Python se představuje

Instalace a spouštění

Pokud nemáte Python nainstalovaný, pokud nevíte, jak se vůbec Python může spouštět, měli byste si nejdříve přečíst Spouštíme Python.

První kroky

Pusťte IDLE nebo Command line. Máte před sebou okno, které má někde nahoře slovo “Python” a ve kterém bude něco jako:

```
>>>
```

To `>>>` se nazývá prompt (výzva), což znamená, že počítač je připraven přijímat vaše příkazy. Zde můžete psát různé věci a počítač je bude vykonávat.

Bohužel, počítač nerozumí česky (ani anglicky). Když napíšete:

```
>>> Řekni mi kolik je dvanáct plus třináct.
```

nebude vůbec rozumět tomu, co po něm chcete. Protože jsou počítače velmi hloupí, musíte k nim mluvit speciálním jazykem vyvinutým k tomu, aby mu počítač snadno rozuměl. Ve skutečnosti je mnoho jazyků vyvinutých k tomu, aby počítače rozuměly. My nyní zkoušíme ten, co se jmenuje Python. Na Pythonu je dobrá věc ta, že mu rozumí snadno i člověk.

Tady je, jak se počítače zeptat, kolik je dvanáct a třináct. Vyzkoušejte. (Nemusíte psát to >>>, ale musíte stisknout klávesu `Enter` na konci řádku):

```
>>> 12 + 13
```

Pár dalších příkladů i s odpověďmi od počítače. Ten # a vše co je za tím psát nemusíte, to je tzv. komentář a je jen pro vás, jen pro komentování kódu. Pro Python to neznamená vůbec nic, nemá to žádnou funkci, python komentáře zvesela ignoruje:

```
>>> 1 + 2 + 3 + 4
10
>>> 1 + 2 * 3 - 4        # násobení je *, ne x.
3                    # Pokud myslíte že to má být 5, přemýšlejte!
>>> 200 * 300
60000
>>> 12 / 4          # Pro dělení používejte /.
3.0
>>> 7/3
2.3333333333333335
```

V dřívějších verzích příklad 7/3 dával výsledek 2, protože se používalo celočíselného dělení.

V Pythonu můžete používat závorky stejně jako v matematice:

```
>>> (1 + 2) * (3 + 4)
21
```

Python zde nejdříve spočítal (1 + 2) a (3 + 4) (dostal 3 a 7), a pak výsledky mezi sebou vynásobil.

Nebojte se experimentovat! Kdykoliv se naučíte něco nového v Pythonu, zkuste dělat malé (nebo větší!) změny a hrajte si s tím, dokud si nebudete jistí, jak to funguje. Neomezujte se na to, co je napsáno na těchto listech!

Pokud vám náhodou stále není jasné proč 1+2*3-4 dává 3 a ne 5, tak ten důvod je ten, že násobení má přednost před sčítáním. Váš učitel matematiky tomu asi říká přednost funkcí nebo podobně. Pokud to stále není jasné, nic si z toho nedělejte, snad to nebudete potřebovat.

Různé typy objektů

Všechny věci, se kterými jsme doposud pracovali, byly čísla. Ale Python umí pracovat kromě čísel i se spoustou dalších věcí. Zkuste například toto:

```
>>> "ahoj " + "svete"
"ahoj svete"
```

Věci mezi uvozovkami se nazývají řetězce. Jak vás asi již napadlo z našeho příkladu, + můžeme používat i u řetězců. Spojuje řetězce; to znamená připojuje jeden hned za ten druhý. Následuje trochu překvapení:

```
>>> 3 * "ahoj"
```

Všimli jste si snad, že jsem vám tentokrát neřekl, co počítač odpověděl. A to proto, že od vás očekávám, že si to zkusíte sami. Nic se nenaučíte, pokud si ty příklady nebude zkoušet sami, pokud je budete jen číst a myslet si - tak tomu rozumím, a tomu taky, tak jedeme dál,.... Napište si do svých poznámek, co počítač odpověděl, když se ho zeptáte , kolik je `3 * "ahoj"`, abyste si to zapamatovali.

