měli pokud možno používat tyto prvky, čímž zajistíme nejlepší možný výkon. Dále přímý převod forem BNF na syntaxi nástroje PyParsing nepředstavuje vždy správný postup. Nástroj PyParsing pracuje s určitými konstrukty formy BNF vlastním způsobem, kterého bychom se měli vždy držet, chceme-li, aby náš analyzátor běžel efektivně. Zde si stručně rozebereme některé z předdefinovaných prvků a způsob, jak s nimi zacházet.

Běžnou definicí formy BNF jsou volitelné prvky. Například:

```
OPTIONAL_ITEM ::= ITEM | EMPTY
```

BNF

Pokud bychom tuto definici převedli přímo do nástroje PyParsing, napsali bychom:

```
optional_item = item | Empty() # ŠPATNĚ!
```

V tomto úryvku kódu předpokládáme, že `item` je nějaký prvek analyzátoru definovaný na jiném místě. Třída `Empy()` poskytuje prvek analyzátoru, který se může shodovat s ničím. To je sice správné z hlediska syntaxe, jde to však proti způsobu práce nástroje PyParsing. Správný způsob převodu do nástroje PyParsing je mnohem jednodušší a zahrnuje použití předdefinovaného prvku:

```
optional_item = Optional(item)
```

Některé definice formy BNF zahrnují definování prvku pomocí nich samotných. Například pro reprezentaci seznamu proměnných (třeba jako argumenty funkce) můžeme mít následující formu BNF:

BNF

```
VAR_LIST ::= VARIABLE | VARIABLE ',' VAR_LIST
VARIABLE ::= [a-zA-Z]\w*
```

Na první pohled nás to může svádět k přímému převodu do syntaxe nástroje PyParsing:

```
variable = Word(alphas, alphanums)
var_list = variable | variable + Suppress(",") + var_list # ŠPATNĚ!
```

Problém spočívá v syntaxi jazyka Python – nemůžeme totiž odkazovat na proměnnou var_list před jejím definováním. Nástroj PyParsing nabízí řešení: pomocí třídy `Forward()` vytvoříme „prázdný" prvek analyzátoru, k němuž později přidáme další prvky analyzátoru (včetně jeho samotného). Výše uvedenou formu BNF tedy můžeme zkusit převést takto:

```
var_list = Forward()
var_list << (variable | variable + Suppress(",") + var_list) # ŠPATNĚ!
```

Tato druhá verze je syntakticky správná, ale opět jde proti způsobu práce nástroje PyParsing, přičemž v rámci rozsáhlejšího analyzátoru by mohla vést k analyzátoru, který je velice pomalý nebo který prostě nefunguje. (Všimněte si, že musíme použít závorky, které zajistí, že se připojí celá pravá strana výrazu, a ne jen první část, protože operátor `<<` má vyšší precedenci než operátor `|`, což jinými slovy znamená, že váže silněji.) Přestože její použití zde není vhodné, dokáže být třída `Forward()` v jiných případech velice užitečná – použijeme ji hned v několika příkladech v následujících oddílech.

Místo třídy `Forward()` lze v takovýchto situacích použít jiné vzory programování, které se lépe hodí ke stylu práce nástroje PyParsing. Zde je nejjednodušší a nejpřesnější verze:

```
var_list = variable + ZeroOrMore(Suppress(",") + variable)
```

Tento vzor je ideální pro zpracování binárních operátorů:

```
plus_expression = operand + ZeroOrMore(Suppress("+") + operand)
```

Oba tyto druhy použití jsou tak časté, že nástroj PyParsing nabízí pomocné funkce, které poskytují vhodné prvky analyzátoru. Podíváme se na funkci `operatorPrecedence()`, kterou použijeme k vytvoření prvků analyzátoru pro unární, binární a ternární operátory v příkladu v posledním z následujících oddílů. Pro seznamy s oddělovačem se používá funkce `delimitedList()`, kterou si ukážeme nyní a kterou použijeme v příkladu v následujících oddílech:

```
var_list = delimitedList(variable)
```

Funkce `delimitedList()` přijímá prvek analyzátoru a volitelný oddělovač (oddělovač zde uvádět nemusíme, protože výchozím oddělovačem je čárka).

Dosud probíraná látka byla poměrně abstraktní. V následujících čtyřech oddílech vytvoříme čtyři analyzátory, z nichž každý bude sofistikovanější než ten předchozí. Na těchto analyzátorech si ukážeme, jak nejlépe používat nástroj PyParsing. První tři analyzátory jsou verze ručních analyzátorů pro nástroj PyParsing, které jsme vytvořili v předchozích částech. Čtvrtý analyzátor je nový a mnohem komplexnější. Ukážeme si jej v této části a potom ve verzi pro nástroj PLY.

Jednoduchá analýza dat ve tvaru klíč-hodnota

Ručně vytvořený analyzátor dat ve tvaru klíč-hodnota
➢ 497

V prvním oddílu předchozí části jsme ručně vytvořili analyzátor dat ve tvaru klíč-hodnota založený na regulárních výrazech, který jsme použili v programu `playlists.py` pro čtení souborů `.pls`. V tomto oddílu vytvoříme analyzátor, který bude provádět stejnou činnost, tentokrát ale pomocí nástroje PyParsing.

I nyní je stejně jako dříve účelem našeho analyzátoru naplnit slovník prvky ve tvaru klíč-hodnota odpovídajícími těm, které jsou v souboru, avšak převedenými na malá písmena. Úryvek ze souboru `.pls` je zachycen na obrázku 14.4 (strana 497) a příslušná forma BNF na obrázku 14.5 (strana 498). Nástroj PyParsing ve výchozím nastavení přeskakuje bílé místo, a proto můžeme neterminál *`BLANK`* a volitelné bílé místo ignorovat (`\s*`).

Analyzátor dat ve tvaru klíč-hodnota na bázi nástroje PLY
➢ 530

Celý kód si rozdělíme na tři části: vytvoření samotného analyzátoru, pomocnou funkci používanou analyzátorem a zavolání analyzátoru pro analýzu souboru `.pls`. Veškerý kód je umístěn v modulu `ReadKeyValue.py`, jejž importuje program `playlists.py`.

```
key_values = {}
left_bracket, right_bracket, equals = map(Suppress, "[]=")
ini_header = left_bracket + CharsNotIn("]") + right_bracket
key_value = Word(alphanums) + equals + restOfLine
key_value.setParseAction(accumulate)
comment = "#" + restOfLine
parser = OneOrMore(ini_header | key_value)
parser.ignore(comment)
```

U tohoto analyzátoru nečteme výsledky na konci, ale shromažďujeme je za běhu každý prvek *`klíč=hodnota`* ukládáme do slovníku `key_values`.

Funkcionální styl programování
➢ 381

Levá a pravá hranatá závorka a znaménko rovná se jsou důležitými prvky gramatiky, samy o sobě však nemají žádný význam. Pro každý z nich vytváříme prvek analyzátoru `Suppress()`, který se shoduje s příslušným znakem, ale nezahrne jej do výsledků. (Mohli bychom každý zapsat zvlášť, například `left_bracket = Suppress("[")` a tak dále, použití vestavěné funkce `map()` je však mnohem pohodlnější.)

Definice prvku analyzátoru `ini_header` je docela přirozeným způsobem odvozena z formy BNF: levá hranatá závorka, potom libovolné znaky kromě pravé hranaté závorky a poté pravá hranatá závorka. Pro tento prvek analyzátoru jsme žádnou akci analyzátoru nedefinovali, i kdyby tedy analyzátor nalezl nějaké shody, nic s nimi neudělá, což je přesně to, co chceme.

Prvek analyzátoru `key_value` je jediným prvkem, který nás skutečně zajímá. Shoduje se se „slovem“ (posloupnost alfanumerických znaků), za nímž následuje rovnítko a za ním zbytek řádku (který může být prázdný). Prvek analyzátoru `restOfLine` poskytuje nástroj PyParsing jako předdefinovaný prvek. Výsledky chceme shromažďovat za běhu, a proto do prvku analyzátoru `key_value` přidáme akci analyzátoru (odkaz na funkci), která se zavolá pro každou nalezenou shodu ve tvaru *`klíč=hodnota`*.

Přestože nástroj PyParsing poskytuje předdefinovaný prvek analyzátoru `pythonStyleComment`, my raději použijeme jednodušší prvek `Literal("#")` následovaný zbytkem řádku. (A díky chytrému

přetěžování operátorů v nástroji PyParsing jsme mohli napsat literál # jako řetězec, protože jej při jeho spojení s jiným prvkem analyzátoru nástroj PyParsing povýší na prvek `Literal()`.)

Samotný analyzátor je prvkem analyzátoru, který se shoduje s jedním či více prvky analyzátoru `ini_header` nebo `key_value` a který ignoruje prvky analyzátoru `comment`.

```
def accumulate(tokens):
    key, value = tokens
    key = key.lower() if lowercase_keys else key
    key_values[key] = value
```

Tato funkce se volá pro každou nalezenou shodu ve tvaru *klíč=hodnota*. Parametr `tokens` je n-tice nalezených prvků analyzátoru. V tomto případě bychom očekávali, že n-tice bude obsahovat klíč, rovnítko a hodnotu. Na rovnítko jsme ale použili prvek `Suppress()`, a proto obdržíme pouze klíč a hodnotu, což je přesně to, co chceme. Proměnnou `lowercase_keys`, která obsahuje logickou hodnotu, jsme vytvořili ve vnějším oboru platnosti a pro soubory `.pls` je nastavena na hodnotu `True`. (Je třeba poznamenat, že kvůli snazšímu výkladu jsme si tuto funkci ukázali až po vytvoření analyzátoru, ačkoliv ve skutečnosti musí být definována před jeho vytvořením, protože analyzátor se na ni odkazuje.)

```
try:
    parser.parseFile(file)
except ParseException as err:
    print("chyba při analýze: {0}".format(err))
    return {}
return key_values
```

Analyzátor máme připravený, a proto se můžeme pustit do zavolání metody `parseFile()`, která v tomto příkladu přijímá soubor `.pls` a pokusí se jej analyzovat. Pokud analýza selže, vypíšeme jednoduchou chybovou zprávu založenou na chybě obdržené z nástroje PyParsing. Na konci vrátíme slovník `key_values` (nebo prázdný slovník, pokud analýza selže), přičemž návratovou hodnotu metody `parseFile()` ignorujeme, poněvadž veškeré zpracování jsme umístili do akce analyzátoru.

Analyzování seznamu skladeb

Ručně vytvořený analyzátor formátu .m3u ➢ 500

Ve druhém oddílu předchozí části jsme ručně vytvořili analyzátor pro soubory `.m3u` založený na regulárních výrazech. V tomto oddílu vytvoříme analyzátor, který bude provádět stejnou činnost, tentokrát ale pomocí nástroje PyParsing. Úryvek ze souboru `.m3u` je k dispozici na obrázku 14.6 (strana 500) a příslušná forma BNF na obrázku 14.7 (strana 500).

Analyzátor formátu .m3u na bázi nástroje PLY ➢ 532

Stejně jako při probírání analyzátoru formátu `.pls` v předchozím oddílu si i zde rozebereme analyzátor formátu `.m3u` ve třech částech: vytvoření analyzátoru, potom pomocná funkce a nakonec volání analyzátoru. Návratovou hodnotu analyzátoru i nyní ignorujeme a naši datovou strukturu naplňujeme v průběhu analýzy. (V následujících dvou oddílech vytvoříme analyzátory, jejichž návratové hodnoty budeme dále používat.)

```
songs = []
title = restOfLine("title")
```

```
filename = restOfLine("filename")
seconds = Combine(Optional("-") + Word(nums)).setParseAction(
        lambda tokens: int(tokens[0]))("seconds")
info = Suppress("#EXTINF:") + seconds + Suppress(",") + title
entry = info + LineEnd() + filename + LineEnd()
entry.setParseAction(add_song)
parser = Suppress("#EXTM3U") + OneOrMore(entry)
```

Začínáme vytvořením prázdného seznamu, který bude uchovávat pojmenované n-tice `Song`.

Pojmenovaná n-tice Song ➤ 501

Přestože forma BNF je docela jednoduchá, některé prvky analyzátoru jsou ve srovnání s těmi, které jsme si dosud ukázali, o něco složitější. Všimněte si také, že prvky analyzátoru vytváříme oproti formě BNF v obráceném pořadí. To je dáno tím, že v Pythonu se můžeme odkazovat pouze na věci, které již existují. Například tedy nemůžeme vytvořit prvek analyzátoru pro neterminál *ENTRY* před vytvořením prvku pro neterminál *INFO*, poněvadž první se odkazuje na druhý.

Prvky analyzátoru `title` a `filename` se shodují se všemi znaky od aktuální pozice analyzátoru až po konec řádku. To znamená, že se mohou shodovat s libovolnými znaky včetně bílého místa, ne však se znakem nového řádku, na němž se zastaví. Těmto prvkům analyzátoru také přiřazujeme jména (např. „title“), díky čemuž k nim můžeme pohodlně přistupovat podle jména přes atributy objektu `tokens`, který je předáván funkcím provádějícím akce analyzátoru.

Prvek analyzátoru `seconds` se shoduje s volitelným znaménkem minus následovaným číslicemi (`nums` je předdefinovaný řetězec nástroje PyParsing, který obsahuje číslice). Prvek `Combine()` zajistí, že se znaménko (je-li uvedeno) a číslice vrátí jako jeden řetězec. (Pro prvek `Combine()` můžeme uvést oddělovač, který ale v tomto případě nepotřebujeme, protože výchozí hodnotou je prázdný řetězec, což nám naprosto vyhovuje.) Akce analyzátoru je tak jednoduchá, že jsme pro ni použili lambda funkci. Prvek `Combine()` zajistí, že v n-tici `tokens` bude vždy právě jeden token, který převedeme na celé číslo pomocí funkce `int()`. Pokud akce analyzátoru vrátí nějakou hodnotu, pak tato hodnota nahradí původně nalezený text a bude spojená k příslušným tokenem. Kromě toho jsme kvůli pozdějšímu pohodlnějšímu přístupu přiřadili tokenu název.

Prvek analyzátoru `info` se skládá z řetězcového literálu, který označuje začátek záznamu, dále z údaje o délce (prvek analyzátoru `seconds`), za nímž následuje titul (prvek analyzátoru `title`). To vše definujeme velice jednoduše a přirozeně takovým způsobem, který odpovídá příslušné definici ve formě BNF. Všimněte si také, že pro řetězcový literál a čárku používáme prvek `Suppress()`. Oba literály jsou sice pro gramatiku nezbytné, v souvislosti se samotnými daty nás ale vůbec nezajímají.

Definice prvku analyzátoru `entry` je velice snadná: prvek analyzátoru `info` následovaný novým řádkem a potom prvek analyzátoru `filename` následovaný novým řádkem (`LineEnd()` je předdefinovaným prvkem analyzátoru nástroje PyParsing, který se shoduje s novým řádkem). Náš seznam skladem nenaplňujeme daty na konci, ale v průběhu analýzy, a proto prvku analyzátoru `entry` přiřadíme akci analyzátoru, která se bude volat při každé shodě s neterminálem *ENTRY*.

Samotný analyzátor je prvek analyzátoru, který se shoduje s řetězcovým literálem označujícím soubor `.m3u`, za nímž následuje jeden nebo více neterminálů *ENTRY*.

```
def add_song(tokens):
    songs.append(Song(tokens.title, tokens.seconds,
                      tokens.filename))
```

Rozbalení mapování ➤ 177

Funkce `add_song()` je jednoduchá, což je dáno zvláště tím, že jsme pojmenovali prvky analyzátoru, které nás zajímají, a které jsou tudíž přístupné jako atributy objektu `tokens`. Tuto funkci bychom mohli samozřejmě napsat kompaktněji převedením tokenů na slovník a použitím rozbalení mapování (např. `songs.append(Song(**tokens.asDict()))`).

```
try:
    parser.parseFile(fh)
except ParseException as err:
    print("chyba při analýze: {0}".format(err))
    return []
return songs
```

Kód pro zavolání metody `ParserElement.parseFile()` je téměř shodný s kódem, který jsme použili pro analyzátor `.pls`, i když v tomto případě nepředáváme název souboru, ale soubor jsme otevřeli v textovém režimu a předáváme objekt `fh` typu `io.TextIOWrapper` obdržený z vestavěné funkce `open()`.

Dokončili jsme prohlídku dvou jednoduchých analyzátorů vytvořených pomocí nástroje PyParsing a ukázali jsme si většinu z běžně používaných prvků rozhraní API tohoto nástroje. V následujících dvou oddílech se podíváme na složitější analyzátory, z nichž oba jsou rekurzivní, což znamená, že obsahují neterminály, jejichž definice obsahuje je samotné. V posledním příkladu se podíváme také na to, jak pracovat s operátory a jejich precedencí a asociativitou.

Analýza bloků jakožto doménově specifického jazyka

Ručně vytvořený analyzátor bloků ➤ 502

Ve třetím oddílu předchozí části jsme vytvořili rekurzivně sestupný analyzátor pro soubory `.blk`. V tomto oddílu vytvoříme implementaci blokového analyzátoru v nástroji PyParsing, který by měl být snadněji pochopitelný a lépe udržovatelný.

Dva ukázkové soubory `.blk` jsou k dispozici na obrázcích 14.8 (strana 502) a 14.10 (strana 503). Forma BNF pro blokový formát je uvedena na obrázku 14.12 (strana 504).

Analyzátor bloků na bázi nástroje PLY ➤ 534

Tvorbu prvků analyzátoru si probereme ve dvou částech, potom se podíváme na pomocnou funkci a poté si ukážeme, jak lze náš analyzátor zavolat. A na konci si ukážeme, jak se výsledky analyzátoru transformují do kořenového bloku s podřízenými bloky (které samy mohou obsahovat podřízené bloky a tak pořád dál), což je námi požadovaný výstup.

```
left_bracket, right_bracket = map(Suppress, "[]")
new_rows = Word("/")("new_rows").setParseAction(
        lambda tokens: len(tokens.new_rows))
name = CharsNotIn("[]/\n")("name").setParseAction(
        lambda tokens: tokens.name.strip())
color = (Word("#", hexnums, exact=7) |
         Word(alphas, alphanums))("color")
```

```
empty_node = (left_bracket + right_bracket).setParseAction(
        lambda: EmptyBlock)
```

Jako u všech analyzátorů postavených na nástroji PyParsing i nyní vytváříme prvky analyzátoru podle příslušné formy BNF od poslední do první definice, takže pro každý vytvářený prvek analyzátoru jsou všechny prvky, na kterých závisí, již vytvořeny.

Hranaté závorky jsou podstatnou součástí formy BNF, pro nás ale nemají z hlediska výsledků žádný význam, a proto pro ně vytváříme vhodný analyzátor `Suppress()`.

U prvku analyzátoru `new_rows`nás to může svádět k použití prvku `Literal("/")`, který však hledá přesnou shodu s textem, kdežto my chceme shodu s tolika lomítky, kolik je jich přítomno. Při vytváření prvku analyzátoru `new_rows` jsme jeho výsledkům přiřadili název a přidali jsme akci analyzátoru, která nahradí řetězec s jedním či více lomítky číslem udávajícím počet těchto lomítek. Všimněte si také, že díky pojmenování můžeme v lambda funkci přistupovat k výsledku (tj. k nalezenému textu) pomocí názvu jakožto atributu objektu `tokens`.

Prvek analyzátoru `name` je malinko odlišný od příslušné specifikace ve formě BNF, poněvadž jsme se rozhodli, že kromě hranatých závorek a lomítek vyloučíme také znaky nového řádku. Výsledku opět přiřazujeme název. Kromě toho nastavujeme akci analyzátoru, tentokráte pro oříznutí bílého místa, protože bílé místo (kromě znaků nového řádku) může být součástí jména, na začátku ani na konci jej ale nechceme.

Pro prvek analyzátoru `color` jsme stanovili, že prvním znakem musí být # a potom přesně šest šestnáctkových číslic (celkem sedm znaků) nebo posloupnost alfanumerických znaků, z nichž první musí být alfabetický.