Uvozovky můžete používat jednoduché nebo dvojité, Pythonu na tom nezáleží:

```
>>> 'ham' + "burger"
'hamburger'
```

Python má taky seznamy:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> [1, 2, 3] + [7, 8]
```

Opět vám neříkám, jak Python odpoví. Zkuste a odpověď si zapište do svých poznámek.

Dáváme věcem jména

Představte si, že máte udělat mnoho výpočtů s číslem 123456. (Možná je to váš měsíční příjem :-)). Dá se to udělat takto:

```
>>> 123456 * 3
370368
>>> 123456 / 6
20576.0
>>> 123456 - 1000
122456
```

Určitě vás to za chvíli přestane bavit, protože koho by bavilo psát pořád dokola tak dlouhé číslo. A navíc - když bude chtít někdo jiný po vás číst, co jste dělali, bude asi zmaten tajemným číslem 123456 a proč je tam tak často. A ještě - také se můžete snadno splést a místo 4 napsat 3.

Všechny tyto problémy můžeme řešit tak, že dáme číslu jméno. Že ho prostě pojmenujeme. Dáme mu tedy jméno, které bude dobře vystihovat smysl toho čísla. Pro náš případ bude nejlepší `plat`. Uděláme to takto:

```
>>> plat = 123456
>>> plat * 4
493824
>>> plat / 12
10288.0
>>> plat
123456
```

Vtip je v tom, že jakmile jednou řeknete `plat=123456`, můžete kdykoliv použít `plat` místo `123456`. Čemu my zde říkáme jména, většina lidí říká proměnné. Později si řekneme proč. Prozatím jsou jména v pohodě. Jména můžete dávat i jiným věcem, než jsou čísla. Například:

```
>>> MojeJmeno = 'Pavel'
>>> 'Ahoj ' + MojeJmeno + '!'
'Ahoj Pavel!'
```

Tisk

V Pythonu říkáme tisk, když chceme něco vytisknout, zobrazit na obrazovce, ne na tiskárně. Doposud se vše tisklo jaksi automaticky:

```
>>> 1+2
3
```

3 je vytištěná automaticky. Někdy však toto jednoduché nefunguje a my musíme použít skutečný tisk, tedy funkci print():

```
>>> cena=1000
>>> print (cena)
1000
```

```
>>> print ("Cena vyrobku byla", cena, "Kc.")
Cena vyrobku byla 1000 Kc.
```

Příkaz print() se používá, když chcete přimět počítač, aby něco vytisknul. Dosud jsme to nepotřebovali, protože Python tiskne téměř vše základní v příkazovém řádku automaticky. Pokud chceme tisknout několik věcí za sebou, tak již print() použít musíme, a jednotlivé věci musíme oddělit čárkou. Stejně jako v programech musíme print k tisku používat vždy. Závorky tam musí být, protože print je funkce, ale o tom více až později.

Děláme něco znovu a znovu

Vaše kalkulačka jistě umí stejně dobře vše, co jsme až doposud dělali. Teď si ukážeme, co už asi neumí. Mimochodem, mezery na začátku druhého řádku jsou důležité! (Je to lépe vysvětleno v Na mezerách záleží.):

```
>>> seznam = [1, 2, 3, 4, 5]
>>> for x in seznam:
...     print (x, 3 * x)  # Prompt se změnil, Python vám naznačuje že ještě něco chce.
...                       # Zde stiskněte jen Enter.
```

Dovedete odhadnout co to bude dělat? ... Blahopřeji, pokud jste si mysleli, že to vytiskne čísla od 1 do 5 včetně svých trojnásobků. Všimnete si, že Python obyčejně vkládá mezeru mezi dvě věci, které tiskne za sebou v řádku.

Grafika

Python se dá používat ke kreslení obrázků. Vyzkoušejte toto. První tři řádky vypadají asi tajemně (vysvětlíme si je později), ty další snad již trochu smyslu dávají. Na tento příklad je lépe si pustit “Python (command line)” než “IDLE (Python GUI)” a zkoušet to v něm, protože výsledky v něm vidíte ihned, narozdíl od IDLE, kde výsledek uvidíte až po poslední řádce mainloop():

```
>>> from tkinter import *
>>> platno=Canvas()
>>> platno.pack()
>>> platno.create_line(0, 0, 200, 100)
>>> platno.create_line(0, 100, 200, 0, fill="red", dash=(4, 4))
>>> platno.create_rectangle(50, 25, 150, 75, fill="blue")
>>> mainloop()
```

Všechno se to příšerně kazí

Jednou se vám určitě stane (tedy pokud se vám to již nestalo), že Python odpovídá pěkně neslušně, třebas jako:

```
 3 + 'mismas'
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

To vypadá pěkně odpudivě. Je to většinou napsané červeně a na posledním řádku se vždy vyskytuje slovo Error. Neděste se. To jen Python dává najevo, že nerozumí tomu, co jste napsali. Všechno kromě poslední řádky snad můžete ze začátku ignorovat. Pokud chcete, podívejte se na Chyby, kde se to vysvětluje podrobněji, kde je seznam nejběžnějších stížností, které na vás Python může mít a co znamenají.

První program

Na konci tohoto listu si vytvoříme první program. Někdy budu používat slovo program, někdy script - je to to samé. V IDLE zvolte File-New Window, tím se vám otevře nové editační okno, ve kterém již počítač okamžitě neodpovídá na příkazy, ale můžete v něm napsat několik těchto příkazů za sebou (1 řádek = 1 příkaz). Poté, co program napíšete a uložíte (File-Save), můžete ho spustit dvojím způsobem. Buď pomocí nabídky Run-Run (případně klávesou F5) nebo v Tento počítač dvojklikem (případně Enterem). Více o tomto v IDLE editor

Upozorňuji také, že zde nesmíte psát ono >>>, stejně jako to nepíšete na příkazovém řádku. Na příkazovém řádku se však >>> zobrazují, v editoru při psaní programů však již ne. Takže např. první kod z této stránky bude napsán jako program takto:

```
print(1 + 2 + 3 + 4)
print(1 + 2 * 3 - 4)
print(200*300)
print(12/4)
print(7/3)
```

Úkol: Od dávných dob se traduje, že první program nedělá nic jiného, než že tiskne Ahoj svete!. Nic víc, nic míň. Takže do toho! Čeština v řetezcích: Doporučuji, alespon zpočátku, české háčky a čárky nepoužívat, nebo pak raději přejít na jiný editor, než je IDLE. Více na Čeština v programech.

Co dál?

Máme dva druhy listů:

1. Pracovní listy, každý z nich vás provede napsáním snad zajímavého programu.
2. Pomocné listy, každý z nich vám řekne něco zajímavého o Pythonu.

Potom vám doporučujeme pokračovat popořadě všemi pracovními listy. Každý z nich se odkazuje na pomocné listy, takže ty si můžete číst v průběhu.

Zkoušení z násobilky

V minulé lekci byl hlavním naším prostředím Příkazový řádek. V této lekci naším hlavním prostředím bude IDLE - nabídka File - New Window, tedy editor, ve kterém budeme psát svůj první program. Pokud budu něco ukazovat na Příkazovém řádku, tak jen jako nápomoc. Vy svoje dílo vytvářejte v editoru jako program. To byste mohli udělat hned: Otevře si IDLE - nabídku File - New Window. A hned si to snad uložte, jako např. nasobilka.py. V editoru, v programu, se již znaky >>> neobjevují, ani se tam nesmí psát. Hned za chvíli, tam začnete vytvářet svůj první program.

Úvod

V jedné věci - počtech - jsou počítače velmi, velmi dobří. Tento list vám ukáže, jak říci počítači, aby vyzkoušel Vás z násobilky. Toto není vůbec jednoduchá lekce. Postupujte pomalu, doporučuji si vše důkladně zkoušet. Lekce je velmi dlouhá, náročná, ale je rozdělena do menších, zvládnutelných úseků. Držíme palce!