Rozhodli jsme se, že prázdné uzly zpracujeme zvláštním způsobem. Prázdný uzel definujeme jako levou hranatou závorku následovanou pravou hranatou závorkou, přičemž hranaté závorky nahrazujeme hodnotou `EmptyBlock`, která je na jiném místě definována jako `EmptyBlock = 0`. To znamená, že v seznamu s výsledky analyzátoru reprezentujeme prázdný blok nulou a (jak jsme si řekli již dříve) nové řádky reprezentujeme celým číslem udávajícím počet řádků (které bude vždy > 0).

```
nodes = Forward()
node_data = Optional(color + Suppress(":")) + Optional(name)
node_data.setParseAction(add_block)
node = left_bracket - node_data + nodes + right_bracket
nodes << Group(ZeroOrMore(Optional(new_rows) +
                          OneOrMore(node | empty_node)))
```

Uzly definujeme jako prvek analyzátoru `Forward()`, protože jej potřebujeme použít dříve, než určíme, s čím se má shodovat. Dále jsme přidali nový prvek analyzátoru `node_data`, který není obsažen ve formě BNF a který se shoduje s volitelným prvkem `color` a volitelným prvek `name`. Tomuto prvku analyzátoru přidělujeme akci analyzátoru, která vytvoří nový objekt typu `Block`, takže při každém výskytu prvku node_data se do seznamu výsledků analyzátoru přidá objekt typu `Block`.

Prvek analyzátoru `node` definujeme velice přirozeně přímo z formy BNF. Všimněte si, že prvky analyzátoru `node_data` a `nodes` jsou volitelné (první je tvořen dvěma volitelnými prvky, druhý je kvantifikován žádným či více výskyty), díky čemuž jsou správně povoleny i prázdné uzly.

Nakonec můžeme definovat prvek analyzátoru `nodes`. Původně byl vytvořen jako prvek `Forward()`, a proto k němu nyní musíme připojit prvky analyzátoru pomocí operátoru `<<`. Zde jsme prvek analyzátoru `nodes` nastavili na žádný či více výskytů volitelného nového řádku a jeden či více uzlů. Všimněte si, že prvek `node` jsme umístili před prvek `empty_node`. Nástroj PyParsing totiž hledá shodu zleva doprava, a proto je třeba umístit prvky analyzátoru se společnými prefixy od toho s nejdelší až po tem s nejkratší shodou.

Výsledky prvku analyzátoru `nodes` jsme seskupili pomocí prvku `Group()`, čímž zajistíme, že každý prvek `nodes` bude sám o sobě vytvořen jako seznam. To znamená, že prvek `node`, který obsahuje prvky `nodes`, bude pro prvek `node` reprezentován jako objekt typu `Block` a pro obsažené prvky `nodes` jako seznam, který zase může obsahovat objekty typu `Block` nebo celá čísla v případě prázdných bloků nebo nové řádky – a tak pořád dál. Právě kvůli této rekurzivní struktuře jsme museli prvek analyzátoru `nodes` vytvořit jako prvek `Forward()` a použít operátor `<<` pro přidání prvku `Group()` a v něm obsažených prvků do prvku `nodes`.

Je tu jedna podstatná a poněkud zapeklitá věc, na kterou je třeba upozornit. V definici prvku analyzátoru node jsme místo operátoru `-` použili operátor +. Stejně snadno bychom mohli použít operátor +, protože oba operátory + (`ParserElement.__add__()`) i - (`Parser- Element.__sub__()`) provádějí tutéž činnost – vracejí prvek analyzátoru, který reprezentuje spojení dvou prvků analyzátoru předaných jako operandy použitého operátoru.

Důvod pro volbu operátoru `-` tkví v jednom podstatném rozdílu mezi těmito operátory. Operátor `-` zastaví analýzu a vyvolá výjimku `ParseSyntaxException` ihned, jakmile narazí na nějakou chybu, což operátor + nedělá. Pokud bychom použili operátor +, pak by všechny chyby měly číslo řádku 1 a sloupec 1. Když ale použijeme operátor `-`, bude mít každá chyba správné číslo řádku i sloupce. Obecně lze operátor + považovat za správný přístup, pokud ale naše testy ukážou, že chyby obsahují nesprávné lokace, pak můžeme začít měnit operátory + na operátory `-`, jak jsme to učinili zde, přičemž v tomto případě byla nezbytná pouze jedna změna.

```
def add_block(tokens):
    return Block.Block(tokens.name, tokens.color if tokens.color
                                                 else "white")
```

Při každé analýze prvku `node_data` provedeme místo vrácení textu a jeho přidání do seznamu výsledků analyzátoru vytvoření objektu typu `Block`, který poté vrátíme. Kromě toho vždy nastavujeme barvu na bílou, není-li explicitně zadána.

V předchozích příkladech jsme analyzovali soubor a otevřený popisovač souboru (`io.TextIOWrapper`). Zde budeme analyzovat řetězec. Z hlediska nástroje PyParsing není žádný rozdíl v tom, zda mu předáme řetězec nebo soubor, stačí jen použít odpovídajícím způsobem metodu `ParserElement.parseFile()` nebo `ParserElement.parseString()`. Ve skutečnosti nástroj PyParsing nabízí i další metody pro analýzu, mezi něž patří metoda `ParserElement.scanString()`, která hledá v řetězci shody, a metoda `ParserElement.transformString()`, která vrací kopii zadaného řetězce, v němž jsou nalezené texty transformovány do nových textů obdržených jako návratové hodnoty z akcí analyzátoru.

```
    stack = [Block.get_root_block()]
    try:
        results = nodes.parseString(text, parseAll=True)
        assert len(results) == 1
        items = results.asList()[0]
        populate_children(items, stack)
    except (ParseException, ParseSyntaxException) as err:
        raise ValueError("Chyba {{0}}: syntaktická chyba, řádek "
                         "{0}".format(err.lineno))
    return stack[0]
```

Toto je první analyzátor nástroje PyParsing, u něhož jsme místo vytváření datové struktury během procesu analýzy použili jeho výsledky. Očekáváme, že výsledky obdržíme ve formě seznamu obsahujícího jediný objekt typu `ParseResults`. Ten převedeme na standardní seznam Pythonu, takže nyní máme seznam obsahující jediný prvek (seznam našich výsledků), který přiřadíme do proměnné `items` a poté jej zpracujeme ve funkci `populate_children()`.

Než si rozebereme zpracování výsledků, musíme stručně zmínit obsluhu chyb. Pokud analyzátor selže, vyvolá se výjimka. My ale nechceme, aby výjimky nástroje PyParsing procházely až ke klientům, protože později se můžeme rozhodnout tento generátor analyzátorů vyměnit. Pokud tedy dojde k nějaké výjimce, zachytíme ji a poté vyvoláme vlastní výjimku (`ValueError`) s příslušnými údaji.

V případě úspěšné analýzy ukázkového souboru `hierarchy.blk` bude seznam `items` vypadat takto (text ve tvaru `<Block.Block object at 0x8f52acd>` jsme pro lepší čitelnost nahradili textem `Block`):

Soubor `hierarchy.blk` ➢ 502

```
[0, Block, [], 2, 0, Block, [], 2, Block, [], 0, Block, []]
```

Kdykoliv analyzujeme prázdný blok, vrátíme do seznamu výsledků analyzátoru hodnotu 0. Kdykoliv analyzujeme nové řádky, vrátíme počet řádků. A kdykoliv narazíme na prvek node_data, vytvoříme objekt typu `Block`, který jej reprezentuje. Objekty typu `Block` mají vždy prázdný seznam potomků (tj. atribut `children` mají nastavený na []), protože v tomto okamžiku nevíme, zda daný blok bude mít nějaké potomky.

Vnější seznam zde tedy reprezentuje kořenový blok, nuly představují prázdné bloky, ostatní čísla (v tomto případě dvojky) představují nové řádky a `[]` jsou prázdné seznamy potomků, protože žádný blok souboru `hierarchy.blk` neobsahuje jiné bloky.

Seznam `items` pro ukázkový soubor `messagebox.blk` vypadá takto (s odsazením a řádkováním pro zdůraznění struktury a opět s textem `Block` pro lepší čitelnost):

Soubor `messagebox.blk` ➢ 503

```
[Block,
    [Block,
        [0, Block, [], 2, Block, [], 0, Block, [], 1, 0]
    ]
]
```

Zde můžeme vidět, že vnější seznam (představující kořenový blok) obsahuje blok, který má seznam potomků s jedním blokem, jenž obsahuje svůj vlastní seznam potomků, který je tvořen bloky (s jejich vlastními prázdnými seznamy potomků), novými řádky (2 a 1) a prázdnými bloky (nuly).

Problém s touto reprezentací seznamu výsledků spočívá v tom, že seznam potomků každého bloku je prázdný – všichni potomci bloku jsou v seznamu, který v seznamu výsledků analyzátoru *následuje* za daným blokem. Tuto strukturu tedy potřebujeme převést na jediný kořenový blok s podřízenými bloky. Až dosud jsme vytvářeli zásobník, tj. seznam obsahující jediný kořenový objekt typu `Block`. Proto nyní zavoláme funkci `populate_children()`, která přijímá seznam prvků vrácený analyzátorem a seznam s kořenovým blokem a těmito prvky naplní potomky kořenového bloku (a jejich potomky a tak pořád dál).

Funkce `populate_children()` je docela krátká, ale poněkud spletitá.

```
def populate_children(items, stack):
    for item in items:
        if isinstance(item, Block.Block):
            stack[-1].children.append(item)
        elif isinstance(item, list) and item:
            stack.append(stack[-1].children[-1])
            populate_children(item, stack)
            stack.pop()
        elif isinstance(item, int):
            if item == EmptyBlock:
                stack[-1].children.append(Block.get_empty_block())
            else:
                for x in range(item):
                    stack[-1].children.append(Block.get_new_row())
```

Procházíme každý prvek ve výsledném seznamu. Jedná-li se o objekt typu `Block`, přidáme jej k seznamu potomků posledního (na vrcholu umístěného) objektu typu `Block`. (Vzpomeňte si, že zásobník inicializujeme jediným kořenovým objektem typu `Block`.) Je-li prvkem neprázdný seznam, jedná se o seznam potomků, který náleží předchozímu bloku. Přidáme tedy tento předchozí blok (tj. posledního potomka objektu typu `Block` na vrcholu zásobníku) na zásobník, čímž z něj uděláme vrchol zásobníku, a poté rekurzivně zavoláme funkci `populate_children()` s tímto prvkem seznamem a zásobníkem. Tím zajistíme, že se tento prvek seznam (tj. jeho podřízené prvky) připojí k seznamu potomků správného prvku. Jakmile se rekurzivní volání dokončí, vyjmeme vrchol zásobníku, který je tak připraven pro další prvek.

Je-li prvkem celé číslo, pak se jedná buď o prázdný blok (0, tj. `EmptyBlock`), nebo o počet nových řádků. Jde-li o prázdný blok, připojíme jej k seznamu potomků objektu typu `Block` umístěnému na vrcholu zásobníku. Jde-li o nový počet řádků, připojíme toto číslo k seznamu potomků objektu typu `Block` umístěnému na vrcholu zásobníku

Je-li prvkem prázdný seznam, pak víme, že se jedná o prázdný seznam potomků, a proto neprovedeme žádnou akci, neboť všechny objekty typu `Block` jsou standardně inicializovány s prázdným seznamem potomků.

Na konci je prvek na vrcholu zásobníku stále kořenový objekt typu `Block`, který má však nyní potomky (kteří mají své vlastní potomky a tak pořád dál). Pro ukázkový soubor `hierarchy.blk` vytvoří funkce `populate_children()` strukturu znázorněnou na obrázku 14.13 (strana 505) a pro ukázkový soubor `messagebox.blk` vytvoří strukturu zachycenou na obrázku 14.14 (strana 505).

Převod do souboru SVG pomocí funkce `BlockOutput.save_blocks_as_svg()` je stejný pro všechny analyzátory bloků, poněvadž všechny produkují stejné struktury kořenového bloku a potomků.

Syntaktická analýza logiky prvního řádu

V tomto posledním oddílu věnovaném nástroji PyParsing vytvoříme analyzátor pro jazyk DSL vyjadřující formule v logice prvního řádu. Budeme pracovat s nejsložitější formou BNF ze všech příkladů v této lekci a v implementaci se musíme vypořádat s operátory včetně jejich precedencí a asociativit, což jsme zatím nikdy neřešili. Žádná ručně vytvořená verze tohoto analyzátoru neexistuje. Jakmile totiž dojdeme na tuto úroveň složitosti, pak je mnohem lepší sáhnout po generátoru analyzátorů. Ovšem kromě verze pro nástroj PyParsing, kterou si ukážeme v tomto oddílu, budeme mít možnost v posledním oddílu následující části porovnat ekvivalentní analyzátor pro nástroj PLY.

Analyzátor logiky prvního řádu na bázi nástroje PLY ➢ 536

Zde je několik příkladů formulí logiky prvního řádu, které chceme být schopni analyzovat:

Formulas

```
a = b
provšechna x: a = b
existuje y: a -> b
~ pravda | pravda & pravda -> provšechna x: existuje y: pravda
(provšechna x: existuje y: pravda) -> pravda & ~ pravda -> pravda
pravda & provšechna x: x = x
pravda & (provšechna x: x = x)
provšechna x: x = x & pravda
(provšechna x: x = x) & pravda
```

Místo řádných symbolů logických operátorů jsme se rozhodli použít obyčejné znaky, abychom se kromě samotného analyzátoru nenechali ničím jiným rozptylovat. Použili jsme tedy `provšechna` místo ∀, `existuje` místo ∃, `->` místo ⇒ (implikace), `|` místo ∨ (disjunkce), `&` místo ∧ (konjunkce) a `~` místo ¬ (negace). Řetězce jazyka Python jsou ve znakové sadě Unicode, a proto by bylo jednoduché použít skutečné symboly nebo upravit analyzátor tak, aby přijímal výše uvedené varianty i skutečné symboly.

U posledních dvou z výše uvedených formulí mají závorky svůj význam (tyto formule jsou odlišné), což ale nelze říci o dvou formulích nad nimi (formule začínající `pravda`), které jsou bez ohledu na závorky stejné. Analyzátor musí tyto podrobnosti samozřejmě správně zpracovat.

Překvapujícím aspektem logiky prvního řádu je, že negace (`~`) má nižší precedenci než rovnítko (`=`), takže `~ a = b` je ve skutečnosti `~ (a = b)`. Právě z tohoto důvodu logici umisťují za negaci mezeru.

Forma BNF pro náš jazyk DSL logiky prvního řádu je uvedena na obrázku 14.15. Pro lepší srozumitelnost neobsahuje žádné explicitní zmínky o bílém místu (žádné prvky `\n` nebo `\s*`), budeme ale předpokládat, že mezi všemi terminály i neterminály je povoleno bílé místo.

```
FORMULA   ::= ('provšechna' | 'existuje') SYMBOL ':'
FORMULA
            | FORMULA '->' FORMULA   # zprava asociativ-
ní
            | FORMULA '|' FORMULA    # zleva asociativní
            | FORMULA '&' FORMULA    # zleva asociativní
            | '~' FORMULA
            | '(' FORMULA ')'
            | TERM '=' TERM
            | 'pravda'
            | 'nepravda'
TERM      ::= SYMBOL | SYMBOL '(' TERM_LIST ')'
TERM_LIST ::= TERM | TERM ',' TERM_LIST
SYMBOL    ::= [a-zA-Z]\w*
```

Obrázek 14.15: Forma BNF pro logiku prvního řádu

Ačkoliv naše podmnožina syntaxe pro formy BNF nemá prostředky pro vyjádření precedence či asociativity, přidali jsme komentáře pro označení asociativity binárních operátorů. Precedence je uspořádána od nejnižší po nejvyšší v pořadí uvedeném ve formě BNF pro několik prvních alternativ. To znamená, že prvky `provšechna` a `existuje` mají nejnižší precedenci, za nimi následuje `->`, poté `|` a pak `&`. Všechny ostatní alternativy pak mají precedenci vyšší.

Než se pustíme do samotného analyzátoru, podíváme se na příkaz `import` a na řádek, který následuje za ním, poněvadž se od předchozího příkladu liší.

```
from pyparsing_py3 import (alphanums, alphas, delimitedList, Forward,
        Group, Keyword, Literal, opAssoc, operatorPrecedence,
        ParserElement, ParseException, ParseSyntaxException, Suppress,
        Word)
ParserElement.enablePackrat()
```

Memoizace ➢ 340

Příkaz import obsahuje několik nových věcí, které jsme dosud neviděli a kterým se budeme věnovat až v okamžiku, kdy na ně narazíme v analyzátoru. Voláním funkce `enablePackrat()` zapneme optimalizaci (na bázi memoizace), která může při analýze hlubokých hierarchií operátorů vést k značnému zrychlení.* Pokud se rozhodneme tuto funkci zavolat, pak je nejlepší tak učinit po importování modulu `pyparsing_py3` a před vytvořením jakýchkoli prvků analyzátoru.

Přestože je analyzátor kratší, rozebereme si jej pro snazší výklad ve třech krocích a poté si ukážeme, jak jej můžeme zavolat. Žádné akce analyzátoru zde nemáme, protože nám bohatě stačí strom AST (Abstract Syntax Tree – abstraktní syntaktický strom, což je v tomto případě seznam reprezentující výsledek analýzy, který můžeme v případě potřeby zpracovat později).

```
    left_parenthesis, right_parenthesis, colon = map(Suppress, "():")
    forall = Keyword("provšechna")
```

* Více informací o této technice syntaktické analýzy najdete v diplomové práci Bryana Forda na adrese *pdos.csail.mit.edu/~baford/packrat/*.

```
exists = Keyword("existuje")
implies = Literal("->")
or_ = Literal("|")
and_ = Literal("&")
not_ = Literal("~")
equals = Literal("=")
boolean = Keyword("nepravda") | Keyword("pravda")
symbol = Word(alphas, alphanums)
```

Všechny zde vytvořené prvky analyzátoru jsou přímočaré, i když jsme na konec některých jmen museli přidat podtržítko, aby nedošlo ke konfliktu s klíčovými slovy jazyka Python. Pokud bychom chtěli uživatelům dát možnost používat obyčejné znaky i odpovídající symboly znakové sady Unicode, mohli bychom část definice změnit například takto:

```
forall = Keyword("provšechna") | Literal("∀")
```

Máme-li editor bez podpory znakové sady Unicode, mohli bychom místo symbolů použít příslušné kódové body (například `Literal("\u2200")`).

```
term = Forward()
term << (Group(symbol + Group(left_parenthesis +
                delimitedList(term) + right_parenthesis)) | symbol)
```

Prvek `term` je definován pomocí sebe samého, a proto jej nejdříve vytvoříme jako prvek `Forward()`. A místo přímého převodu z formy BNF používáme jeden z programovacích vzorů nástroje PyParsing. Vzpomeňte si, že funkce `delimitedList()` vrací prvek analyzátoru, který se shoduje se seznamem jednoho nebo více výskytů zadaného prvku analyzátoru oddělených čárkami (nebo něčím jiným, pokud explicitně stanovíme oddělovač). Dle naší definice je tedy prvek analyzátoru `term` buď symbol, za nímž následuje seznam termů, nebo symbol. Obě varianty začínají stejným prvkem analyzátoru, a proto musíme jako první uvést tu s delší potenciální shodou.

```
formula = Forward()
forall_expression = Group(forall + symbol + colon + formula)
exists_expression = Group(exists + symbol + colon + formula)
operand = forall_expression | exists_expression | boolean | term
formula << operatorPrecedence(operand, [
                              (equals, 2, opAssoc.LEFT),
                              (not_, 1, opAssoc.RIGHT),
                              (and_, 2, opAssoc.LEFT),
                              (or_, 2, opAssoc.LEFT),
                              (implies, 2, opAssoc.RIGHT)])
```

Formule ve formě BNF sice může vypadat docela komplikovaně, v syntaxi nástroje PyParsing to ale není tak hrozné. Nejdříve definujeme prvek analyzátoru `formula` jako prvek `Forward()`, neboť je definován pomocí sebe samého. Definice prvků analyzátoru `forall_expression` a `exists_expression` je přímočará. Prvek `Group()` jsme použili proto, abychom z nich v rámci seznamu výsledků udělali podseznamy, které udržující své složky pohromadě a současně rozlišené jako samostatné jednotky.