Co potřebujete vědět

Pro dokončení tohoto listu potřebujete znát:

- Jak editovat a spustit pythonýrský program: viz IDLE editor
- Základy jazyka Python z Python se představuje
- Jednoduché počty: http://prirodni-vedy.tvujtest.cz/jine-prirodni-vedy/jednoduche-pocty ;-)
- Jak fungují české háčky a čárky v programech, pokud budete chtít, aby program mluvil česky: Čeština v programech

Rozmyslíme si to

Když začínáte přemýšlet o tom, že napíšete nějaký program, je užitečné se zamyslet nad tím, co se má stát, když se program spustí. Mělo by to být něco jako toto:

```
Kolik je 6 x 7? 49
Ne, lituji, správná odpověď je 42.
Kolik je 3 x 2? 6
Ano - to je dobře.

*A tak dále několik dalších otázek...*

Kolik je 5 x 9? 45
Ano - to je dobře.

Zeptal jsem se 10 x. 7 x jsi odpověděl správně.
Výborně!
```

Máme tu několik věcí, které musí program umět:

- Náhodně volit čísla
- Ty čísla vytisknout jako příklad
- Vypočítat správnou hodnotu (zatím netisknout)
- Zjistit odpověď od člověka, který program používá
- Zjistit, jestli je správná nebo ne
- Zobrazit zprávu “to je dobře” nebo “to je špatně”
- Sledovat, kolikrát jsme již odpovídali
- Zeptat se celkem řekněme 10 otázek a pak zastavit
- Zobrazit poslední zprávu o tom, jak jste si vedli

Náhodná čísla

Začneme s první položkou na seznamu: Náhodné generování čísel. Zkuste ve svém programu napsat následující řádky, a pusťte ho (F5).:

```
import random
print ("Nahodne cislo:", random.randint(10,15))
```

Co myslíte, že `randint` dělá? Zkuste opakovat poslední řádku několikrát, dokud si nebudete jistí, jak se chová.

Ta první řádka je důležitá. Musíte ji dát na začátek každého programu, který využívá `randint`. Umožňuje vám naimportovat neboli vtáhnout do vašeho programu dodatky k čistému Pythonu. Stejně jako to vkládáte do programu, měli byste to napsat i na promptu, pokud budete používat náhodná čísla v promptu. Jinak nebude Python rozumět, co po něm chcete.

Úkol: Vytvořte program, který bude tisknout:

```
Kolik je 7 x 2 ?
```

kde ty 2 čísla (7 i 2) jsou náhodně generovány mezi 1 a 10 (včetně).

Pamatujeme si čísla

Dobře, takže teď se začneme starat o to, jak si má program zapamatovat náhodné číslo. Dáme náhodnému číslu jméno, uložíme ho do proměnné! Jména jsme probírali v Python se představuje, tak jen krátkou připomínku. Věcem dáváme jména použitím znaku =, a pak můžete používat jméno místo samotné věci, kterou jste pojmenovali:

```
>>> neco=1234   # do proměnné neco zapis 1234
>>> 5*neco
6170
```

Do proměnné se dá uložit i náhodné číslo:

```
import random
cislo1= random.randint(10,15)
print (cislo1)
```

Úkol: Opravte svůj program, aby náhodná čísla se nejdříve uložila do proměnných a teprve potom je vytiskla včetně jejich součinu.

Otázky a odpovědi

Chceme, aby se program na chvíli zastavil, vytiskl otázku, a počkal, než uživatel něco odpoví (napíše) a zmáčkne Enter. Zadaná hodnota se uloží do nějaké proměnné. To se může udělat takto:

```
odpoved = input('Zadejte cislo: ')
odpoved = int(odpoved)
print ("Zadal jste cislo:", odpoved)
```

Příkaz input vrací řetězec, a to i když zadáme číslo. Musíme tedy (na druhé řádce) číslo - řetezec převést na obyčejné číslo, se kterým se dá počítat. To dělá právě funkce int(). Pro lepší pochopení se zkuste podívat na Vstup.

Úkol: Najděte si program, který jste napsali a opravte a doplňte tak, aby uměl dělat následující:

1. Zvolit dvě náhodná čísla od 1 do 10 s uložením do proměnných.
2. Zobrazit příklad na násobení.
3. Zeptat se na odpověď.
4. Vytisknout kolik to má být.

Dobře nebo špatně?

Program, který jste právě napsali, by se asi mohl používat na zkoušení někoho z násobilky, ale bude hloupé chtít po člověku, aby si sám kontroloval výsledky; to je právě ta nudná práce, kterou počítače zvládají dobře. Takže v dalším stupni necháme stroj prověřit, jestli odpověď, kterou od uživatele dostal, je správná, namísto toho, že jen napíšeme kolik to mělo být a nechali na uživateli ať si to zhodnotí sám.

Abychom toho dosáhli, potřebujeme novou - a velmi důležitou - myšlenku.

If...(Když...)

Napište nový program, který vypadá jako tento a spusťte ho. (Mezery na začátku některých řádek jsou důležité!):

```
odpoved=input('Zadej cislo: ')
odpoved = int(odpoved)
if odpoved < 10:
    print ('Cislo je mensi nez 10.!')
else:
    print ('Cislo je vetsi nebo rovno 10!')
```

Program dělá toto: Pokud (if) bude číslo menší než 10, Python provede jen ten první print, jinak (else), tedy pokud bude odpoved větší nebo rovno 10, provede jen ten druhý print.

Na mezerách záleží

Poznámka

Důvod, proč záleží na mezerách na začátku řádku je ten, že Python je používá při rozhodování, kolik toho z vašeho programu patří do if. Řekněme, že napíšete

```
odpoved=input('Zadej cislo: ')

odpoved = int(odpoved)

if odpoved > 1000:

        print ('Och!')

print ('Bink!')
```

Pokud zadáte číslo menší než 1000, podmínka odpoved>1000 nebude splněna, a proto se její tělo print ('Och!') neprovede (nevytiskne) a pokračuje se nejbližším příkazem za ním. Vytiskne se tedy jen Bink!, protože řádka print ('Bink!') nepatří do těla if. Ale když by řádka print ('Bink!') měla na začátku mezery, jako má řádka print ('Och!') , tak by to bylo součástí těla if, a nevytisklo by se vůbec nic. Je to velmi důležitá myšlenka, proto si s ní větší než malou chvíli pohrajte, abyste ji porozuměli.

Příkaz bychom mohli číst také takto: Pokud je odpoved větší než 1000, tak vytiskni ‘Och’ a pokračuj pak za vnitřkem (tělem) if, tedy vytiskni ještě ‘Bink!’. Pokud ale číslo bude menší nebo rovno 1000, vnitřku (těla) if si vůbec nevšímej a pokračuj hned prvním příkazem za ním - vytiskni jen ‘Bink!’.

Varování

Většina programovacích jazyků si nevšímá, jestli jsou na začátku řádku nějaké mezery. To znamená, že musí řešit podobný problém jinou cestou; obvykle potřebují nějaké endif na konci if, nebo trvají na tom, že vše uvnitř if musí být ohraničeno nějakými druhy závorek. Pythonýrský způsob je snadnější číst.

Prostředí IDLE toto z podstaty podporuje. To znamená, že když za tou : dáte Enter, tak editor sám odsadí pár mezer od začátku. Zpětný návrat na začátek řádku musíte udělat sami, pomocí Mazání.

Nyní změňte < na > a spusťte program znovu. Můžete odhadnout, co se to děje? (V případě, že jste se ve škole ještě nesetkali s těmito symboly: < značí “je menší než” a > značí “je větší než”.)

Pomocný úkol: Napište program, který se vás zeptá na číslo a pak vytiskne různé hlášky v závislosti na tom, jestli je číslo větší nebo menší než 100.

My samozřejmě pro náš testovací program nepotřebujeme znát, jestli číslo, které jste napsali, je větší než správná odpověď; my chceme vědět, jestli je stejné.

Možná byste napsali `if neco = necojineho:` a tak dále a očekávali, že to bude správně; nicméně Python, aby zabránil kolizím mezi použitím = ve smyslu "je stejné jako" a použitím = ve smyslu "je jméno pro", používá dva různé symboly pro tyto dva smysly. Již jsme si ukazovali, že = značí "je jméno pro" a tak tedy "je stejné jako" musí být něco jiného.