Funkce `operatorPrecedence()` (která by se ve skutečnosti měla jmenovat spíše `createOperators()`) vytváří prvek analyzátoru, který se shoduje s jedním nebo více unárními, binárními a ternárními operátory. Před jejím zavoláním nejdříve stanovíme, jak vypadají naše operandy. V tomto případě jde o prvek `forall_expression` nebo `exists_expression` nebo `boolean` nebo `term`. Funkce `operatorPrecedence()` přijímá prvek analyzátoru, který se shoduje s platnými operandy, a poté seznam prvků analyzátoru, s nimiž se má pracovat jako s operátory, společně s jejich aritou (počtem operandů) a asociativitou. Výsledný prvek analyzátoru (v tomto případě `formula`) se bude shodovat s uvedenými operátory a jejich operandy.

Každý operátor je specifikován jako n-tice se třemi nebo čtyřmi prvky. Prvním prvkem je prvek analyzátoru představující daný operátor, druhým je jeho arita jako celé číslo (1 pro unární operátor, 2 pro binární operátor a 3 pro ternární operátor), třetím asociativita a čtvrtým volitelná akce analyzátoru.

Nástroj PyParsing odvodí pořadí operátorů z hlediska precedence z jejich relativních pozic v seznamu argumentů předávaných funkci `operatorPrecedence()`, přičemž první operátor má nejvyšší precedenci a poslední nejnižší, takže pořadí prvků v tomto seznamu argumentů je velice důležité. V tomto příkladu má operátor = nejvyšší precedenci (a žádnou asociativitu, takže bude zleva asociativní) a operátor `->` nejnižší precedenci a k tomu je zprava asociativní.

Tímto máme analyzátor hotový, takže se podíváme, jak jej můžeme zavolat.

```
    try:
        result = formula.parseString(text, parseAll=True)
        assert len(result) == 1
        return result[0].asList()
    except (ParseException, ParseSyntaxException) as err:
        print("Syntaktická chyba:\n{0.line}\n{1}^".format(err,
              " " * (err.column - 1)))
```

Tento kód je podobný tomu, který jsme použili v příkladu s bloky v předchozím oddílu. Zde se ovšem pokoušíme realizovat sofistikovanější obsluhu chyby. Pokud dojde k nějaké chybě, vypíše řádek, který obsahuje tuto chybu, a na dalším řádku mezery následované stříškou (^) označující pozici detekované chyby. Pokud například analyzujeme neplatnou formuli `provšechna x: = x & pravda`, obdržíme následující chybovou zprávu:

```
Syntaktická chyba:
provšechna x: = x & pravda
           ^
```

V tomto případě je lokace chyby vyznačena malinko vedle. Chyba spočívá v tom, že `= x` by mělo mít tvar `y = x`, i tak je to ale docela dobré.

V případě úspěšné analýzy obdržíme seznam objektů typu `ParseResults`, jež obsahují jediný výsledek, který stejně jako dříve převedeme na seznam Pythonu.

Dříve jsme si ukázali několik příkladů formulí. Nyní se podíváme na některé z nich i s výsledným seznamem, který vytvořil náš analyzátor a který si pěkně vypíšeme, aby tak lépe vyšla najevo jeho struktura.

Již dříve jsme si řekli, že operátor ~ má nižší precedenci než operátor =. Pojďme se tedy podívat, zda si náš analyzátor s touto skutečností správně poradí.

```
# ~pravda -> ~b = c
[
    ['~', 'pravda'],
    '->',
    ['~',
        ['b', '=', 'c']
    ]
]
```

```
# ~pravda -> ~(b = c)
[
    ['~', 'pravda'],
    '->',
    ['~',
        ['b', '=', 'c']
    ]
]
```

Zde obdržíme pro obě formule naprosto stejné výsledky, z nichž je patrné, že operátor = má vyšší precedenci než operátor ~. Samozřejmě bychom potřebovali napsat více testovacích formulí, abychom zkontrolovali všechny případy, vypadá to ale slibně.

Mezi dvě formule, které jsme si dříve ukázali, patří i formule `provšechna x: x = x & pravda` a `(provšechna x: x = x) & pravda`, u nichž jsme si řekli, že i přes to, že jediným rozdílem mezi nimi jsou závorky, jedná se o dvě zcela odlišné formule. Zde jsou seznamy, které pro ně vytvořil náš analyzátor:

```
# provšechna x: x = x & pravda
[
    'provšechna', 'x',
        [
            ['x', '=', 'x'],
            '&',
            'pravda'
        ]
]
```

```
# (provšechna x: x = x) & pravda
[
    [
        'provšechna', 'x',
            ['x', '=', 'x']
    ],
    '&',
    'pravda'
]
```

Je vidět, že analyzátor je schopen mezi těmito dvěma formulemi rozlišovat, a proto vytvoří zcela odlišné stromy (vnořené seznamy). Bez závorek patří do formule `provšechna` vše na pravé straně od dvojtečky, ale se závorkami je její rozsah omezen jen na tyto závorky.

Co ale s formulemi, které se liší také v tom, že jedna obsahuje závorky, které ale v tomto případě nehrají žádnou roli, takže obě formule jsou ve skutečnosti stejné? Těmito dvěma formulemi jsou `pravda & provšechna x: x = x` a `pravda & (provšechna x: x = x)` a při analýze naštěstí vedou k vytvoření naprosto stejného seznamu:

```
[
    'pravda',
    '&',
    [
        'provšechna', 'x',
            ['x', '=', 'x']
    ]
]
```

Na závorkách zde nezáleží, protože lze provést jen jedinou platnou analýzu.

Dokončili jsme analyzátor logiky prvního řádu na bázi nástroje PyParsing a v podstatě i všechny příklady v této knize věnované nástroji PyParsing. Pokud vás tento nástroj zaujal, podívejte se na jeho webovou stránku (*pyparsing.wikispaces.com*), která obsahuje spoustu dalších příkladů a rozsáhlou dokumentaci. Najdete zde též diskusní fórum a aktivní stránky Wiki.

V následující části se podíváme na stejné příklady, kterým jsme se věnovali v této části, tentokráte ale s použitím analyzátoru na bázi nástroje PLY, který funguje zcela jinak, než nástroj PyParsing.

Syntaktická analýza s nástrojem PLY podle nástrojů Lex a Yacc

Nástroj PLY (Python Lex Yacc) je čistou implementaci klasických unixových nástrojů lex a yacc v jazyku Python. Lex je nástroj, který vytváří lexikální analyzátory a yacc je nástroj, který vytváří syntaktické analyzátory, často s použitím lexikálního analyzátoru vytvořeného nástrojem lex. Nástroj PLY popisuje jeho autor David Beazley jako „přiměřeně efektivní a vhodný pro rozsáhlejší gramatiky. Poskytuje většinu standardních možností nástrojů lex a yacc včetně podpory pro prázdné produkce, precedenční pravidla, zotavení po chybě a podpory pro nejednoznačné gramatiky. Nástroj PLY se používá přímočarým způsobem a poskytuje *velice* rozsáhlou kontrolu chyb."

Nástroj PLY je dostupný pod licencí LPGL, a lze jej tedy použít ve většině situací. Podobně jako nástroj PyParsing ani nástroj PLY není součástí standardní knihovny Pythonu, a proto je nutné jej stáhnout a nainstalovat zvlášť – i když pro linuxové uživatele je téměř jistě k dispozici prostřednictvím systému pro správu balíčku. Od verze 3.0 fungují stejné moduly nástroje PLY s Pythonem 2 i 3.

Je-li zapotřebí získat a nainstalovat nástroj PLY ručně, jeho archiv tarball je dostupný na stránce *www.dabeaz.com/ply*. Na unixových systémech, jako je Linux a Mac OS X, lze archiv tarball rozbalit provedením příkazu `tar xvfz ply-3.2.tar.gz` v konzole. (Přesná verze nástroje PLY se může být samozřejmě jiná.) Uživatelé systému Windows mohou použít ukázkový program `untar.py`, který je součástí zdrojových kódu k této knize. Jsou-li příklady z této knihy umístěny v adresáři `C:\py3eg`, pak je nutné spustit na příkazovém řádku příkaz `C:\Python31\python.exe C:\py3eg\untar.py ply-3.2.tar.gz`.

Jakmile je archiv tarball rozbalen, změňte adresář na adresář nástroje PLY, který by měl obsahovat soubor s názvem `setup.py` a podadresář s názvem `ply`. Nástroj PLY lze nainstalovat automaticky nebo ručně. Pro automatickou instalaci spusťte v konzole příkaz `python setup.py install` nebo na příkazovém řádku Windows spusťte `C:\Python31\python.exe setup.py install`. Další možností je zkopírovat či přesunout adresář `ply` a jeho obsah do adresáře s webovými balíčky Pythonu (nebo do svého lokálního adresáře s webovými balíčky). Po nainstalování jsou moduly nástroje PLY dostupné jako `ply.lex` a `ply.yacc`.

Nástroj PLY činí jasný rozdíl mezi lexikální analýzou (tokenizací) a syntaktickou analýzou. Lexikální analyzátor nástroje PLY je ve skutečnosti tak výkonný, že nám bohatě postačí pro zpracování všech příkladů v této lekci kromě analyzátoru logiky prvního řádu, pro kterou použijeme oba moduly `ply.lex` a `ply.yacc`.

Když jsme probírali nástroj PyParsing, rozebrali jsme si nejdříve různé principy vztahující se k tomuto nástroji, zvláště pak způsob, jakým se do jeho syntaxe převádějí určité konstrukty formy

BNF. To ale v případě nástroje PLY není vůbec nutné, protože je navržen pro přímou práci s regulárními výrazy a formami BNF, takže místo nějakého koncepčního rámce si shrneme několik klíčových konvencí nástroje PLY a poté se pustíme přímo do příkladů a veškeré detaily si vysvětlíme při jejich výkladu.

Nástroj PLY značným způsobem využívá konvence pro pojmenování a introspekci, což je důležité mít na paměti při tvorbě lexikálních a syntaktických analyzátorů pomocí tohoto nástroje.

Každý lexikální a syntaktický analyzátor závisí na proměnné s názvem `tokens`. Tato proměnná musí uchovávat n-tici nebo seznam názvů tokenů, které jsou obvykle tvořeny velkými písmeny, což odpovídá neterminálům ve formě BNF. Každý token musí mít odpovídající proměnnou či funkci, jejíž název má tvar `t_`*`NÁZEV_TOKENU`*. Je-li definována proměnná, musí být nastavena na řetězec obsahující regulární výraz. K tomu účelu se tedy běžně používá holý řetězec. Je-li definována funkce, musí mít dokumentační řetězec, jenž obsahuje regulární výraz, opět obvykle ve formě holého řetězce. V každém případě tento regulární výraz stanoví vzor, který se shoduje s odpovídajícím tokenem.

Jedním jménem, které má pro nástroj PLY speciální význam, je `t_error()`. Dojde-li k chybě při lexikální analýze a je-li definována funkce s tímto názvem, pak se tato funkce zavolá.

Pokud chceme, aby se lexikální analyzátor shodoval s tokenem, který ale zároveň nemá být součástí výsledků (např. komentář v programovacím jazyku), můžeme použít jeden ze dvou způsobů. Pokud používáme proměnnou, pak jí dáme název ve tvaru `t_ignore_`*`NÁZEV_TOKENU`*. Pokud používáme funkci, pak použijeme běžný název `t_`*`NÁZEV_TOKENU`*, ovšem s tím, že zajistíme, aby tato funkce vracela hodnotu `None`.

Syntaktický analyzátor nástroje PLY se drží stejných konvencí jako lexikální analyzátor, takže pro každé pravidlo formy BNF vytvoříme funkci s prefixem `p_`, jejíž dokumentační řetězec obsahuje pravidlo formy BNF, s nímž se má hledat shoda (ovšem s tím, že místo `::=` použijeme `:`). Při každé nalezené shodě se pak zavolá odpovídající funkce s parametrem (podle příkladů v dokumentaci jej budeme označovat jako `p`). Tento parametr lze indexovat s tím, že `p[0]` odpovídá neterminálu, který dané pravidlo definuje, a `p[1]` a další indexy odpovídají částem na pravé straně formy BNF.

Precedenci a asociativitu lze nastavit vytvořením proměnné s názvem precedence, které přiřadíme n-tici n-tic (v pořadí daném požadovanou precedencí), které určují asociativity tokenů.

Podobně jako u lexikálního analyzátoru se i v případě výskytu syntaktické chyby zavolá funkce `p_error()`, je-li definována.

Při prohlídce příkladů budeme používat všechny výše popsané konvence a ještě několik dalších.

Abychom se vyhnuli duplikování informací z předchozích částí této lekce, zaměříme se v příkladech a jejich výkladu pouze na analýzu pomocí nástroje PLY. Předpokládáme, že již znáte analyzované formáty a kontexty, ve kterých se používají – že jste buď četli alespoň druhou část této lekce a poslední oddíl třetí části nebo že se v případě potřeby vrátíte zpět pomocí uváděných zpětných odkazů.

Analyzování jednoduchých dat ve tvaru klíč-hodnota

Ručně vytvořený analyzátor dat ve tvaru klíč-hodnota ➤ 497

Lexikální analyzátor nástroje PLY je dostatečný pro práci s daty ve tvaru klíč-hodnota uchovávanými v souborech `.pls`. Každý lexikální (i syntaktický) analyzátor nástroje PLY obsahuje seznam tokenů, které *musejí* být uloženy v proměnné `tokens`. Nástroj PLY hojně využívá introspekci, takže názvy proměnných a funkcí, a dokonce i obsah dokumentačních řetězců musí dodržovat jeho konvence. Zde jsou tokeny a jejich regulární výrazy a funkce pro analyzátor souborů `.pls` na bázi nástroje PLY:

Analyzátor dat ve tvaru klíč-hodnota na bázi nástroje PyParsing ➤ 515

```
tokens = ("INI_HEADER", "COMMENT", "KEY", "VALUE")

t_ignore_INI_HEADER = r"\[[^]]+\]"
t_ignore_COMMENT = r"\#.*"

def t_KEY(t):
    r"\w+"
    if lowercase_keys:
        t.value = t.value.lower()
    return t

def t_VALUE(t):
    r"=.*"
    t.value = t.value[1:].strip()
    return t
```

Forma BNF pro soubory ve formátu `.pls` ➤ 498

Tokeny *`INI_HEADER`* a *`COMMENT`* používají pro hledání shody obyčejné regulární výrazy, a protože oba používají prefix `t_ignore_`, budou po nalezení náležité shody zahozeny. Alternativní přístup k ignorování shod spočívá v definování funkce, která používá prefix t_ (např. `t_COMMENT()`) a jejíž sadu tvoří příkaz `pass` (nebo `return None`), protože je-li návratová hodnota `None`, daný token se zahodí.

Pro tokeny *`KEY`* a *`VALUE`* jsme nepoužili regulární výrazy, ale funkce. Regulární výraz pro hledání shody musíme tedy uvést v dokumentačním řetězci funkce. Zde používáme pro dokumentační řetězce holé řetězce, poněvadž takto jsme zvyklí zapisovat regulárními výrazy, ve kterých pak nemusíme zdvojovat zpětná lomítka. Při použití funkce se daný token předá jako objekt `t` (podle konvence pro pojmenování v příkladech nástroje PLY) typu `ply.lex.LexToken`. Nalezený text je uchováván v atributu `ply.lex.LexToken.value`, který můžeme v případě potřeby změnit. Z funkce musíme vždy vrátit objekt `t`, má-li se tento token zahrnout do výsledků.

Ve funkci `t_KEY()` převádíme nalezený text na malá písmena, má-li proměnná `lowercase_keys` (z vnějšího oboru platnosti) hodnotu `True`. A ve funkci t_VALUE() ořezáváme rovnítko a případné počáteční i koncové bílé místo.

Kromě našich vlastních tokenů je vhodné definovat několik funkcí určených pro hlášení chyb.

```
def t_newline(t):
    r"\n+"
    t.lexer.lineno += len(t.value)

def t_error(t):
```

```
    line = t.value.lstrip()
    i = line.find("\n")
    line = line if i == -1 else line[:i]
    print("Chyba při analýze řádku {0}: {1}".format(t.lineno + 1,
                                                    line))
```

Atribut tokenu `lexer` (typu `ply.lex.Lexer`) poskytuje přístup k samotnému lexikálnímu analyzátoru. Zde jsme aktualizovali atribut `lineno` lexikálního analyzátoru o počet nalezených nových řádků.

Všimněte si, že vůbec nemusíme řešit prázdné řádky, protože se o ně postará funkce pro hledání shody `t_newline()`.

Pokud dojde k nějaké chybě, zavolá se funkce `t_error()`. Vypíšeme chybovou zprávu a nejvýše jeden řádek vstupu. K číslu řádku přičítáme jedničku, protože atribut `lexer.lineno` počítá řádky od nuly.

Definice všech tokenů máme připraveny, můžeme se tedy pustit do lexikální analýzy nějakých dat a vytvoření odpovídajícího slovníku s prvky ve tvaru klíč-hodnota.

```
key_values = {}
lexer = ply.lex.lex()
lexer.input(file.read())
key = None
for token in lexer:
    if token.type == "KEY":
        key = token.value
    elif token.type == "VALUE":
        if key is None:
            print("Chyba při analýze: hodnota '{0}' bez klíče"
                  .format(token.value))
        else:
            key_values[key] = token.value
            key = None
```

Lexikální analyzátor čte celý vstupní text a lze jej použít jako iterátor, který v každé iteraci vytvoří jeden token. Atribut `token.type` uchovává název aktuálního tokenu (jedná se o jeden z názvů ze seznamu `tokens`) a atribut `token.value` uchovává text nalezené shody (nebo cokoliv, čím jsme jej nahradili).

Postupně procházíme jednotlivé tokeny. Jedná-li se o token *`KEY`*, uložíme jej a počkáme na jeho hodnotu. Jde-li o token *`VALUE`*, přidáme jej s aktuálním klíčem do slovníku `key_values`. Na konci (který zde není uveden) vrátíme slovník volajícímu stejně jako v případě analyzátorů pro formát `.pls` (na bázi regulárních výrazů i nástroje PyParsing).

Analyzování seznamu skladeb

Ručně vytvořený analyzátor formátu .m3u
➤ 500

V tomto oddílu vyvineme analyzátor na bázi nástroje PLY pro formát `.m3u`. Tento analyzátor bude stejně jako v předchozích implementacích vracet své výsledky ve formě seznamu objektů typu `Song` (`collections.namedtuple()`), které uchovávají titul, délku v sekundách a název souboru.

Analyzátor formátu .m3u na bázi nástroje PyParsing
➤ 516

Tento formát je velice jednoduchý, a proto si pro celou analýzu bohatě vystačíme s lexikálním analyzátorem nástroje PLY. Stejně jako dříve vytvoříme i nyní seznam `tokens`, v němž každý prvek odpovídá neterminálu ve formě BNF:

```
tokens = ("M3U", "INFO", "SECONDS", "TITLE", "FILENAME")
```

Forma BNF pro soubory ve formátu .m3u
➤ 500

Není zde token *`ENTRY`*, který představuje neterminál tvořený neterminály *`SECONDS`* a *`TITLE`*. Místo něj definujeme dva stavy s názvy `entry` a `filename`. Když se lexikální analyzátor nachází ve stavu `entry`, pokusíme se přečíst vteřiny (neterminál *`SECONDS`*) a titul (neterminál *`TITLE`*), což je vlastně neterminál *`ENTRY`*. A je-li lexikální analyzátor ve stavu `filename`, pokusíme se přečíst název souboru (neterminál *`FILENAME`*). Aby nástroj PLY rozuměl stavům, musíme vytvořit proměnnou `state`, kterou nastavíme na seznam jedné nebo více n-tic se dvěma prvky. Prvním prvkem každé n-tice je název stavu a druhým prvkem typ stavu, který může mít hodnotu `inclusive` (tzn. tento stav je doplňkem aktuálního stavu) nebo `exclusive` (tzn. tento stav je jediným aktivním stavem). Nástroj PLY definuje stav `INITIAL`, v němž zahajují svou činnost všechny lexikální analyzátory. Zde je definice proměnné `states` pro analyzátor pro formát `.m3u` na bázi nástroje PLY:

```
states = (("entry", "exclusive"), ("filename", "exclusive"))
```

Tokeny a stavy máme definované, takže se můžeme pustit do definování regulárních výrazů a funkcí odpovídajících formě BNF.

```
    t_M3U = r"\#EXTM3U"

    def t_INFO(t):
        r"\#EXTINF:"
        t.lexer.begin("entry")
        return None

    def t_entry_SECONDS(t):
        r"-?\d+,"
        t.value = int(t.value[:-1])
        return t

    def t_entry_TITLE(t):
        r"[^\n]+"
        t.lexer.begin("filename")
        return t

    def t_filename_FILENAME(t):
        r"[^\n]+"
```

```
        t.lexer.begin("INITIAL")
        return t
```

Tokeny, regulární výrazy a funkce pracují standardně ve stavu `INITIAL`. Nicméně vložením názvu stavu za prefix `t_` můžeme stanovit, aby byly aktivní pouze v určeném stavu. V tomto případě se bude regulární výraz `t_M3U` a funkce `t_INFO()` shodovat pouze ve stavu INITIAL, funkce `t_entry_SECONDS()` a `t_entry_TITLE()` pouze ve stavu `entry` a funkce `t_filename_FILENAME()` pouze ve stavu `filename`.