A opravdu v Pythonu se "je stejné jako" píše se dvěma rovnítky, tedy takto: `==`. Použití jediného `=` vede k chybě. (Pokud chcete vědět více o `if` a podobných věcech, podívejte se na Podmínky a podmíněné příkazy.) Tedy např.:

```
if odpoved==1000:
    print ("cislo je tisic")
```

Pomocný úkol: Změňte prográmek, který jste před chvilkou napsali a který testoval, jestli číslo je větší než 100 tak, aby testoval jestli číslo je rovno 100.

Úkol: Nyní by již neměl být problém předělat náš testovací program tak, aby tiskl "ano!" nebo "ne!" podle toho, jestli vaše odpověď bude správná nebo ne. Pokud bude odpověď špatná, vytiskne ještě navíc kolik to má být správně. A máme tedy program, který vás otestuje najeden příklad na násobení a pak skončí. Děláme pokroky.....

Znovu a znovu a znovu

Možná si pamatujete cyklus `for` z Python se představuje. Ať ano nebo ne tady je příklad, jak ho použít. Napište krátký program:

```
seznam = [1, 2, 3, 4, 5, 6]
for x in seznam:
    print ('x =', x)
```

Dáváte si stále pozor na to, co znamenají mezery na začátku řádku? Mělo by se vytisknout 6 čísel, od 1 do 6 včetně. Podobného výsledku můžeme také dosáhnout funkcí `range()`:

```
for x in range(6):
    print ('x =', x)
```

Takto dostanete také šest čísel, ale od 0 do 5 včetně. Místo 6 vyzkoušejte i jiná čísla.

Úkol: Pokud se vám to podařilo, najděte svůj testovací program a udělejte něco podobného tak, že se to celé bude opakovat 10 krát. Tip: Asi budete potřebovat odsadit několik řádek, které musí být uvnitř v cyklu for.

Pokud chcete vědět více o tom, jak dělat věci v Pythonu znovu a znovu dokola, podívejte se na Cykly.

Kdo to počítá?

Před chvílí jsem si stěžoval, že náš program nutí člověka, který ho používá, k tomu, aby si sám kontroloval své odpovědi. To jsme již zvládli, ale nyní si stále ještě uživatel musí počítat správné odpovědi. Počítač by měl být schopný umět i toto.

Tady je další pomocný prográmek na vyzkoušení. Nemá toho mnoho společného s testerem násobilky, ale snad takto snadněji pochopíte, jak dále postupovat:

```
pocet=0

pocet=pocet+1
print ('Celkem je v pocet: ', pocet)

pocet=pocet+1
print ('Celkem je v pocet: ', pocet)
```

Jak může být `pocet` rovno `pocet+1`? Pamatujte, co jsem říkal před chvilkou, že Python pro “je rovno” používá `==`, a `=` značí “je jméno pro”. Ta řádka říká Pythonu toto: vypočítej kolik je `pocet + 1` a výsledek ulož zpětně do `pocet`.

Výše uvedený příklad by šel řešit také takto:

```
pocet=0
for cislo in range(2):
    pocet=pocet+1
    print ('Celkem je v pocet: ', pocet)
```

V Python se představuje jsem zmiňoval, že často jménům říkáme proměnné. Nyní jste objevili proč: věci, které takto pojmenováváme, se mohou měnit, se mohou proměňovat. V programu nahoře `pocet` na začátku je 0; pak 1, pak 2.

Úkol: OK. Zpět k testeru násobilky. Přidejte následující řádky do vašeho programu. Musíte ale vyřešit, kam to tam dát. Některé řádky budou potřebovat na začátek vložit několik mezer:

```
dobre = 0
spatne = 0
dobre = dobre + 1
spatne = spatne + 1
print ('Mate', dobre, 'spravnych odpovedi a', spatne, 'spatnych.')
```

Hotovo! Skvělé! Máte program, který zkouší člověka z násobilky!