Stav lexikálního analyzátoru změníme zavoláním metody `begin()` na objektu, který představuje lexikální analyzátor, a předáme jí název nového stavu jako argument. Takže v tomto příkladu přejdeme při shodě s tokenem *`INFO`* do stavu `entry`, v němž lze hledat shody pouze s tokeny *`SECONDS`* a *`TITLE`*. Při shodě s tokenem *`TITLE`* přejdeme do stavu `filename` a při shodě s tokenem *`FILENAME`* přejdeme zpět do stavu `INITIAL`, v němž jsme připraveni na další token *`INFO`*.

Všimněte si, že ve funkci `t_INFO()` vracíme hodnotu `None`, což znamená, že tento token bude zahozen. To je v pořádku, protože pro každý záznam sice musí existovat shoda s textem `#EXTINF:`, který však pro další zpracování vůbec nepotřebujeme. Ve funkci `t_entry_SECONDS()` ořízneme koncovou čárku a nahradíme hodnotu tokenu počtem vteřin ve formě celého čísla.

V tomto analyzátoru chceme ignorovat falešné bílé místo, které se může objevit mezi tokeny, a to bez ohledu na stav, v němž se lexikální analyzátor nachází. Toho můžeme docílit vytvořením proměnné `t_ignore`, které přiřadíme stav `ANY`, což znamená, že bude aktivní v libovolném stavu:

```
t_ANY_ignore = " \t\n"
```

Tímto způsobem zajistíme, že se případné bílé místo mezi tokeny bezpečně a pohodlně ignoruje.

Dále jsme definovali dvě funkce `t_ANY_newline()` a `t_ANY_error()`, jež mají naprosto stejná těla jako funkce `t_newline()` a `t_error()` definované v předchozím oddílu (strana 530), a proto si je zde ukazovat nebudeme. Tyto funkce se svými názvy hlásí ke stavu ANY, a tudíž budou aktivní bez ohledu na to, v jakém stavu se lexikální analyzátor nachází.

```
songs = []
title = seconds = None
lexer = ply.lex.lex()
lexer.input(fh.read())
for token in lexer:
    if token.type == "SECONDS":
        seconds = token.value
    elif token.type == "TITLE":
        title = token.value
    elif token.type == "FILENAME":
        if title is not None and seconds is not None:
            songs.append(Song(title, seconds, token.value))
            title = seconds = None
        else:
            print("Chyba, soubor '{0}' bez titulu/délky"
                  .format(token.value))
```

Lexikální analyzátor používáme stejným způsobem jako v případě lexikálního analyzátoru pro formát `.pls`. Procházíme tedy tokeny, shromažďujeme hodnoty (pro vteřiny a tituly), a kdykoliv obdržíme název souboru, přidáme jej společně s příslušnou délkou a titulem do seznamu skladeb jako novou skladbu. Na konci (který jsme si neukázali) vrátíme volajícímu stejně jako předtím slovník `key_values`.

Analýza bloků jakožto doménově specifického jazyka

Ručně vytvořený analyzátor bloků ➢ 502

Blokový formát je oproti formátu `.pls` na bázi dat ve tvaru klíč-hodnota nebo formátu .m3u o něco sofistikovanější, neboť umožňuje vzájemné vnořování bloků do jiných bloků. To ale není pro nástroj PLY žádný problém a ve skutečnosti lze definice tokenů provést jen pomocí regulárních výrazů, aniž bychom museli sáhnout po funkcích či stavech.

Analyzátor bloků na bázi nástroje PyParsing ➢ 518

```
tokens = ("NODE_START", "NODE_END", "COLOR", "NAME", "NEW_ROWS",
          "EMPTY_NODE")

t_NODE_START = r"\["
t_NODE_END = r"\]"
t_COLOR = r"(?:\#[\dA-Fa-f]{6}|[a-zA-Z]\w*):"
t_NAME = r"[^][/\n]+"
t_NEW_ROWS = r"/+"
t_EMPTY_NODE = r"\[\]"
```

Forma BNF pro soubory ve formátu `.blk` ➢ 504

Regulární výrazy jsme převzali přímo z formy BNF s tím, že jsme se rozhodli zakázat nové řádky v názvech. Kromě toho jsme definovali regulární výraz `t_ignore` tak, aby přeskakoval mezery a tabulátory, a dále funkce `t_newline()` a `t_error()`, které jsou stejné jako dříve, tedy až na to, že funkce `t_error()` místo vypsání chybové zprávy vyvolá vlastní výjimku `LexError`, které chybovou zprávu předá.

S připravenými tokeny se můžeme pustit do lexikální analýzy, kterou si poté vyzkoušíme.

```
stack = [Block.get_root_block()]
block = None
brackets = 0
lexer = ply.lex.lex()
try:
    lexer.input(text)
    for token in lexer:
```

Stejně jako u předchozích analyzátorů bloků začneme vytvořením zásobníku (seznamu) s prázdným kořenovým blokem (objektem typu `Block`). Ten bude naplněn podřízenými bloky (a podřízené bloky dalšími podřízenými bloky a tak pořád dál) podle analyzovaných bloků. Na konci vrátíme kořenový blok se všemi jeho potomky. Proměnná `block` se používá k uchovávání odkazu na blok, který se aktuálně analyzuje, a který je tedy možné v průběhu analýzy aktualizovat. Kvůli lepšímu hlášení chyb dále uchováváme počet hranatých závorek.

Rozdílem oproti předchozím verzím je to, že lexikální a syntaktickou analýzu tokenů provádíme uvnitř sady `try ... except`. Díky tomu totiž můžeme zachytit jakékoli výjimky `LexError` a převést je na výjimky `ValueError`.

```
if token.type == "NODE_START":
    brackets += 1
    block = Block.get_empty_block()
    stack[-1].children.append(block)
    stack.append(block)
elif token.type == "NODE_END":
    brackets -= 1
    if brackets < 0:
        raise LexError("příliš mnoho ']'s")
    block = None
    stack.pop()
```

Kdykoliv zahájíme analýzu nového uzlu, zvýšíme počet hranatých závorek a vytvoříme nový prázdný blok. Ten je přidán jako poslední potomek do seznamu potomků bloku na vrcholu zásobníku a sám uložen na zásobník. Má-li blok nějakou barvu či název, můžeme je nastavit, protože v proměnné `block` udržujeme odkaz na aktuální blok.

Zde použitá logika se trošku liší od logiky použité v rekurzivně sestupném analyzátoru, u něhož jsme ukládali nové bloky na zásobník pouze tehdy, když jsme věděli, že mají vnořené bloky. Zde vždy ukládáme nové bloky na zásobník, což je v pořádku, protože budou ihned odstraněny, neobsahují-li žádné vnořené bloky. Kód je tak mnohem jednodušší a regulérnější.

Když dorazíme na konec bloku, snížíme počet hranatých závorek. Je-li toto číslo záporné, pak víme, že jsme prošli příliš mnoha uzavíracími hranatými závorkami, a můžeme tedy okamžitě nahlásit chybu. V opačném případě nastavíme proměnnou `block` na hodnotu `None`, protože nyní nemáme žádný aktuální blok, a odebereme blok z vrcholu zásobníku (který by v tomto stavu neměl být nikdy prázdný).

```
elif token.type == "COLOR":
    if block is None or Block.is_new_row(block):
        raise LexError("syntaktická chyba")
    block.color = token.value[:-1]
elif token.type == "NAME":
    if block is None or Block.is_new_row(block):
        raise LexError("syntaktická chyba")
    block.name = token.value
```

Pokud obdržíme barvu nebo název, nastavíme odpovídající atribut objektu `block` (aktuálního bloku), který by neměl obsahovat hodnotu `None` ani označovat nový řádek, ale měl by odkazovat na objekt typu `Block`.

```
elif token.type == "EMPTY_NODE":
    stack[-1].children.append(Block.get_empty_block())
elif token.type == "NEW_ROWS":
    for x in range(len(token.value)):
        stack[-1].children.append(Block.get_new_row())
```

Pokud obdržíme prázdný uzel nebo jeden či více nových řádků, přidáme je jako poslední potomky do seznamu potomků bloku na vrcholu zásobníku.

```
        if brackets:
            raise LexError("nevyvážené závorky []")
    except LexError as err:
        raise ValueError("Chyba {{0}}:řádek {0}: {1}".format(
                         token.lineno + 1, err))
```

Po dokončení lexikální analýzy zkontrolujeme, zda jsou hranaté závorky vyvážené, a pokud ne, tak vyvoláme výjimku `LexError`. Pokud k výjimce `LexError` došlo během lexikální analýzy, syntaktické analýzy nebo při kontrole hranatých závorek, pak vyvoláme výjimku `ValueError`, která obsahuje název pole pro metodu `str.format()` – očekáváme, že jej volající použije pro vložení názvu souboru, což na tomto místě provést nemůžeme, protože máme k dispozici pouze text souboru, a ne jeho název ani příslušný objekt.

Na konci (který si zde neukážeme) vrátíme objekt `stack[0]` typu `Block`, který by měl nyní mít potomky (a kteří zase mohou mít své potomky) reprezentující analyzovaný soubor `.blk`. Tento blok je vhodný pro předání funkci `BlockOutput.save_blocks_as_svg()`, stejně jako v případě rekurzivně sestupného analyzátoru bloků a analyzátoru bloků na bázi nástroje PyParsing.

Syntaktická analýza logiky prvního řádu

Analyzátor logiky prvního řádu na bázi nástroje PyParsing ➢ 523

V posledním oddílu v části věnované nástroji PyParsing jsme vytvořili analyzátor pro logiku prvního řádu. V tomto oddílu vytvoříme jeho verzi pro nástroj PLY, která bude navržena tak, aby generoval naprosto stejný výstup.

Forma BNF pro logiku prvního řádu ➢ 524

Činnosti související s přípravou lexikálního analyzátoru jsou velmi podobné těm, které jsme prováděli již dříve. Jediným novým aspektem je, že udržujeme slovník „klíčových slov“, který používáme ke kontrole při každé nalezené shodě s neterminálem SYMBOL (jedná se o ekvivalent identifikátoru v programovacím jazyku). Zde je kompletní kód lexikálního analyzátoru kromě regulárního výrazu `t_ignore` a funkcí `t_newline()` a `t_error()`, které jsou úplně stejné jako jejich protějšky v předchozím oddílu.

```
keywords = {"existuje": "EXISTS", "provšechna": "FORALL",
            "pravda": "TRUE", "nepravda": "FALSE"}
tokens = (["SYMBOL", "COLON", "COMMA", "LPAREN", "RPAREN",
           "EQUALS", "NOT", "AND", "OR", "IMPLIES"] +
          list(keywords.values()))

def t_SYMBOL(t):
    r"[a-zA-Z]\w*"
    t.type = keywords.get(t.value, "SYMBOL")
    return t

t_EQUALS = r"="
t_NOT = r"~"
```

```
t_AND = r"&"
t_OR = r"\|"
t_IMPLIES = r"->"
t_COLON = r":"
t_COMMA = r","
t_LPAREN = r"\("
t_RPAREN = r"\)"
```

Funkce `t_SYMBOL()` se používá pro hledání shody se symboly (identifikátory) i s klíčovými slovy. Pokud klíč předaný metodě `dict.get()` není ve slovníku, vrátí se výchozí hodnota (v tomto případě `"SYMBOL"`). V opačném případě se vrátí název tokenu odpovídající zadanému klíči. Všimněte si, že na rozdíl od předchozího lexikálního analyzátoru zde neměníme hodnotu atributu `value` objektu typu `ply.lex.LexToken`, měníme však jeho atribut `type`, který tak bude buď `"SYMBOL"`, nebo název tokenu příslušného klíčového slova. Pro všechny ostatní tokeny hledáme shody pomocí jednoduchých regulárních výrazů, které se shodují s jedním či dvěma znakovými literály.

Typ `dict` ➤ 128

Ve všech předchozích příkladech postavených na nástroji PLY jsme si vystačili se samotným lexikálním analyzátorem. Avšak v případě formy BNF pro logiku prvního řádu potřebujeme kromě lexikálního také syntaktický analyzátor nástroje PLY. Příprava syntaktického analyzátoru na bázi nástroje PLY je docela přímočará – a na rozdíl od verze pro nástroj PyParsing nemusíme přetvářet naši formu BNF tak, aby odpovídala určitým vzorům, ale můžeme ji použít přímo.

Pro každou definici ve formě BNF vytvoříme funkci, jejíž název má prefix `p_` a dokumentační řetězec obsahuje pravidla formy BNF, která má daná funkce zpracovat. V průběhu syntaktické analýzy volá analyzátor funkci se shodujícím se pravidlem formy BNF a předává jí jediný argument typu `ply.yacc.YaccProduction` s názvem `p` (podle konvenci pro pojmenování v příkladech nástroje PLY). Pokud pravidlo formy BNF obsahuje alternativy, pak můžeme vytvořit jen jednu funkci, která zpracuje všechny alternativy, i když ve většině případů bývá vytvoření jedné funkce pro každou alternativu nebo skupinu strukturálně podobných alternativ srozumitelnější. Podíváme se na každou funkci syntaktického analyzátoru a začneme tou, která má na starosti kvantifikátory.

```
def p_formula_quantifier(p):
    """FORMULA : FORALL SYMBOL COLON FORMULA
               | EXISTS SYMBOL COLON FORMULA"""
    p[0] = [p[1], p[2], p[4]]
```

Dokumentační řetězec obsahuje pravidlo formy BNF, kterému tato funkce odpovídá a jehož definice má místo `::=` dvojtečku. Všimněte si, že slova ve formě BNF jsou buď tokeny, s nimiž se shoduje lexikální analyzátor, nebo neterminály (např. *`FORMULA`*), s nimiž se shoduje forma BNF. Je třeba si dát pozor na jednu podivnost nástroje PLY: pokud totiž máme alternativy jako v tomto případě, musí být každá na samostatném řádku v dokumentačním řetězci.

Definice neterminálu *`FORMULA`* ve formě BNF obsahuje mnoho alternativ, my jsme zde ale použili pouze ty části, jež se týkají kvantifikátorů, a ostatní alternativy vyřešíme v jiných funkcí. Argument p typu `ply.yacc.YaccProduction` podporuje rozhraní API jazyka Python pro posloupnosti s tím, že každý prvek odpovídá určitému prvku ve formě BNF. Ve všech případech tedy platí, že `p[0]` odpovídá definovanému neterminálu (v tomto případě *`FORMULA`*) a ostatní prvky se shodují s částmi na pravé straně. Ve výše uvedené funkci se `p[1]` shoduje s jedním ze symbolů `"existuje"` nebo

Typy představující posloupnost ➤ 110

`"provšechna"`, `p[2]` se shoduje s kvantifikovaným identifikátorem (typicky `x` nebo `y`), `p[3]` se shoduje s tokenem *`COLON`* (literál `:`, který ignorujeme) a `p[4]` se shoduje s kvantifikovanou formulí. Jedná se o rekurzivní definici, takže samotný prvek `p[4]` je formule, která může obsahovat další formule a tak pořád dál. O bílé místo mezi tokeny se starat nemusíme, protože jsme vytvořili regulární výraz `t_ignore`, který lexikálnímu analyzátoru oznámil, že má bílé místo ignorovat (tj. přeskakovat).

V tomto příkladu bychom mohli stejně jednoduše vytvořit dvě samostatné funkce, třeba `p_formula_forall()` a `p_formula_exists()`, a každé přidělit jednu alternativu z formy BNF a stejnou sadu. My jsme se ale rozhodli pro jejich sloučení (a také pro sloučení dalších) jednoduše proto, že mají stejné sady.

Formule ve formě BNF mají tři binární operátory zahrnující formule. Všechny lze zpracovat jedinou sadou, a proto jsme se rozhodli provést jejich syntaktickou analýzu pomocí jediné funkce a formy BNF s alternativami.

```
def p_formula_binary(p):
    """FORMULA : FORMULA IMPLIES FORMULA
               | FORMULA OR FORMULA
               | FORMULA AND FORMULA"""
    p[0] = [p[1], p[2], p[3]]
```

Výsledkem (kterým je neterminál *`FORMULA`* uložený v `p[0]`) je jednoduše seznam obsahující levý operand, operátor a pravý operand. Tento kód neříká nic o precedenci či asociativitě. Přesto víme, že neterminál *`IMPLIES`* je zprava asociativní a ostatní dva jsou zleva asociativní a že neterminál *`IMPLIES`* má nižší precedenci než ostatní dva. Způsob ošetření těchto aspektů si ukážeme po prohlídce funkcí analyzátoru.

```
def p_formula_not(p):
    "FORMULA : NOT FORMULA"
    p[0] = [p[1], p[2]]

def p_formula_boolean(p):
    """FORMULA : FALSE
               | TRUE"""
    p[0] = p[1]

def p_formula_group(p):
    "FORMULA : LPAREN FORMULA RPAREN"
    p[0] = p[2]

def p_formula_symbol(p):
    "FORMULA : SYMBOL"
    p[0] = p[1]
```

Všechny tyto alternativy neterminálu *`FORMULA`* jsou unární. Přestože jsou však sady funkcí `p_formula_boolean()` a `p_formula_symbol()` stejné, přiřadili jsme každé z nich vlastní funkci, protože se z logického hlediska liší. Překvapivým aspektem funkce `p_formula_group()` je, že její hodnotu nenastavu-

jeme na `[p[1]]`, ale na `p[1]`. To je v pořádku, protože pro vyjádření všech operátorů již používáme seznamy. Použití seznamu na tomto místě by tedy bylo neškodné (a pro jiné analyzátory dokonce podstatné), ale v tomto příkladu naprosto zbytečné.

```
def p_formula_equals(p):
    "FORMULA : TERM EQUALS TERM"
    p[0] = [p[1], p[2], p[3]]
```

Toto je část formy BNF, která dává do souvislosti formule a termy. Implementace je přímočará a mohli bychom ji začlenit k ostatním binárním operátorům, protože sada této funkce je stejná. Rozhodli jsme se řešit tento případ zvlášť jen kvůli tomu, že je z logického hlediska oproti ostatním binárním operátorům jiný.

```
def p_term(p):
    """TERM : SYMBOL LPAREN TERMLIST RPAREN
            | SYMBOL"""
    p[0] = p[1] if len(p) == 2 else [p[1], p[3]]

def p_termlist(p):
    """TERMLIST : TERM COMMA TERMLIST
                | TERM"""
    p[0] = p[1] if len(p) == 2 else [p[1], p[3]]
```

Termem může být buď jediný symbol, nebo symbol následovaný seznamem termů v závorce (seznam termů oddělených čárkou); tyto dvě funkce se postarají o oba případy.

```
def p_error(p):
    if p is None:
        raise ValueError("Neznámá chyba")
    raise ValueError("Syntaktická chyba, řádek {0}: {1}".format(
                     p.lineno + 1, p.type))
```

Pokud při analýze dojde k nějaké chybě, zavolá se funkce `p_error()`. Přestože jsme s argumentem typu `ply.yacc.YaccProduction` až dosud pracovali jen jako s pouhou posloupností, nabízí také několik atributů. My jsme zde použili atribut `lineno` pro označení místa výskytu problému.

```
precedence = (("nonassoc", "FORALL", "EXISTS"),
              ("right", "IMPLIES"),
              ("left", "OR"),
              ("left", "AND"),
              ("right", "NOT"),
              ("nonassoc", "EQUALS"))
```

Pro nastavení precedencí a asociativit operátorů v analyzátoru na bázi nástroje PLY musíme vytvořit proměnnou precedence a předat jí seznam n-tic, jejichž první prvek představuje požadovanou asociativitu a druhý a následující prvky jsou příslušné tokeny. Nástroj PLY se bude řídit podle uvedených asociativit a nastaví precedenci od nejnižší (první n-tice v seznamu) po nejvyšší (poslední

n-tice v seznamu).* Pro unární operátory není v nástroji PLY asociativita ve skutečnosti žádný problém (i když v nástroji PyParsing být může), takže pro neterminál *NOT* bychom mohli použít `"nonassoc"` a nijak bychom tím výsledky analýzy neovlivnili.

V tomto okamžiku máme token, funkce lexikálního analyzátoru, funkce syntaktického analyzátoru a proměnnou `precedence`. Nyní tedy můžeme vytvořit lexikální a syntaktický analyzátor nástroje PLY a analyzovat nějaký text.

```
lexer = ply.lex.lex()
parser = ply.yacc.yacc()
try:
    return parser.parse(text, lexer=lexer)
except ValueError as err:
    print(err)
    return []
```

Tento kód analyzuje zadanou formuli a vrací seznam, který má naprosto stejný formát jako seznamy vracené verzí pro nástroj PyParsing. (Příklady tohoto druhu seznamů najdete na konci oddílu věnovaného analyzátoru logiky prvního řádu na bázi nástroje PyParsing – strana 527.)

Nástroj PLY vyvíjí maximální úsilí pro poskytování užitečných a zevrubných chybových zpráv, ačkoliv v některých případech může být až příliš horlivý. Když například nástroj PLY poprvé vytvoří analyzátor logiky prvního řádu, vypíše varování, že se vyskytlo „6 konfliktů posun/redukce“. V praxi pak nástroj PLY standardně použije posun, což je obvykle správně a zcela jistě se jedná o správnou akci pro analyzátor logiky prvního řádu. Tento a mnoho dalších problémů, které mohou nastat, vysvětluje dokumentace nástroje PLY, přičemž soubor analyzátoru `parser.out` vytvářený při každém vytvoření analyzátoru obsahuje všechny informace nezbytné pro zjištění všech souvisejících podrobností. Nepsaným pravidlem je, že varování ohledně posunu/redukce mohou být neškodná, ovšem jakýkoliv jiný druh varování je nutné eliminovat opravou analyzátoru.

Prohlídka příkladů postavených na nástroji PLY je u konce. Dokumentace k nástroji PLY (*www.dabeaz.com/ply*) nabízí mnohem více informací, než poskytuje prostor v této knize – například kompletní výklad všech možností nástroje PLY včetně mnoha takových, které jsme pro příklady v této lekci vůbec nepotřebovali.

Shrnutí

Pro nejjednodušší situace a pro nerekurzivní gramatiky jsou regulární výrazy dobrou volbou, tedy alespoň pro ty, kteří se v syntaxi regulárních výrazů cítí jako doma. Další možností je vytvoření konečného automatu – například čtením textu znak po znaku a udržováním jedné či více stavových proměnných, což ale může vést k příkazům `if` se spoustou klauzulí `elif` a vnořených příkazů `if ... elif`, které mohou být obtížně udržovatelné. Pro složitější a rekurzivní gramatiky jsou oproti regulárním výrazům, konečným automatům nebo ruční tvorbě rekurzivně sestupného analyzátoru mnohem lepší volbou nástroje PyParsing, PLY a další generické generátory analyzátorů.

* V nástroji PyParsing se precedence nastavují obráceně, od nejvyšší po nejnižší.

Ze všech těchto přístupů vyžaduje nejméně kódu nástroj PyParsing, může v něm ale být obtížné, tedy alespoň napoprvé, správně implementovat rekurzivní gramatiky. Nástroj PyParsing funguje nejlépe tehdy, když plně využijeme jeho předdefinované prvky (kterých je kromě těch, kterým jsme se věnovali v této lekci, ještě mnohem více) a použijeme takové programovací vzory, které se k nim hodí. To znamená, že ve složitějších případech nemůžeme jednoduše převést formu BNF přímo do syntaxe nástroje PyParsing, ale musíme její implementaci přizpůsobit tak, aby zapadla do filozofie tohoto nástroje. PyParsing je skvělý nástroj, který se používá ve spoustě softwarových projektů.

Nástroj PLY nejen že přímý převod forem BNF podporuje, ale přímo jej od nás vyžaduje, tedy alespoň co se modulu `ply.yacc` týče. Dále nabízí výkonný a flexibilní lexikální analyzátor, který je pro řadu jednoduchých gramatik dostačující jen sám o sobě. Nástroj PLY disponuje také skvělým hlášením chyb. Používá algoritmus řízený tabulkami, který činí jeho rychlost nezávislou na velikosti nebo složitosti gramatiky, a proto má tendenci běžet rychleji než analyzátory používající rekurzivní sestup, mezi něž patří kupříkladu nástroj PyParsing. Jediným aspektem, na který může být někdy nutné si zvykat, je jeho naprostá závislost na introspekci, takže podstatný význam mají jak dokumentační řetězce, tak i názvy funkcí. Nicméně PLY je skvělý nástroj, který se již nějaký čas používá pro tvorbu složitějších analyzátorů, mezi něž patří například analyzátory pro programovací jazyky C a ZXBasic.

Přestože je tvorba analyzátoru přijímajícího platný vstup obecně přímočarou záležitostí, vytvoření takového, který přijímá veškerý platný vstup a veškerý neplatný vstup zamítá, může být docela obtížné. Vezměme například analyzátory logiky prvního řádu v poslední části této lekce. Přijímají všechny platné formule a zamítají všechny neplatné? A pokud bychom zvládli zamítat neplatný vstup, dovedli bychom poskytovat chybové zprávy, které by správně identifikovaly, o jaký problém se jedná a kde k němu došlo? Syntaktická analýza je rozsáhlé a fascinující téma a tato Lekce je navržena pro seznámení s úplnými základy, takže další studium a praktická zkušenost jsou nezbytnými předpoklady pro ty, kteří chtějí jít dále.

Dále jsme se v této lekci dotkli skutečnosti, že v podobném rozsahu jako standardní knihovna Pythonu existuje i řada vysoce kvalitních balíčků a modulů třetích stran, které poskytují velice užitečné dodatečné funkční prvky. Většina z nich je dostupná prostřednictvím seznamu balíčků pro jazyk Python (*pypi.python.org/pypi*), avšak některé lze objevit pouze pomocí vyhledávačů. Obecně lze říci, že pokud máte nějakou zvláštní potřebu, kterou standardní knihovna nenaplňuje, pak vždy stojí za to poohlédnout se před vytvořením vlastního řešení po modulech třetích stran.

Cvičení

Vytvořte vhodnou formu BNF a poté napište jednoduchý program pro analyzování bibliografie knih v základním formátu BibTeX, který vytváří výstup ve formě slovníku slovníků. Vstup může vypadat například takto:

```
@Book{erl09,
  author    = "Thomas Erl",
  title     = "SOA,
              Servisně orientovaná architektura",
  year      = 2009,
  publisher = "Computer Press"
```

```
}
Na základě toho vstupu bude mít očekávaný výstup podobu následujícího slovníku:
{'erl09': {
        'author': 'Thomas Erl',
        'publisher': 'Computer Press',
        'title': 'SOA, Servisně orientovaná architektura',
        'year': 2009
    }
}
```

Každá kniha má identifikátor, který by měl být použitý jako klíč pro vnější slovník. Samotnou hodnotou by měl být slovník prvků ve tvaru klíč-hodnota.

Identifikátor každé knihy může obsahovat libovolné znaky kromě bílého místa a hodnotou každého pole *klíč*:*hodnota* může být buď celé číslo, nebo řetězec ve dvojitých uvozovkách. Řetězcové hodnoty mohou obsahovat libovolné bílé místo včetně nových řádků, takže je třeba nahradit každou interní posloupnost bílých míst (včetně nových řádků) jedinou mezerou a samozřejmě oříznout bílé místo z konců. Všimněte si, že za poslední dvojicí *klíč*:*hodnota* není uvedena čárka.

Analyzátor vytvořte pomocí nástroje PyParsing nebo PLY. Při použití nástroje PyParsing se pro identifikátor bude hodit třída `Regex()` a pro definování hodnoty třída `QuotedString()`. Pro zpracování seznamu *klíč*:*hodnota* použijte funkci `delimitedList()`. Pokud použijete nástroj PLY, pak postačí lexikální analyzátor, ovšem za předpokladu, že pro celočíselné a řetězcové hodnoty použijete samostatné tokeny.

Řešení využívající nástroj PyParsing by mělo mít kolem 30 řádků, zatímco řešení postavené na nástroji PLY může mít kolem 60 řádků. Řešení zahrnující obě verze pro PyParsing i PLY je k dispozici v souboru `BibTeX.py`.

LEKCE 15

Seznámení s programováním grafického uživatelského rozhraní

V této lekci:

- Programy ve stylu dialogového okna
- Programy s hlavním oknem

Python nabízí nativní podporu pro programování grafického uživatelského rozhraní (Graphical User Interface, zkráceně GUI). Programátoři v Pythonu mohou navíc využít řadu knihoven GUI napsaných v jiných jazycích. To je možné díky tomu, že spousta knihoven GUI nabízí tzv. *obaly* (wrapper) či *vazby* (bindings) pro jazyk Python – jedná se o balíčky a moduly, které se importují a používají jako ostatní balíčky a moduly Pythonu, které ale přistupují k funkčnosti, jež se nachází v knihovnách jiných programovacích jazyků.

Standardní knihovna Pythonu obsahuje podporu pro Tcl/Tk. Tcl je skriptovací jazyk s velice jednoduchou syntaxí a Tk je knihovna GUI napsaná v jazycích Tcl a C. Modul Pythonu `tkinter` poskytuje vazbu na knihovnu GUI Tk. Knihovna Tk má ve srovnání s ostatními knihovnami GUI dostupnými pro Python tři výhody. Za prvé se instaluje standardně s Pythonem, takže je vždy k dispozici. Za druhé je velice malá (i včetně jazyka Tcl). A za třetí dodává se s editorem IDLE, což je velice užitečné pro experimentování s Pythonem a pro úpravu a ladění programů napsaných v Pythonu.

3.x

Naneštěstí měla knihovna Tk před verzí 8.5 velmi zastaralý vzhled a velice omezenou sadu ovládacích prvků. I když je v knihovně Tk docela jednoduché vytvořit nové ovládací prvky složením jiných ovládacích prvků do určitého layoutu, knihovna Tk neposkytuje žádný přímý způsob pro tvorbu vlastních ovládacích prvků, u nichž by měl programátor možnost kreslit kdekoliv, kde potřebuje. Dodatečné ovládací prvky kompatibilní s knihovnou Tk jsou dostupné prostřednictvím knihovny Ttk (pouze s Pythonem 3.1 a knihovnou Tk ve verzi 8.5 a novější) a knihovny Tix, které jsou též součástí standardní knihovny Pythonu. Je třeba poznamenat, že knihovna Tix je obvykle k dispozici pouze na platformě Windows a na ostatních nemusí být podporována, zvláště pak na systému Ubuntu, který ji v době psaní této knihy nabízí pouze jako nepodporovaný doplněk, takže pro maximální přenositelnost je nejlepší knihovnu Tix vůbec nepoužívat. Dokumentace pro knihovny Tk, Ttk a Tix změřená na jazyk Python je poměrně vzácná. Většina dokumentace pro tyto knihovny je napsána pro programátory v jazyku Tcl, takže pro ostatní může být obtížné ji dešifrovat.

Pro vývoj programů GUI, které musejí běžet na kterékoli nebo na všech desktopových platformách Pythonu (např. Windows, Mac OS X a Linux) a používat pouze standardní instalaci Pythonu bez dodatečných knihoven, existuje pouze jediná volba: knihovna Tk.

Je-li možné použít knihovny třetích stran, pak počet možností výrazně narůstá. Jednou z nich je například sada WCK (Widget Construction Kit, *www.effbot.org/zone/wck.htm*), která poskytuje dodatečné funkční prvky kompatibilní s knihovnou Tk, včetně možnosti vytvářet vlastní ovládací prvky, jejichž obsah lze kreslit přímo v kódu.

Další možnosti jdou mimo knihovnu Tk a spadají do dvou kategorií podle toho, zda jsou specifické pro určitou platformu, nebo zda jsou přenositelné. Knihovny GUI specifické pro určitou platformu nám mohou dát přístup k prvkům dostupným pouze na dané platformě, avšak za cenu nemožnosti přejít na platformu jinou. Mezi tři dobře zavedené multiplatformní knihovny GUI s vazbou na jazyk Python patří PyGtk (*www.pygtk.org*), PyQt (*www.riverbankcomputing.com/software/pyqt*) a wxPython (*www.wxpython.org*). Všechny tyto knihovny nabízejí ve srovnání s knihovnou Tk mnohem více ovládacích prvků, vytvářejí lépe vypadající grafická uživatelská rozhraní (i když jejich náskok snížila verze knihovny Tk 8.5 a ještě více pak knihovna Ttk) a umožňují tvorbu vlastních ovládacích prvků kreslených přímo v kódu. Všechny se ve srovnání s knihovnou Tk snadněji učí a používají a nabízejí bohatší a mnohem lepší dokumentaci zaměřenou na jazyk Python. Obecně lze říci, že programy používající knihovnu PyGtk, PyQt nebo wxPython potřebují ve srovnání s programy napsanými pomocí

knihovny Tk méně kódu a produkují lepší výsledky. (V době psaní této knihy byla již knihovna PyQt přenesena do verze Python 3, avšak přenos knihoven wxPython a PyGtk ještě stále probíhal.)

Přesto navzdory jejím omezením a nezdarům lze knihovnu Tk použít pro sestavování užitečných programů GUI – z nichž je ve světě Pythonu nejznámější editor IDLE. Kromě toho to vypadá, že se vývoj knihovny Tk v poslední době vzchopil. Verze 8.5 nabízí použití motivů, díky čemuž pak programy vypadají mnohem nativněji, a dále vítané doplnění mnoha nových ovládacích prvků.

Smyslem této lekce je nabídnout ochutnávku programování s knihovnou Tk. Pro seriózní vývoj aplikací GUI je nejlepší tuto aplikaci přeskočit (poněvadž si v ní ukážeme poněkud specifický přístup knihovny Tk k programování grafického uživatelského rozhraní) a použít jednu z alternativních knihoven. Je-li však knihovna Tk vaší jedinou možností (např. proto, že vaši uživatelé mají pouze standardní instalaci Pythonu a nemohou nebo nechtějí instalovat knihovnu GUI třetí strany), pak se budete muset do jisté míry naučit jazyk Tcl, abyste byly schopni číst dokumentaci ke knihovně Tk.*

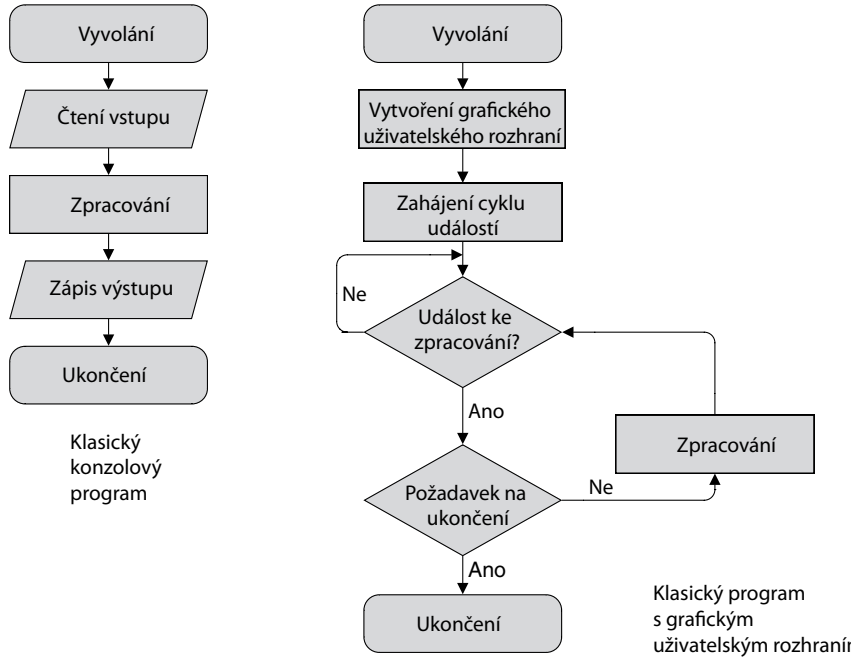
V následujících částech použijeme knihovnu Tk pro vytvoření dvou programů GUI. Prvním je velice malý prográmek ve stylu dialogového okna, který provádí výpočet složeného úroku. Druhým je propracovanější program s hlavním oknem, který spravuje seznam záložek (názvů a adres URL). Díky jednoduchým datům se můžeme nerušeně soustředit na ty aspekty programování, které souvisejí s grafickým uživatelským rozhraním. Při probírání programu se záložkami si ukážeme, jak vytvořit vlastní dialogové okno, jak vytvořit hlavní okno s nabídkami a panely nástrojů a jak jejich zkombinováním vytvořit kompletní fungující program.

Oba ukázkové programy používají jen knihovnu Tk bez knihoven Ttk a Tix, čímž zajistíme kompatibilitu s Pythonem 3.0. Není těžké je upravit, aby používaly knihovnu Ttk, avšak v době psaní této knihy poskytovaly některé ovládací prvky knihovny Ttk ve srovnání s jejich protějšky z knihovny Tk menší podporu pro uživatele s klávesnicí. Programy postavené na knihovně Ttk sice mohou vypadat lépe, jejich použití ale může být méně pohodlné.

Ještě dříve, než se pustíme do samotného kódu, se musíme podívat na základy programování grafického uživatelského rozhraní, které se totiž od psaní konzolových programů malinko liší.

Konzolové programy a soubory modulů napsané v jazyku Python mají vždy příponu `.py`, ovšem pro programy GUI napsané v Pythonu používáme příponu `.pyw` (soubory modulů přesto používají vždy příponu `.py`). Na Linuxu fungují bez problémů oba typy souborů, `.py` i `.pyw`, ale ve Windows je nutné použít soubory `.pyw`, které zajišťují, že se místo interpretu `python.exe` použije `pythonw.exe` a že se při spuštění programu GUI neobjeví žádné zbytečné konzolové okno. Systém Mac OS X funguje podobně jako Windows, takže pro programy GUI používá příponu `.pyw`.

* Pravděpodobně jedinou knihou o jazyku Python a knihovně Tk je kniha „Python and Tkinter Programming" od Johna Graysona vydaná v roce 2000. V některých oblastech je však již dnes zastaralá. Kvalitní knihou o jazyku Tcl a knihovně Tk je „Practical Programming in Tcl and Tk" od Brenta Welche a Kena Jonese. Veškerá dokumentace je k dispozici na webu *www.tcl.tk* a výukové lekce naleznete na adrese *www.tkdocs.com*.



Obrázek 15.1: Konzolové programy vs. programy s grafickým uživatelským rozhraním

Programu GUI při svém spuštění běžně začíná vytvořením svého hlavního okna a všech ovládacích prvků hlavního okna, jako je panel nabídek, panely nástrojů, centrální oblast a stavový panel. Po vytvoření hlavního okna program GUI, podobně jako serverový program, jednoduše čeká. Zatímco server čeká na připojení klientského programu, program GUI čeká na interakci uživatele, jako jsou klepnutí myši a stisky kláves. Srovnání s konzolovými programy znázorňuje obrázek 15.1. Program GUI nečeká pasivně, ale běží v cyklu událostí, který vypadá v pseudokódu takto:

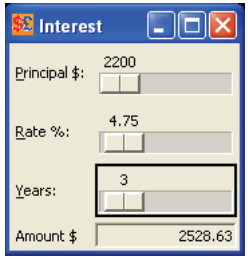
```
while True:
    event = getNextEvent()
    if event:
        if event == Terminate:
            break
        processEvent(event)
```

Když dojde k interakci uživatele s programem nebo k nějaké jiné akci, jako je kupříkladu tiknutí časovače nebo aktivace okna programu (třeba kvůli uzavření jiného programu), vygeneruje se uvnitř knihovny GUI událost, která se přidá do fronty událostí. Cyklus událostí programu neustále kontroluje, zda je k dispozici nějaká událost na zpracování, a pokud ano, zpracuje ji (nebo ji předá na zpracování funkci či metodě spojené s danou událostí).

Jako programátoři grafického uživatelského rozhraní se můžeme opřít o knihovnu GUI, která poskytuje cyklus událostí. Naší odpovědností je vytvořit třídy reprezentující okna a ovládací prvky potřebné v našem programu a vybavit je metodami, které odpovídají na příslušné interakce uživatele.