Vylepšujeme program

Pokud jste dosud vše zvládli, měli byste mít zhruba program, který dělá to, co jsem popsal v úvodu tohoto listu. Je ale spousta věcí, která by šla udělat lépe. Nemáte-li ještě programu dost, mohli byste vyzkoušet následující:

1. Vylepšete vzhled. Je nehezké, když otázka a uživatelova odpověď je na různých řádkách, a že je tam několik zbytečných mezer. Již jsme si říkali, že čárka u příkazu print způsobí, že se další věc tiskne na tu samou řádku. Funguje to i na konci, např. `print(a,'+',b,'=',end=" ")`.
2. Nastavujeme obtížnost. Někteří lidé zvládají počty velmi dobře. Pro ně je pak nudné nechat se zkoušet z čísel od 1 do 10. Někdo ale je zase na tom s počítám velmi špatně, takže to pro něho může být těžké. Je zřejmé, že nebude těžké změnit program tak, aby používal větší nebo menší čísla. (Koukněte na program a ujistěte se, že víte jak na to.) Snad by bylo zajímavější, kdyby program po každé správné odpovědi trochu "přitvrdil" a po každé špatné odpovědi trochu "ubral" na obtížnosti. (Budete asi potřebovat více jak 10 otázek, aby toto fungovalo dobře.)
3. Nastavujeme délku. Také by šlo udělat nějaké rychlo-zkoušení, tedy třebas jen 4 otázky. A jeden superdlouhý test se 100 otázkami, abychom se ujistili, jak dlouho dokáže uživatel zůstat bdělý. Ať se tedy program zeptá, kolik otázek chcete a pak tolik otázek položí. Na to budete potřebovat vědět něco o věcech, kterým se říká "řady" (ranges). Jsou popsány v Cykly.

Co dál?¶

Další list v řadě je Pěkné obrázky, který je celý o grafice: kreslení v Pythonu.

Pěkné obrázky

Tento list vás naučí něco málo o grafice (obrázky nebo kresby) v Pythonu. Také vysvětlí velmi důležité téma : funkce neboli procedury. Není rozumné tyto příklady zkoušet v grafickém prostředí GUI IDLE, protože toto prostředí používá ke svému běhu stejnou grafiku, jako naše příklady a často pak dochází ke kolizím. Daleko lepší je Python (command line). Více o této věci v Spouštění her.

Je to docela dlouhý list. Nedělejte si moc starosti, pokud to bude chvíli trvat, než ho projdete.

Body, čáry a barvy

V Python se představuje jste viděli celkem nudný příklad grafiky v Pythonu. Projděme si ho ještě jednou, ale více podrobněji.:

```
>>> from tkinter import *
>>> platno=Canvas(width=640, height=480)
>>> platno.pack()

>>> platno.create_line(0, 0, 200, 100)
>>> platno.create_line(0, 100, 200, 0, fill="red", dash=(4, 4))
>>> platno.create_rectangle(50, 25, 150, 75, fill="blue")

>>> mainloop()
```

První řádka by vám měla být povědomá a známá. Druhá a třetí říká počítači, aby vykreslil okno, do kterého budete umísťovat čáry a kruhy a jiné tvary. Poslední řádka, `mainloop()`, čeká na to, až grafické okno zavřete a pak resetuje některé věci, abyste mohli říci znovu`platno=Canvas()` a začít načisto.

Všechny zajímavé věci se ději mezi tím.:

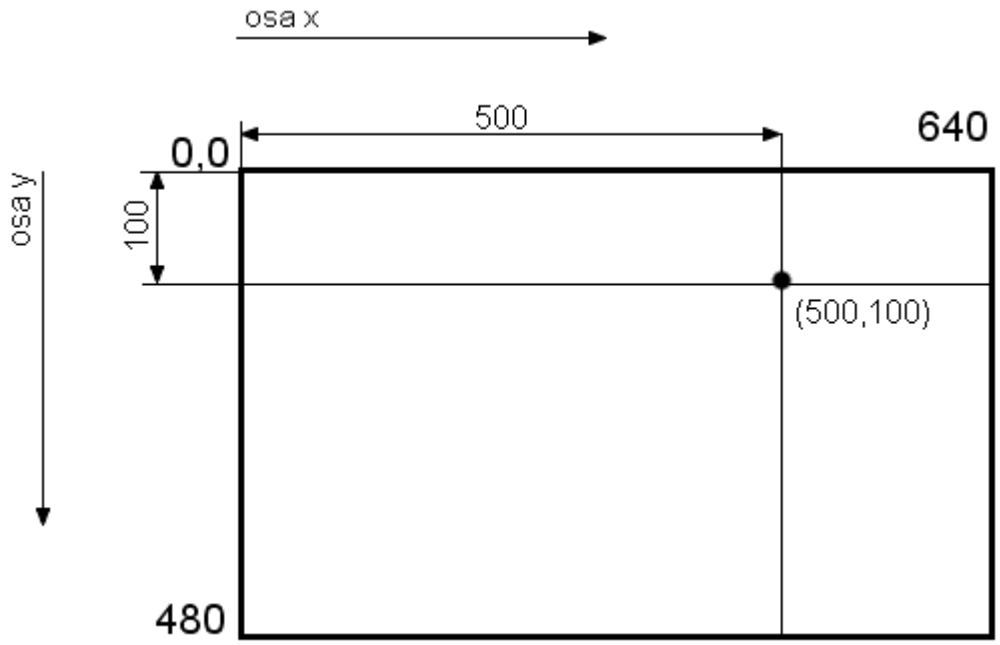
```
>>> platno.create_line(0, 0, 200, 100)
>>> platno.create_line(0, 100, 200, 0, fill="red", dash=(4, 4))
>>> platno.create_rectangle(50, 25, 150, 75, fill="blue")
```