Programy ve stylu dialogových oken

První program, na který se podíváme, je program Úrok. Jedná se o program ve stylu dialogového okna (tj. nemá žádné nabídky), pomocí něhož mohou uživatelé vypočítat složený úrok. Program je zachycen na obrázku 15.2.



Obrázek 15.2: Program Úrok

Ve většině objektově orientovaných programů GUI se pro reprezentaci jediného hlavního či dialogového okna používá vlastní třída, přičemž většina v něm obsažených ovládacích prvků jsou instance standardních ovládacích prvků, jako jsou tlačítka nebo zaškrtávací pole, poskytované knihovnou. Knihovna Tk stejně jako většina multiplatformních knihoven GUI ve skutečnosti nerozlišuje mezi oknem a ovládacím prvkem. Okno je jednoduše ovládací prvek, který nemá nadřízený ovládací prvek (tj. není obsažen uvnitř jiného ovládacího prvku). Ovládací prvky, které nemají nadřízený ovládací prvek (okna), jsou automaticky doplněny rámem a dekoracemi okna (mezi něž patří například panel záhlaví a tlačítka pro uzavření) a obvykle obsahují další ovládací prvky.

Většina ovládacích prvků je vytvářena jako potomci jiného ovládacího prvku (a jsou obsaženi uvnitř svého rodiče), zatímco okna jsou vytvářena jako potomci objektu typu `tkinter.Tk`, který představuje aplikaci a k němuž se vrátíme později. Kromě rozlišování mezi ovládacími prvky a okny (označovanými též jako ovládací prvky nejvyšší úrovně) pomáhá vztah rodič-potomek zajistit, aby se ovládací prvky vymazaly ve správném pořadí a aby se při vymazání nadřízeného ovládacího prvku vymazaly také jeho podřízené ovládací prvky.

V inicializační metodě se vytváří uživatelské rozhraní (rozvrhnou se přidávané ovládací prvky, ustaví se vazby na myši a klávesnici) a ostatní metody se používají pro odpovědi na interakce uživatele. Knihovna Tk nám umožňuje vytvářet vlastní ovládací prvky buď odvozením od předdefinovaného ovládacího prvku (jako je např. `tkinter.Frame`), nebo vytvořením běžné třídy, do níž přidáme ovládací prvky jako atributy. Zde používáme odvození – v dalším příkladu si ukážeme oba přístupy.

Program Úrok má jen jedno hlavní okno, které je implementováno v jediné třídě. Začneme tedy pohledem na inicializační metodu této třídy, kterou si kvůli její délce rozdělíme na pět částí.

```
class MainWindow(tkinter.Frame):

    def __init__(self, parent):
        super().__init__(parent)
        self.parent = parent
        self.grid(row=0, column=0)
```

Začínáme inicializací bázové třídy a uložením kopie nadřízeného ovládacího prvku (`parent`) pro pozdější použití. Místo absolutních pozic a velikosti se ovládací prvky rozvrhují uvnitř jiných ovládacích prvků pomocí správců rozvržení. Voláním metody `grid()` rozvrhneme rám pomocí mřížkového správce rozvržení. Všechny zobrazované ovládací prvky včetně prvků na nejvyšší úrovni musejí být rozvrženy. Knihovna Tk obsahuje několik správců rozvržení, z nichž je mřížka nejsrozumitelnější a nejsnadněji se používá, i když pro rozvržení na nejvyšší úrovni, kde máme pouze jediný ovládací prvek, bychom mohli zavoláním `pack()` místo `grid(row=0, column=0)` použít balicího správce rozvržení a docílit tak stejného efektu.

```
self.principal = tkinter.DoubleVar()
self.principal.set(1000.0)
self.rate = tkinter.DoubleVar()
self.rate.set(5.0)
self.years = tkinter.IntVar()
self.amount = tkinter.StringVar()
```

Knihovna Tk nám umožňuje vytvářet proměnné, které jsou spojené s určitými ovládacími prvky. Je-li hodnota takovéto proměnné v programu změněna, projeví se tato změna v přidruženém ovládacím prvku. A podobně pokud uživatel změní hodnotu v ovládacím prvku, změní se hodnota v přidružené proměnné. Zde jsme vytvořili dvě proměnné s pohyblivou řádovou čárkou, celočíselnou proměnnou a řetězcovou proměnnou a pro dvě z nich jsme nastavili počáteční hodnoty.

```
principalLabel = tkinter.Label(self, text="Základ Kč:",
                               anchor=tkinter.W, underline=0)
principalScale = tkinter.Scale(self, variable=self.principal,
        command=self.updateUi, from_=100, to=10000000,
        resolution=100, orient=tkinter.HORIZONTAL)
rateLabel = tkinter.Label(self, text="Sazba %:", underline=0,
                          anchor=tkinter.W)
rateScale = tkinter.Scale(self, variable=self.rate,
        command=self.updateUi, from_=1, to=100,
        resolution=0.25, digits=5, orient=tkinter.HORIZONTAL)
yearsLabel = tkinter.Label(self, text="Počet let:", underline=0,
                           anchor=tkinter.W)
yearsScale = tkinter.Scale(self, variable=self.years,
        command=self.updateUi, from_=1, to=50,
        orient=tkinter.HORIZONTAL)
amountLabel = tkinter.Label(self, text="Částka Kč",
                            anchor=tkinter.W)
actualAmountLabel = tkinter.Label(self,
        textvariable=self.amount, relief=tkinter.SUNKEN,
        anchor=tkinter.E)
```

V této části inicializační metody vytváříme ovládací prvky. Ovládací prvek `tkinter.Label` se používá pro zobrazení textu určeného pouze pro čtení. Vytváříme jej jako u všech ovládacích prvků s rodičem (v tomto případě – a jako obvykle – je rodičem obklopující ovládací prvek) a poté nastavíme nejrůznější aspekty jeho vzhledu a chování pomocí klíčovaných argumentů. U ovládacího prvku `principalLabel`

jsme odpovídajícím způsobem nastavili `text` a jeho ukotvení (`anchor`) jsme nastavili na hodnotu `tkinter.W`, což znamená, že se text popisku zarovná západně (vlevo). Parametr `underline` se používá pro stanovení, který znak v popisku by měl být podtržen pro zvýraznění *akcelerátoru* (např. Alt+P). Později uvidíme, jak tyto akcelerátory zprovoznit. (Akcelerátor je posloupnost kláves ve tvaru Alt+písmeno, kde písmeno je ono podtržené písmeno a jejímž výsledkem je přesun kurzoru na ovládací prvek spojený s daným akcelerátorem, což je nejčastěji ovládací prvek napravo nebo pod popiskem s akcelerátorem.)

Ovládacím prvkům typu `tkinter.Scale` jsme jako obvykle přidělili nadřízený ovládací prvek `self` a s každým z nich jsme spojili jednu proměnnou. Dále jsme jako jejich příkaz (`command`) uvedli odkaz na funkční objekt (nebo v tomto případě metodu). Tato metoda se automaticky zavolá při každé změně hodnoty stupnice. Nastavili jsme také hodnoty pro její minimum (`from_` – bez podtržítka jde o klíčové slovo) a maximum (`to`) a určili jsme vodorovnou orientaci (`orient`). Pro některé stupnice nastavujeme rozlišení (`resolution` – velikost kroku) a pro stupnici `rateScale` i počet číslic, které musí být schopna zobrazit.

Ovládací prvek `actualAmountLabel` je také spojen s proměnnou, abychom mohli později snadno změnit text v popisku. Tomuto popisku jsme dále přidělili vpadlý reliéf, aby vizuálně lépe zapadl k našim stupnicím.

```
principalLabel.grid(row=0, column=0, padx=2, pady=2,
                    sticky=tkinter.W)
principalScale.grid(row=0, column=1, padx=2, pady=2,
                    sticky=tkinter.EW)
rateLabel.grid(row=1, column=0, padx=2, pady=2,
               sticky=tkinter.W)
rateScale.grid(row=1, column=1, padx=2, pady=2,
               sticky=tkinter.EW)
yearsLabel.grid(row=2, column=0, padx=2, pady=2,
                sticky=tkinter.W)
yearsScale.grid(row=2, column=1, padx=2, pady=2,
                sticky=tkinter.EW)
amountLabel.grid(row=3, column=0, padx=2, pady=2,
                 sticky=tkinter.W)
actualAmountLabel.grid(row=3, column=1, padx=2, pady=2,
                       sticky=tkinter.EW)
```

Po vytvoření ovládacích prvků je musíme rozvrhnout. Námi použité mřížkové rozvržení znázorňuje obrázek 15.3.

principalLabel	principalScale
rateLabel	rateScale
yearsLabel	yearsScale
amountLabel	actualAmountLabel

Obrázek 15.3: Rozvržení ovládacích prvků v programu Úrok

Každý ovládací prvek podporuje metodu `grid()` (a některé další metody rozvrhování prvků, jako je například `pack()`). Při zavolání metody `grid()` se ovládací prvky rozvrhnout uvnitř svého rodiče tak, aby obsadily zadaný řádek a sloupec. Ovládací prvky můžeme pomocí klíčovaných argumentů `rowspan` a `columnspan` nastavit tak, aby se rozkládaly přes více sloupců nebo řádků, a pomocí klíčovaných argumentů `padx` (levý a pravý vnější okraj) a `pady` (horní a dolní vnější okraj) můžeme kolem nich přidat vnější okraj zadávaný jako celé číslo představující počet pixelů. Je-li ovládacímu prvku přiřazeno více prostoru, než potřebuje, použije se pro určení, co se má s tímto prostorem stát, jeho volba pro přichycení (`sticky`). Není-li zadána, zabere ovládací prvek centrální část přiděleného prostoru. U všech popisků v prvním sloupci jsme volbu `sticky` nastavili na hodnotu `tkinter.W` (západ) a u ovládacích prvků ve druhém sloupci na hodnotu `tkinter.EW` (východ a západ), díky níž se roztáhnou tak, aby vyplnily celou jim dostupnou šířku.

Všechny ovládací prvky jsou uchovávány v lokálních proměnných, nejsou ale naplánovány na úklid z paměti, protože vztah rodič-potomek zajišťuje, že se po opuštění oboru platnosti na konci inicializační metody nesmažou, neboť všechny mají jako svého rodiče hlavní okno. Někdy se ovládací prvky vytvářejí jako proměnné instance, například tehdy, když se na ně potřebujeme odkazovat mimo inicializační metodu. V tomto případě jsme ale proměnné instance použili pro proměnné spojené s ovládacími prvky (`self.principal`, `self.rate` a `self.years`) a tyto proměnné pak použijeme mimo inicializační metodu.

```
principalScale.focus_set()
self.updateUi()
parent.bind("<Alt-z>", lambda *ignore: principalScale.focus_set())
parent.bind("<Alt-s>", lambda *ignore: rateScale.focus_set())
parent.bind("<Alt-p>", lambda *ignore: yearsScale.focus_set())
parent.bind("<Control-k>", self.quit)
parent.bind("<Escape>", self.quit)
```

Na konci inicializační metody přidělíme zaměření klávesnice ovládacímu prvku `principalScale`, takže ihned po spuštění programu může uživatel nastavit počáteční částku. Potom zavoláme metodu `self.updateUi()`, která vypočítá výslednou částku.

Dále ustavíme několik klávesových vazeb. (*Vazba* má naneštěstí hned tři různé významy. Svázání proměnné je akce, při níž se název, tj. odkaz na objekt, sváže s nějakým objektem. Klávesová vazba je akce, při níž se událost klávesnice, jako stisk či uvolnění klávesy, spojí s určitou funkcí či metodou, která se má zavolat při výskytu této události. A vazba pro knihovnu je spojovací kód, díky němuž je knihovna napsaná v jiném jazyku dostupná prostřednictvím modulů Pythonu i pro programátory v jazyku Python.) Klávesové vazby jsou naprosto zásadní pro postižené uživatele, kteří nemohou používat nebo jen obtížně používají myš, a navíc jsou skvělou pomůckou pro uživatele s rychlými prsty, které myš při práci zdržuje.

První tři klávesové vazby se používají pro přesun zaměření klávesnice na ovládací prvek stupnice. Například text ovládacího prvku `principalLabel` je nastaven na `"Základ Kč:"` a index pro podtržení (`underline`) na 0, takže popisek se zobrazí jako „Základ Kč:“. S ustavenou první klávesovou vazbou pak dojde k tomu, že když uživatel stiskne kombinaci kláves Alt+P, přepne se zaměření klávesnice na ovládací prvek `principleScale`. Totéž platí o pro další dvě vazby. Všimněte si, že metodu `focus_set()` nevážeme přímo. To je dáno tím, že se funkcím či metodám volaným

jako výsledek vazby události předává jako první argument událost, která je vyvolala, ale kterou my nechceme. Proto používáme lambda funkci, která událost přijímá, ale ignoruje a volá metodu bez nechtěného argumentu.

Dále jsme vytvořili dvě *klávesové zkratky* – jedná se o kombinace kláves, které vyvolávají určitou akci. Zde jsme nastavili Ctrl+K a Esc a svázali je s metodou `self.quit()`, která čistě ukončí program.

Je též možné vytvořit klávesové vazby pro jednotlivé ovládací prvky, zde jsme ale všechny nastavili na rodiči (kterým je aplikace), takže fungují bez ohledu na to, kde se zaměření klávesnice nachází.

Metodu knihovny Tk s názvem `bind()` lze použít pro svázání klepnutí myší i stisknutí klávesy a také pro svázání událostí definovaných programátorem. Speciální klávesy, jako jsou Ctrl a Esc, mají v knihovně Tk specifické názvy (`Control` a `Escape`), přičemž běžná písmena odpovídají příslušným klávesám. Klávesové posloupnosti se vytvářejí umístěním do lomených závorek a oddělením pomlčkou.

Po vytvoření a rozvržení ovládacích prvků a po ustavení klávesových vazeb máme připravený vzhled a základní chování programu. Nyní se podíváme na metody, které odpovídají na akce uživatele, čímž dokončíme implementaci chování programu.

```
    def updateUi(self, *ignore):
        amount = self.principal.get() * (
                 (1 + (self.rate.get() / 100.0)) ** self.years.get())
        self.amount.set("{0:.2f}".format(amount))
```

Tato metoda se volá při každé změně základní částky, sazby nebo počtu let. Jedná se totiž o příkaz spojený s každou z našich stupnic. Nedělá nic jiného, než že získá hodnoty z proměnné přidružené ke každé stupnici, provede výpočet složeného úroku a výsledek uloží (jako řetězec) do proměnné spojené s popiskem výsledné částky. Díky tomu je v tomto popisku vždy aktuální částka.

```
    def quit(self, event=None):
        self.parent.destroy()
```

Pokud se uživatel rozhodne ukončit program (stiskem kláves Ctrl+K nebo Esc nebo klepnutím na uzavírací tlačítko okna), zavolá se tato metoda. Žádná data nemusíme ukládat, a proto stačí oznámit rodičovskému ovládacímu prvku (kterým je objekt aplikace), aby se zrušil. Rodič pak zruší všechny své potomky (všechna okna, která následně zruší všechny své ovládací prvky), takže se provede čisté ukončení.

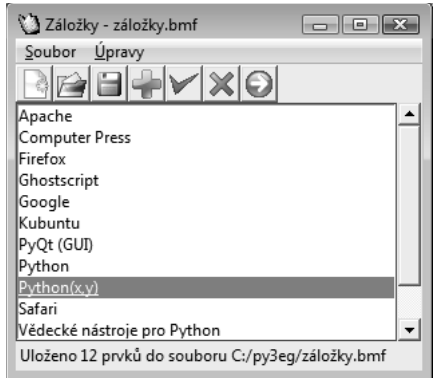
```
application = tkinter.Tk()
path = os.path.join(os.path.dirname(__file__), "images/")
if sys.platform.startswith("win"):
    icon = path + "interest.ico"
else:
    icon = "@" + path + "interest.xbm"
application.iconbitmap(icon)
application.title("Úrok")
window = MainWindow(application)
application.protocol("WM_DELETE_WINDOW", window.quit)
application.mainloop()
```

Po definování třídy pro hlavní (a v tomto případě jediné) okno následuje kód, který spouští běh programu. Začínáme vytvoření objektu pro reprezentaci aplikace jako celku. Aby měl program ve Windows ikonku, použijeme soubor `.ico`, jehož název předáme (i s celou cestou) metodě `iconbitmap()`. Avšak pro unixové platformy musíme poskytnout bitmapu (tj. monochromatický obrázek). Knihovna Tk nabízí několik vestavěných bitmap, takže pro odlišení té, která pochází ze souborového systému, musíme před její název umístit symbol `@`. Dále přiřadíme aplikaci název (který se objeví v záhlaví) a poté vytvoříme instanci naší třídy `MainWindow`, které předáme aplikační objekt jako jejího rodiče. Na konci voláme metodu `protocol()`, jejímž prostřednictvím oznámíme, co by se mělo stát, pokud uživatel klepne na uzavírací tlačítko. V tomto případě chceme, aby se zavolala metoda `MainWindow.quit()`. Úplně nakonec spustíme cyklus událostí, přičemž až zde se zobrazí okno a je schopno reagovat na interakce uživatele.

Programy s hlavním oknem

Přestože programy ve stylu dialogového okna jsou pro jednoduché činnosti často dostatečné, se vzrůstající škálou funkcí nabízených programem je často rozumné vytvořit kompletní aplikaci s hlavním oknem s nabídkami a panelem nástrojů. Takovéto aplikace se ve srovnání s programy ve stylu dialogového okna obvykle snadněji rozšiřují, protože lze přidávat další nabídky nebo volby nabídky a tlačítka panelu nástrojů, aniž by to mělo nějaký vliv na rozvržení hlavního okna.

V této části se podíváme na program `bookmarks-tk.pyw` zachycený na obrázku 15.4. Tento program udržuje záložky jako dvojice řetězců (název, URL) a uživatelům nabízí možnost záložky přidávat, upravovat a odstraňovat a otevřít webový prohlížeč na webové stránce uložené pod určitou záložkou.



Obrázek 15.4: Program Záložky

Program obsahuje dvě okna: hlavní okno s panelem nabídek, panelem nástrojů, seznamem záložek a stavovým panelem a dialogové okno pro přidávání nebo úpravu záložek.

Vytvoření hlavního okna

Hlavní okno je podobné dialogovému oknu v tom, že má ovládací prvky, které je nutné vytvořit a rozvrhnout. A kromě toho mu musíme přidat panel nabídek, nabídky, panel nástrojů a stavový panel a také metody k provádění akci vyžádaných uživatelem.

Uživatelské rozhraní je zřízeno v inicializační metodě hlavního okna, kterou si kvůli její délce rozdělíme na pět částí.

```
class MainWindow:

    def __init__(self, parent):
        self.parent = parent

        self.filename = None
        self.dirty = False
        self.data = {}

        menubar = tkinter.Menu(self.parent)
        self.parent["menu"] = menubar
```

Pro toto okno jsme nepoužili odvození od ovládacího prvku jako v předchozím případě, ale vytvořili jsme běžnou třídu Pythonu. V případě odvození lze reimplementovat metody nadřazené třídy, v opačném případě stačí jednoduše použít kompozici, jak jsme to provedli i zde. O vzhled se postaráme vytvořením proměnných instance představujících ovládací prvky, které budou, jak uvidíme za okamžik, obsaženy v objektu typu `tkinter.Frame`.

Potřebujeme uchovávat čtyři údaje: rodičovský objekt (aplikace), název aktuálního souboru se záložkami, příznak změny záložek (hodnota `True` znamená, že v datech neuložených na disk došlo k nějakým změnám) a samotná data ve formě slovníku, jehož klíči jsou názvy záložek a hodnotami adresy URL.

Pro vytvoření panelu nabídek musíme vytvořit objekt typu `tkinter.Menu`, jehož rodičem je rodič tohoto okna, a dále musíme rodičovskému ovládacímu prvku sdělit, že obsahuje nabídku. (Možná to vypadá zvláštně, že panel nabídek je nabídka, ale knihovna Tk prošla velmi dlouhým vývojem, který v ní zanechal několik podivností.) Panely nabídek vytvořené tímto způsobem nepotřebují rozvržení. Postará se o něj za nás knihovna Tk.

```
        fileMenu = tkinter.Menu(menubar)
        for label, command, shortcut_text, shortcut in (
                ("Nový...", self.fileNew, "Ctrl+N", "<Control-n>"),
                ("Otevřít...", self.fileOpen, "Ctrl+O", "<Control-o>"),
                ("Uložit", self.fileSave, "Ctrl+U", "<Control-u>"),
                (None, None, None, None),
                ("Konec", self.fileQuit, "Ctrl+K", "<Control-k>")):
            if label is None:
                fileMenu.add_separator()
            else:
                fileMenu.add_command(label=label, underline=0,
                        command=command, accelerator=shortcut_text)
                self.parent.bind(shortcut, command)
        menubar.add_cascade(label="Soubor", menu=fileMenu, underline=0)
```

Každá nabídka v panelu nabídek se vytváří stejným způsobem. Nejdříve vytvoříme objekt typu `tkinter.Menu`, který je potomkem panelu nabídek, a poté do této nabídky přidáme oddělovače nebo příkazy. (Všimněte si, že akcelerátor v terminologii knihovny Tk je ve skutečnosti klávesovou zkratkou a že jsme všechny volby `accelerator` nastavili na text zkratky, čímž se ale ve skutečnosti neustaví žádná klávesová vazba.) Podtržítko (argument `underline`) určuje index znaku, který se má podtrhnout. V tomto případě se jedná o první znak každé položky nabídky, který se tak stane její akcelerační klávesou.

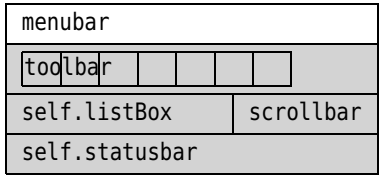
Kromě přidání položky nabídky (označované jako příkaz) definujeme také klávesovou zkratku svázáním posloupnosti kláves s daným příkazem, který se vyvolá při volbě odpovídající položky v nabídce. Na konci nabídku přidáme do panelu nabídek pomocí metody `add_cascade()`.

Nabídku „Úpravy“ jsme vynechali, protože její kód je z hlediska struktury naprosto stejný jako kód nabídky „Soubor“.

```
        frame = tkinter.Frame(self.parent)
        self.toolbar_images = []
        toolbar = tkinter.Frame(frame)
        for image, command in (
                ("images/filenew.gif", self.fileNew),
                ("images/fileopen.gif", self.fileOpen),
                ("images/filesave.gif", self.fileSave),
                ("images/editadd.gif", self.editAdd),
                ("images/editedit.gif", self.editEdit),
                ("images/editdelete.gif", self.editDelete),
                ("images/editshowwebpage.gif", self.editShowWebPage)):
            image = os.path.join(os.path.dirname(__file__), image)
            try:
                image = tkinter.PhotoImage(file=image)
                self.toolbar_images.append(image)
                button = tkinter.Button(toolbar, image=image,
                                        command=command)
                button.grid(row=0, column=len(self.toolbar_images) -1)
            except tkinter.TclError as err:
                print(err)
        toolbar.grid(row=0, column=0, columnspan=2, sticky=tkinter.NW)
```

Začínáme vytvořením rámu, v němž budou umístěny všechny ovládací prvky okna. Poté vytvoříme další rám (`toolbar`), který bude obsahovat vodorovný řádek tlačítek s obrázky místo textu sloužících jako tlačítka panelu nástrojů. Tlačítka panelu nástrojů umístíme jedno za druhým v mřížce s jedním řádkem a s tolika sloupci, kolik je tlačítek. Na konci umístíme samotný rám s panelem nástrojů jako první řádek rámu hlavního okna s přichycením k severní a východní straně, takže bude vždy přimknutý k levé horní straně okna. (Knihovna Tk automaticky umístí panel nabídek nad všechny ovládací prvky umístěné v okně.)

Rozvržení ovládacích prvků znázorňuje obrázek 15.5. Panel nabídek umístěný knihovnou Tk má bílé pozadí a námi umístěné prvky mají pozadí šedé.



Obrázek 15.5: Rozvržení hlavního okna programu Záložky

Když k tlačítku přidáme obrázek, přidá se jako slabý odkaz, takže jakmile opustí obor platnosti, naplánuje se na úklid z paměti. Tomu však musíme zabránit, protože chceme, aby se po dokončení inicializační metody zobrazily na tlačítkách obrázky, a proto vytvoříme proměnnou instance `self.toolbar_images`, která jen uchovává odkazy na obrázky, aby po dobu činnosti programu nedošlo k jejich odstranění.

Knihovna Tk dokáže sama o sobě číst pouze několik málo obrazových formátů, a proto musíme použít obrázky `.gif`.* Není-li možné některý z obrázků najít, vyvolá se výjimka `tkinter.TclError`, kterou nesmíme zapomenout zachytit, jinak by program skončil jen kvůli chybějícímu obrázku.

Všimněte si, že jsme záměrně nevytvořili tlačítka pro úplně všechny akce dostupné v nabídkách, což je běžná praxe.

```
scrollbar = tkinter.Scrollbar(frame, orient=tkinter.VERTICAL)
self.listBox = tkinter.Listbox(frame,
                               yscrollcommand=scrollbar.set)
self.listBox.grid(row=1, column=0, sticky=tkinter.NSEW)
self.listBox.focus_set()
scrollbar["command"] = self.listBox.yview
scrollbar.grid(row=1, column=1, sticky=tkinter.NS)

self.statusbar = tkinter.Label(frame, text="Připraven...",
                               anchor=tkinter.W)
self.statusbar.after(5000, self.clearStatusBar)
self.statusbar.grid(row=2, column=0, columnspan=2,
                    sticky=tkinter.EW)

frame.grid(row=0, column=0, sticky=tkinter.NSEW)
```

Centrální oblast hlavního okna (oblast mezi panelem nástrojů a stavovým panelem) zabírá seznam a s ním spojený posuvník. Seznam je umístěn s přichycením do všech směrů a posuvník je přichycen pouze k severní a jižní straně (svisle). Oba ovládací prvky přidáváme jeden vedle druhého do mřížky hlavního okna.

Musíme zajistit, aby v případě, když se uživatel přemístí na posuvník pomocí tabulátoru a použije kurzorové klávesy nahoru a dolů nebo když posune samotný posuvník, zůstaly oba ovládací prvky synchronizované. Toho docílíme tak, že nastavíme parametr seznamu `yscrollcommand` na metodu `set()` posuvníku (aby pohyb uživatele v seznamu způsobil případný přesun posuvníku) a parametr

* Je-li nainstalováno rozšíření knihovny Tk s názvem Python Imaging Library, pak jsou podporovány všechny moderní obrazové formáty. Více informací najdete na adrese *www.pythonware.com/products/pil/*.

posuvníku `command` na metodu `yview()` seznamu (aby pohyby posuvníku způsobily odpovídající přesun seznamu).

Stavový panel tvoří jen popisek. Metoda `after()` je neopakující se časovač (tj. časovač, který po zadaném intervalu skončí), jehož prvním argumentem je doba v milisekundách a druhým argumentem funkce či metoda, která se zavolá po dosažení zadané doby. To znamená, že po spuštění programu bude na stavovém panelu pět vteřin zobrazen text „Připraven...“, který se poté vymaže. Stavový panel je umístěn jako poslední řádek a je přichycen k západní a východní straně (vodorovně).

Na konci umístíme samotný rám okna. Dokončili jsme tvorbu a rozvržení ovládacích prvků hlavního okna, které ale za stávající situace použijí fixní výchozí velikost, takže při změně velikosti okna nebudou reagovat odpovídajícím způsobem zvětšením či zmenšením. Tento problém řeší následující úryvek kódu, který zároveň zakončuje inicializační metodu.

```
        frame.columnconfigure(0, weight=999)
        frame.columnconfigure(1, weight=1)
        frame.rowconfigure(0, weight=1)
        frame.rowconfigure(1, weight=999)
        frame.rowconfigure(2, weight=1)

        window = self.parent.winfo_toplevel()
        window.columnconfigure(0, weight=1)
        window.rowconfigure(0, weight=1)

        self.parent.geometry("{0}x{1}+{2}+{3}".format(400, 500,
                                                      0, 50))
        self.parent.title("Záložky - Bez názvu")
```

Metody `columnconfigure()` a `rowconfigure()` umožňují přidělit váhu buňkám mřížky. Začínáme rámem okna a přidělujeme veškerou váhu prvnímu sloupci a druhému řádku (na němž je seznam), takže při změně velikost rámu je všechen nadbytečný prostor přidělen seznamu. To ale samo o sobě nestačí. Musíme totiž umožnit změnu velikosti také pro okno na nejvyšší úrovni, jež obsahuje tento rám, což provádíme tak, že pomocí metody `wininfo_toplevel()` získáme odkaz na okno, u něhož poté aktivujeme změnu velikosti nastavením váhy pro jeho řádek a sloupec na 1.

Na konci inicializační metody nastavujeme počáteční velikost a pozici okna pomocí řetězce ve tvaru "*šířkaxvýška+x+y*". (Pokud bychom chtěli nastavit pouze velikost, mohli bychom použít kratší tvar "*šířkaxvýška*".) Nakonec nastavujeme název okna, čímž jsme dokončili přípravu uživatelského rozhraní tohoto okna.

Pokud uživatel klepne na tlačítko panelu nástrojů nebo pokud zvolí položku nabídky, zavolá se pro provedení požadované akce příslušná metoda. Některé z těchto metod se opírají o pomocné metody. Nyní se na jednotlivé metody postupně podíváme. Začneme tou, která se zavolá pět vteřin po spuštění programu.

```
    def clearStatusBar(self):
        self.statusbar["text"] = ""
```

Stavový panel je obyčejný ovládací prvek typu `tkinter.Label`. Pro jeho vymazání bychom sice při volání metody `after()` mohli použít lambda výraz, my ale potřebujeme vymazat stavový panel z více než jednoho místa, a proto jsme pro tento účel raději vytvořili samostatnou metodu.

```
def fileNew(self, *ignore):
    if not self.okayToContinue():
        return
    self.listBox.delete(0, tkinter.END)
    self.dirty = False
    self.filename = None
    self.data = {}
    self.parent.title("Záložky - Bez názvu")
```

Chce-li uživatel vytvořit nový soubor se záložkami, musíme mu nejdříve dát šanci uložit jakékoli neuložené změny stávajících záložek. Tuto funkčnost jsme vyčlenili do metody `MainWindow.okayToContinue()`, protože ji používáme na více různých místech. Tato metoda vrací hodnotu `True`, může-li operace pokračovat, nebo hodnotu `False` v opačném případě. Pokud pokračujeme, vyčistíme seznam vymazáním všech záznamů od prvního po poslední (`tkinter.END` je konstanta používaná pro označení posledního prvku v situacích, kdy nějaký ovládací prvek může obsahovat více prvků). Potom obnovíme do původního stavu příznak změny (`dirty`), název souboru (`filename`) a data, protože soubor je nový a nezměněný, a nastavíme název okna tak, aby odrážel skutečnost, že se pracuje s novým, neuloženým souborem.

Proměnná `ignore` uchovává posloupnost žádného nebo více pozičních argumentů, které nás nezajímají. V případě metod vyvolaných v důsledku volby položky z nabídky nebo stisku tlačítka nástrojového panelu žádné argumenty neignorujeme. Když se však použije klávesová zkratka (např. Ctrl+N), obdržíme zdrojovou událost. Nás ale nezajímá, jak uživatel danou akci vyvolal, a proto tuto událost ignorujeme.

```
def okayToContinue(self):
    if not self.dirty:
        return True
    reply = tkinter.messagebox.askyesnocancel(
                    "Záložky - Neuložené změny",
                    "Uložit změny?", parent=self.parent)
    if reply is None:
        return False
    if reply:
        return self.fileSave()
    return True
```

Chce-li uživatel provést akci, která vymaže seznam (např. vytvořením či otevřením nového souboru), musíme mu dát šanci na uložení jakýchkoli neuložených změn. Pokud stávající seznam nebyl změněn, pak není co ukládat, a proto ihned vrátíme hodnotu `True`. V opačném případě zobrazíme standardní dialogové okno s tlačítky Ano, Ne a Storno. Pokud uživatel operaci stornuje, odpovědí je hodnota `None`. Znamená to tedy, že nechce pokračovat v akci, kterou zahájil, a proto jen vrátíme hodnotu `False`. Pokud se uživatel vyjádří kladně, odpovědí je hodnota `True`, a proto mu dáme šanci

soubor uložit vrátíme hodnotu `True`, pokud soubor uložil, nebo `False` v opačném případě. A pokud se uživatel vyjádří záporně, odpovědí je hodnota `False`, která říká, že se uložení nemá provést. My ale přesto vrátíme hodnotu `True`, protože uživatel chce pokračovat v akci, kterou započal, a zahodit neuložené změny.

Standardní dialogová okna knihovny Tk se příkazem import `tkinter` neimportují, takže musíme navíc provést příkaz import `tkinter.messagebox` a pro následující metodu i příkaz `import tkinter.filedialog`. V systémech Windows a Mac OS X se použijí standardní nativní dialogová okna, zatímco na ostatních platformách se použijí dialogová okna specifická pro knihovnu Tk. Standardním dialogovým oknům vždy předáváme rodičovský objekt, což zajistí, že se při svém zobrazení automaticky vycentrují nad rodičovským oknem.

Všechna standardní dialogová okna jsou *modální*. To znamená, že jakmile se jedno z nich zobrazí, stane se jediným oknem v programu, s nímž může uživatel pracovat. Před interakcí se zbytkem programu jej tedy musí nejdříve zavřít (klepnutím na tlačítko OK, Otevřít, Storno nebo na podobné tlačítko). S modálními dialogovými okny se programátorům pracuje nejsnadněji, protože uživatel nemůže změnit stav programu za zády dialogového okna, které navíc blokuje, dokud není uzavřeno. Blokování znamená, že při vytvoření nebo vyvolání modálního dialogového okna se následující příkaz provede až po uzavření tohoto dialogového okna.

```
    def fileSave(self, *ignore):
        if self.filename is None:
            filename = tkinter.filedialog.asksaveasfilename(
                           title="Záložky - Uložit soubor",
                           initialdir=".",
                           filetypes=[("Soubory se záložkami", "*.bmf")],
                           defaultextension=".bmf",
                           parent=self.parent)
            if not filename:
                return False
            self.filename = filename
            if not self.filename.endswith(".bmf"):
                self.filename += ".bmf"
        try:
            with open(self.filename, "wb") as fh:
                pickle.dump(self.data, fh, pickle.HIGHEST_PROTOCOL)
            self.dirty = False
            self.setStatusBar("Uloženo {0} prvků do souboru {1}".format(
                              len(self.data), self.filename))
            self.parent.title("Záložky - {0}".format(
                              os.path.basename(self.filename)))
        except (EnvironmentError, pickle.PickleError) as err:
            tkinter.messagebox.showwarning("Záložky - Chyba",
                    "Nemohu uložit soubor {0}:\n{1}".format(
                    self.filename, err), parent=self.parent)
        return True
```

Pokud žádný stávající soubor neexistuje, musíme požádat uživatele o zadání názvu souboru. Pokud zadání stornuje, vrátíme hodnotu `False`, která říká, že celá operace by měla být stornována. V opačném případě se postaráme, aby měl zadaný název souboru správnou příponu. Pomocí stávajícího nebo nového názvu souboru zapíšeme naložený (pickled) slovník do souboru. Po uložení záložek obnovíme příznak změny, protože žádné neuložené změny již nemáme, a do stavového panelu umístíme zprávu (jejíž zobrazení po určité době vyprší, jak uvidíme za okamžik) a aktualizujeme záhlaví okna, aby obsahovalo název souboru (bez cesty). Pokud soubor nemohl být uložen, zobrazíme dialogové okno s varováním (které bude automaticky obsahovat tlačítko OK), které o vzniklé situaci informuje uživatele.

```
def setStatusBar(self, text, timeout=5000):
    self.statusbar["text"] = text
    if timeout:
        self.statusbar.after(timeout, self.clearStatusBar)
```

Tato metoda nastavuje text popisku stavového panelu. Je-li použit parametr `timeout` (s výchozí hodnotou pět vteřin), pak vytvoří neopakující se časovač, který po uplynutí zadané doby vyčistí stavový panel.

```
def fileOpen(self, *ignore):
    if not self.okayToContinue():
        return
    dir = (os.path.dirname(self.filename)
           if self.filename is not None else ".")
    filename = tkinter.filedialog.askopenfilename(
                    title="Záložky - Otevřít soubor",
                    initialdir=dir,
                    filetypes=[("Soubory se záložkami", "*.bmf")],
                    defaultextension=".bmf", parent=self.parent)
    if filename:
        self.loadFile(filename)
```

Tato metoda na začátku dává stejně jako metoda `MainWindow.fileNew()` uživateli šanci uložit jakékoli neuložené změny nebo akci otevření souboru stornovat. Pokud se uživatel rozhodne pokračovat, chceme mu nabídnout rozumný výchozí adresář. Proto použijeme adresář aktuálního souboru, pokud byl již dříve nějaký otevřen, nebo v opačném případě aktuální pracovní adresář. Argument `filetypes` je seznam n-tic se dvěma prvky (popis, zástupný symbol), který by mělo dialogové okno zobrazit. Pokud uživatel vybere název souboru, uložíme jej do aktuálního názvu souboru a zavoláme metodu `loadFile()`, která provede vlastní načtení souboru.

Vyčlenění metody `loadFile()` je běžnou praxí s cílem usnadnit načítání souboru bez interakce s uživatelem. Kupříkladu některé programy načítají při spouštění naposledy používaný soubor anebo mají v nabídce seznam naposledy používaných souborů, takže pokud si uživatel některý z nich vybere, zavolá se přímo metoda `loadFile()` se zvoleným názvem souboru.

```
def loadFile(self, filename):
    self.filename = filename
```

```
        self.listBox.delete(0, tkinter.END)
        self.dirty = False
        try:
            with open(self.filename, "rb") as fh:
                self.data = pickle.load(fh)
            for name in sorted(self.data, key=str.lower):
                self.listBox.insert(tkinter.END, name)
            self.setStatusBar("Načteno {0} záložek ze souboru {1}".format(
                              self.listBox.size(), self.filename))
            self.parent.title("Záložky - {0}".format(
                              os.path.basename(self.filename)))
        except (EnvironmentError, pickle.PickleError) as err:
            tkinter.messagebox.showwarning("Záložky - Chyba",
                    "Nemohu načíst soubor {0}:\n{1}".format(
                    self.filename, err), parent=self.parent)
```

Při zavolání této metody víme, že jakákoli neuložená data byla uložena nebo zahozena, takže můžeme klidně vymazat seznam. Zadaný název souboru uložíme jako aktuální, vyčistíme seznam a příznak změny a poté se pokusíme otevřít soubor a vytáhnout z něj slovník `self.data`. Jakmile máme data, projdeme všechny názvy záložek a připojíme každou k seznamu. Nakonec ve stavovém panelu zobrazíme informativní zprávu a aktualizujeme záhlaví okna. Pokud se soubor nepodařilo přečíst nebo pokud se z něj nepodařilo vytáhnout slovník, zobrazíme dialogové okno s varováním, které patřičným způsobem informuje uživatele.

```
    def fileQuit(self, event=None):
        if self.okayToContinue():
            self.parent.destroy()
```

Toto je poslední metoda pro položky z nabídky „Soubor“. Uživatel dostane šanci uložen jakékoli neuložené změny. Pokud operaci zruší, nic se nestane a program pokračuje dál. V opačném případě oznámíme rodičovskému objektu, aby se zničil, což povede k čistému ukončení programu. Pokud bychom chtěli uložit preference uživatele, učinili bychom tak v této metodě, těsně před zavoláním metody `destroy()`.

```
    def editAdd(self, *ignore):
        form = AddEditForm(self.parent)
        if form.accepted and form.name:
            self.data[form.name] = form.url
            self.listBox.delete(0, tkinter.END)
            for name in sorted(self.data, key=str.lower):
                self.listBox.insert(tkinter.END, name)
            self.dirty = True
```

Pokud uživatel požádá o přidání nové záložky (zvolením příkazu Úpravy > Přidat, klepnutím na příslušné tlačítko v panelu nástrojů nebo stiskem klávesové zkratky Ctrl+P), zavolá se tato metoda. Třída `AddEditForm` představuje naše vlastní dialogové okno, kterému se budeme věnovat v následujícím oddílu. Nyní stačí, když víme, že má příznak `accepted`, jehož hodnota je `True`, pokud uživatel

klepl na tlačítko OK, nebo `False`, pokud uživatel klepl na tlačítko Storno, a dva datové atributy `name` a `url`, které uchovávají název a adresu URL záložky, kterou uživatel přidal nebo upravil.

Vytvoříme nový formulář typu `AddEditForm`, který se okamžitě zobrazí jako modální dialogové okno, a proto blokuje, takže příkaz `if form.accepted ...` se neprovede, dokud se toto dialogové okno nezavře.

Pokud uživatel v dialogovém okně `AddEditoForm` zadá název záložky a klepne na tlačítko OK, přidáme název a adresu URL nové záložky do slovníku `self.data`. Potom vyčistíme seznam a opětovně vložíme všechna data v seřazeném pořadí. Mnohem efektivnější by bylo prostě vložit novou záložku na správné místo, ale i se stovkou záložek by byl rozdíl na moderním stroji stěží postřehnutelný. Na konci nastavujeme příznak změny záložek, které nyní obsahují neuloženou změnu.

```
def editEdit(self, *ignore):
    indexes = self.listBox.curselection()
    if not indexes or len(indexes) > 1:
        return
    index = indexes[0]
    name = self.listBox.get(index)
    form = AddEditForm(self.parent, name, self.data[name])
    if form.accepted and form.name:
        self.data[form.name] = form.url
        if form.name != name:
            del self.data[name]
            self.listBox.delete(0, tkinter.END)
            for name in sorted(self.data, key=str.lower):
                self.listBox.insert(tkinter.END, name)
        self.dirty = True
```

Úprava je malinko složitější než přidání záložky, protože nejdříve musíme záložku, kterou chce uživatel upravit, najít. Metoda `curselection()` vrací (případně prázdný) seznam indexových pozic pro všechny své vybrané prvky. Je-li vybrán právě jeden prvek, pak získáme jeho text, který je zároveň názvem záložky, kterou chce uživatel upravit (a také klíčem do slovníku `self.data`). Potom vytvoříme nový formulář typu `AddEditForm`, kterému předáme název a adresu URL záložky, kterou chce uživatel upravit.

Pokud uživatel uzavřel formulář tlačítkem OK a nenechal název záložky prázdný, aktualizujeme slovník `self.data`. Je-li nový i starý název stejný, pak stačí jen nastavit příznak změny záložek (v tomto případě předpokládáme, že uživatel změnil adresu URL). Pokud se ale název záložky změnil, vymažeme prvek slovníku, jehož klíč odpovídá původnímu názvu, vyčistíme seznam a poté jej znovu naplníme záložkami stejně jako při po přidání záložky.

```
def editDelete(self, *ignore):
    indexes = self.listBox.curselection()
    if not indexes or len(indexes) > 1:
        return
    index = indexes[0]
```

```
        name = self.listBox.get(index)
        if tkinter.messagebox.askyesno("Záložky - Vymazání",
                                       "Vymazat '{0}'?".format(name)):
            self.listBox.delete(index)
            self.listBox.focus_set()
            del self.data[name]
            self.dirty = True
```

Pro vymazání záložky musíme nejdříve zjistit, kterou záložku uživatel vybral, takže tato metoda začíná stejnými řádky jako metoda `MainWindow.editEdit()`. Je-li zvolena právě jedna záložka, zobrazíme dialogové okno s dotazem, zda se má skutečně vymazat. Pokud uživatel odpoví kladně, dialogové okno vrátí hodnotu `True`, vymažeme zvolenou záložku ze seznamu a ze slovníku `self.data` a nastavíme příznak změny záložek. Kromě toho nastavíme zaměření klávesnice zpět na seznam.

```
    def editShowWebPage(self, *ignore):
        indexes = self.listBox.curselection()
        if not indexes or len(indexes) > 1:
            return
        index = indexes[0]
        url = self.data[self.listBox.get(index)]
        webbrowser.open_new_tab(url)
```

Jakmile uživatel vyvolá tuto metodu, vyhledáme zvolenou záložku a ze slovníku `self.data` získáme odpovídající adresu URL. Potom pomocí funkce `webbrowser.open_new_tab()` z modulu `webbrowser` otevřeme webový prohlížeč uživatele na dané adrese URL. Pokud webový prohlížeč neběží, tímto voláním se spustí.

```
application = tkinter.Tk()
path = os.path.join(os.path.dirname(__file__), "images/")
if sys.platform.startswith("win"):
    icon = path + "bookmark.ico"
    application.iconbitmap(icon, default=icon)
else:
    application.iconbitmap("@" + path + "bookmark.xbm")
window = MainWindow(application)
application.protocol("WM_DELETE_WINDOW", window.fileQuit)
application.mainloop()
```

Poslední řádky programu jsou podobné těm, které jsme použili v dříve uvedeném programu `interest-tk.pyw`, ovšem se třemi odlišnostmi. Jedna z nich spočívá v tom, že pokud uživatel klepne na tlačítko pro uzavření okna programu, zavolá se v programu Záložky jiná metoda než ta, kterou jsme použili v programu Úrok. Další odlišností je, že ve Windows má metoda `iconbitmap()` dodatečný argument, který umožňuje zadat výchozí ikonku pro všechna okna programu, což není na unixových platformách potřeba, protože se to děje automaticky. A poslední odlišností je to, že název aplikace (v záhlaví) nenastavujeme na tomto místě, ale v metodě třídy `MainWindow`. V případě programu Úrok se název programu nikdy nezměnil, takže jej stačilo nastavit pouze jednou, ale v programu Záložky měníme název v záhlaví okna aplikace podle názvu soubor se záložkami, s nímž se právě pracuje.

Rozebrali jsme si implementaci třídy hlavního okna a kód, který inicializuje program a spouští cyklus událostí, a proto se nyní můžeme zaměřit na dialogové okno `AddEditForm`.

Vytvoření vlastního dialogového okna

Dialogové okno `AddEditForm` poskytuje prostředky, které umožňují uživatelům přidávat a upravovat názvy a adresy URL záložek. Je zachyceno na obrázku 15.6, kde se používá pro úpravu stávající záložky (proto je v záhlaví uvedeno „Úprava“). Stejné dialogové okno lze použít také pro přidání nové záložky. Začneme pohledem na inicializační metodu dialogového okna, kterou si rozdělíme na čtyři části.

Obrázek 15.6: Dialogové okno programu Záložky pro přidání a úpravu záložky

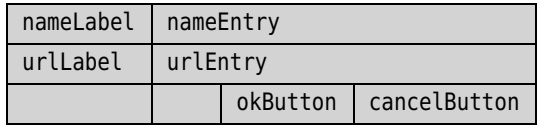
```
class AddEditForm(tkinter.Toplevel):

    def __init__(self, parent, name=None, url=None):
        super().__init__(parent)
        self.parent = parent
        self.accepted = False
        self.transient(self.parent)
        self.title("Záložky - " + (
                   "Úprava" if name is not None else "Přidání"))

        self.nameVar = tkinter.StringVar()
        if name is not None:
            self.nameVar.set(name)
        self.urlVar = tkinter.StringVar()
        self.urlVar.set(url if url is not None else "http://")
```

Naši třídu jsme odvodili od třídy `tkinter.TopLevel`, což je holý ovládací prvek navržený jako bázová třída pro ovládací prvky představující okna na nejvyšší úrovni. Odkaz na rodičovské okno si ukládáme a vytváříme atribut `self.accepted`, který nastavujeme na hodnotu `False`. Zavoláním metody `transient()` informujeme rodičovské okno, že nad ním musí být vždy umístěno toto okno. Název okna nastavujeme podle toho, zda byl předán název a adresa URL záložky. Vytváříme dva objekty typu `tkinter.StringVar`, které budou uchovávat název a adresu URL záložky a které v případě, že bylo dialogové okno použito pro úpravu, inicializujeme předanými hodnotami.

```
        frame = tkinter.Frame(self)
        nameLabel = tkinter.Label(frame, text="Název:", underline=0)
        nameEntry = tkinter.Entry(frame, textvariable=self.nameVar)
        nameEntry.focus_set()
        urlLabel = tkinter.Label(frame, text="Adresa URL:", underline=0)
```

```
urlEntry = tkinter.Entry(frame, textvariable=self.urlVar)
okButton = tkinter.Button(frame, text="OK", command=self.ok)
cancelButton = tkinter.Button(frame, text="Storno",
                              command=self.close)

nameLabel.grid(row=0, column=0, sticky=tkinter.W, pady=3,
               padx=3)
nameEntry.grid(row=0, column=1, columnspan=3,
               sticky=tkinter.EW, pady=3, padx=3)
urlLabel.grid(row=1, column=0, sticky=tkinter.W, pady=3,
              padx=3)
urlEntry.grid(row=1, column=1, columnspan=3,
              sticky=tkinter.EW, pady=3, padx=3)
okButton.grid(row=2, column=2, sticky=tkinter.EW, pady=3,
              padx=3)
cancelButton.grid(row=2, column=3, sticky=tkinter.EW, pady=3,
                  padx=3)
```

Vytváříme ovládací prvky a umisťujeme je do mřížky, což znázorňuje obrázek 15.7. Ovládací prvky pro zadání textu pro název a adresu URL jsou spojeny s odpovídajícími objekty typu `tkinter.StringVar` a dvě tlačítka jsou nastavena na volání metod `self.ok()` a `self.close()`, které si ukážeme později.



Obrázek 15.7: Rozvržení ovládacích prvků v dialogovém okně programu Záložky pro přidání a úpravu záložky

```
frame.grid(row=0, column=0, sticky=tkinter.NSEW)
frame.columnconfigure(1, weight=1)
window = self.winfo_toplevel()
window.columnconfigure(0, weight=1)
```

U tohoto dialogového okna je smysluplné povolit změnu velikosti pouze ve vodorovném směru, a proto nastavíme váhu druhého sloupce v rámu okna na 1, což znamená, že při roztažení rámu ve vodorovném směru bude velikost ovládacích prvků ve sloupci 1 (pole pro zadání názvu a adresy URL) růst tak, aby zabrala celý dostupný prostor. Podobným způsobem povolíme změnu velikosti ve vodorovném směru i u okna (nastavením jeho váhy na 1). Pokud uživatel změní výšku dialogového okna, ovládací prvky si udrží své relativní pozice a všechny budou vycentrované na střed okna. Pokud ale uživatel změní šířku dialogového okna, změní ovládací prvky pro zadání názvu a adresy URL svoji šířku tak, aby zaplnily dostupný prostor ve vodorovném směru.

```
self.bind("<Alt-n>", lambda *ignore: nameEntry.focus_set())
self.bind("<Alt-a>", lambda *ignore: urlEntry.focus_set())
self.bind("<Return>", self.ok)
self.bind("<Escape>", self.close)

self.protocol("WM_DELETE_WINDOW", self.close)
```

```
        self.grab_set()
        self.wait_window(self)
```

Vytvořili jsme dva popisky „Název:“ a „Adresa URL:“, z čehož je patrné, že jejich klávesovými akcelerátory jsou Alt+N a Alt+A, po jejichž stisku dojde k přesunu zaměření klávesnice na odpovídající textové pole. Aby toto fungovalo, musíme specifikovat příslušné klávesové vazby. Metody `focus_set()` nepředáváme přímo, ale používáme lambda funkce, díky nimž můžeme ignorovat argument s informacemi o události. Dále jsme definovali standardní klávesové vazby (Enter a Esc) pro tlačítka OK a Storno.

Pomocí metody `protocol()` stanovíme metodu, která se má zavolat, pokud uživatel dialogové okno uzavře klepnutím na uzavírací tlačítko. Volání metod `grab_set()` a `wait_window()` je nezbytné pro převedení tohoto okna na modální dialogové okno.

```
    def ok(self, event=None):
        self.name = self.nameVar.get()
        self.url = self.urlVar.get()
        self.accepted = True
        self.close()
```

Pokud uživatel klepne na tlačítko OK (nebo pokud stiskne klávesu Enter), zavolá se tato metoda. Texty z objektů typu `tkinter.StringVar` kopírujeme do odpovídajících proměnných instance (které se vytvořily až nyní), proměnnou `self.accepted` nastavujeme na hodnotu `True` a zavoláním metody `self.close()` dialogové okno uzavřeme.

```
    def close(self, event=None):
        self.parent.focus_set()
        self.destroy()
```

Tato metoda se volá z metody `self.ok()` nebo pokud uživatel klepne na tlačítko pro uzavření okna nebo na tlačítko Storno (nebo při stisku klávesy Esc). Zaměření klávesnice se přenese zpět na rodičovské okno a dialogové okno se zničí. Zničení v tomto kontextu znamená, že se zničí pouze okno a jeho ovládací prvky. Samotná instance třídy `AddEditForm` bude existovat i nadále, protože se na ni odkazuje volající kód.

Po uzavření dialogového okna volající zkontroluje proměnnou `accepted`. Má-li hodnotu `True`, pak získá zadaný název a adresu URL. Poté, po dokončení metody `MainWindow.editAdd()` nebo `MainWindow.editEdit()`, opustí objekt typu `AddEditForm` obor platnosti a je naplánován na úklid z paměti.

Shrnutí

V této lekci jste mohli okusit něco z programování grafického uživatelského rozhraní pomocí knihovny GUI Tk. Velkou výhodou knihovny Tk je, že se standardně dodává s Pythonem. Na druhou stranu má ale spoustu nevýhod, přičemž její poněkud zastaralejší způsob fungování lišící se od většiny modernějších alternativ rozhodně nepatří mezi ty nejmenší.

Je-li pro vás programování grafického uživatelského rozhraní nové, pak mějte na paměti, že přední konkurenti knihovny Tk schopní pracovat na více platformách (PyGtk, PyQt a wxPython) se osvojují a používají mnohem snadněji a lze s nimi docílit lepších výsledků s použitím menšího množství kódu. Ba co víc, všichni tito konkurenti knihovny Tk mají lepší dokumentaci věnovanou Pythonu, více ovládacích prvků a lepší vzhled i chování. Navíc umožňují vytvářet ovládací prvky úplně od začátku s naprostou kontrolou nad jejich vzhledem a chováním.

Přestože je knihovna Tk užitečná pro tvorbu velmi malých programů nebo pro situace, kdy je dostupná pouze standardní knihovna Pythonu, bývá ve všech ostatních případech mnohem lepší volbou kterákoli z ostatních multiplatformních knihoven.

Cvičení

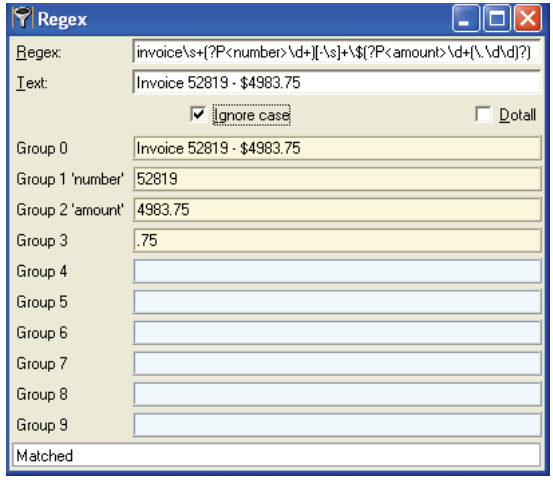
V prvním cvičení budete kopírovat a upravovat program Záložky, který jsme vytvořili v této lekci. Ve druhém cvičení vytvoříte program GUI úplně od začátku.

1. Zkopírujte program `bookmarks-tk.pyw` a upravte jej tak, aby uměl importovat a exportovat soubory DBM, které používá konzolový program `bookmarks.py` (vytvořený jako cvičení v lekci 12). Přidejte do nabídky „Soubor“ dvě nové položky nabídky „Import“ a „Export“. Nezapomeňte k oběma přidat také klávesové zkratky. Dále vytvořte dvě odpovídající tlačítka na panelu nástrojů. Celkem to znamená přidat asi pět řádků kódu do inicializační metody hlavního okna.

   Bude třeba vytvořit dvě metody `fileImport()` a `fileExport()` implementující nově přidanou funkčnost, které by neměly přesáhnout 60 řádků kódu včetně ošetření chyb. V případě importu se můžete rozhodnout, zda stávající záložky sloučit, nebo nahradit importovanými. Kód není obtížný, ale vyžaduje jistou pečlivost. Řešení (které importované záložky slučuje se stávajícími) je k dispozici v souboru `bookmarks-tk_ans.py`.

   Je třeba poznamenat, že zatímco na unixových systémech je přípona `.dbm` v pořádku, ve Windows představuje každý „soubor“ DBM ve skutečnosti tři soubory. Ve Windows by proto měl být v dialogových oknech pro práci se soubory vzor `*.dbm.dat` a výchozí přípona`*.dbm.dat`, přičemž skutečný název souboru by měl mít příponu `.dbm`, a proto je nutné poslední čtyři písmena v názvu souboru odříznout.

2. V lekci 13 jsme si ukázali, jak vytvářet a používat regulární výrazy pro hledání shody v textu. Vytvořte program GUI ve stylu dialogového okna, který lze použít pro zadávání a testování regulárních výrazů (viz obrázek 15.8).



Obrázek 15.8: Program Regulární výraz

Budete si muset pročíst dokumentaci k modulu re, protože program se musí chovat korektně i v případě neplatných regulárních výrazů nebo při procházení nalezených skupin, poněvadž ve většině případů nebude mít regulární výraz tolik skupin, kolik je k dispozici popisků pro jejich zobrazení. Nezapomeňte, aby měl program plnou podporu i pro uživatele pracující s klávesnicí. Pro přesun na pole pro zadání textu by měly sloužit kombinace kláves Alt+R a Alt+T, pro zaškrtávací políčka Alt+I a Alt+N a pro ukončení programu Ctrl+K a Esc. Kromě toho by při stisku a uvolnění klávesy v některém z textových polí nebo při aktivaci či deaktivaci zaškrtávacího pole mělo dojít k přepočítání výsledků.

Program není příliš těžký, i když kód pro zobrazení nalezených shod a čísel skupin (a názvů, jsou-li uvedeny) je trošku obtížnější. Řešení je k dispozici v souboru `regex-tk.pyw`, který má kolem 140 řádků.

Závěrem

Pokud jste přečetli alespoň prvních šest lekcí a buď dokončili cvičení, nebo nezávisle na nich napsali své vlastní programy v Pythonu 3, pak byste měli mít dobrou výchozí pozici pro budování svých zkušeností a programovacích dovedností až do té míry, do jaké budete chtít – Python vás rozhodně brzdit nebude!

Pokud jste přečetli pouze prvních šest lekcí a toužíte po zlepšení a prohloubení svých schopností v oblasti jazyka Python, pak se ujistěte, že znáte látku probíranou v lekci 7 a že jste prostudovali alespoň některá témata v lekci 8 a experimentovali s nimi, zvláště s příkazem `with` a se správci kontextu. Kromě toho stojí za přečtení alespoň část lekce 9 věnovaná testování.

Nicméně mějte na paměti, že vývoj všeho od samého začátku je v Pythonu kromě potěšení a možnosti naučit se nové věci jen zřídka nezbytný. Zmínili jsme se o standardní knihovně a o seznamu balíčků pro jazyk Python (*pypi.python.org/pypi*), které poskytují obrovské množství funkčních prvků. Na adrese *code.activestate.com/recipes/langs/python/* jsou dále k dispozici recepty pro jazyk Python nabízející rozsáhlou sbírku triků, tipů a nápadů, i když v době psaní této knihy je většina z nich určena pro Python 2.

Moduly pro Python lze vytvářet i v jiných jazycích (v kterémkoli jazyku, který dokáže exportovat funkce ve stylu jazyka C, což umí většina programovacích jazyků). Pomocí aplikačního rozhraní Pythonu pro jazyk C je možné je vyvinout tak, aby spolupracovaly s Pythonem. Z Pythonu lze pomocí modulu `ctypes` přistupovat ke sdíleným knihovnám (ve Windows se jedná o soubory DLL), ať už vytvořeným námi nebo získaným od třetí strany, čímž díky dovednosti a velkorysosti programátorů softwaru s otevřeným zdrojovým kódem získáme prakticky neomezený přístup k ohromnému množství funkcí dostupných přes Internet.

A pokud se chcete zapojit do komunity kolem Pythonu, pak je nejlepší začít na stránce *www.python.org/community*, kde najdete stránky Wiki a spoustu obecných i specifických dizkusních skupin.

Rejstřík

C

Č

D

G

H

CH

I

J

K

L

M

N

Q

R

Ř

S