Musíte nejdříve poznat jak Python smýšlí o grafice, abyste pochopili tyto řádky. Ve skutečnosti to není samotný Python, ale spíše modulTkinter, ale to nyní nevadí.

Pixely (body)

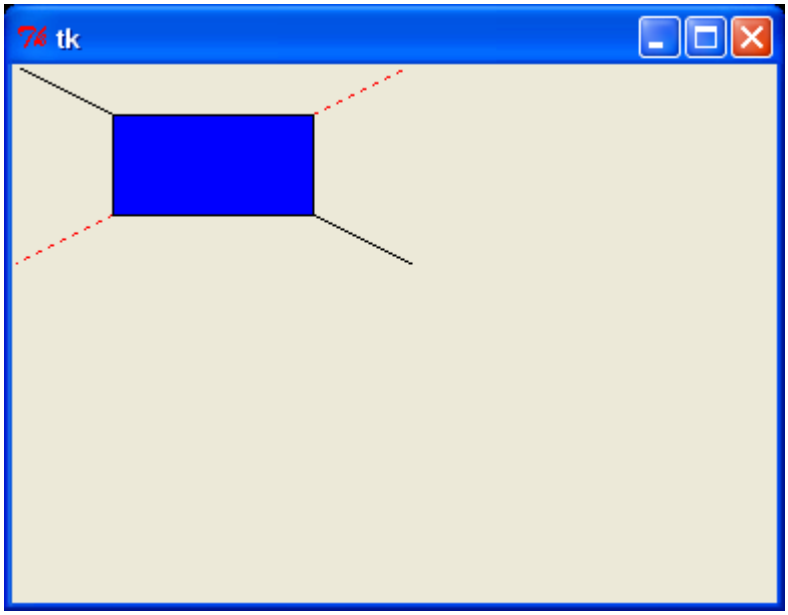
Grafické okno – jako vše na obrazovce – je složeno z pixelů (bodů): malých barevných čtverečků nebo obdélníků. Pokud se podíváte zblízka na vaší počítačovou obrazovku, uvidíte co tím myslím. Proč tak zvláštní jméno pixely? Myslím, že je to zkratka pro picture element.

Každé bod v grafickém okně má své souřadnice : nejdříve kolik je pixelů (bodů) zleva (souřadnice x) a pak kolik je pixelů (bodů) zezhora (souřadnice y). Tady je obrázek:



Body, čáry a barvy podruhé

```
>>> platno.create_line(0, 0, 200, 100)
>>> platno.create_line(0, 100, 200, 0, fill="red", dash=(4, 4))
>>> platno.create_rectangle(50, 25, 150, 75, fill="blue")
```



Nyní si již tedy můžeme říci, co výše uvedené řádky dělají:

1. první řádka maluje čáru z bodu (0,0) do bodu (200,100)
2. druhá řádka maluje červenou (fill=”red”) šrafovanou (dash) čáru z bodu (0,100) do bodu (200,0)

3. třetí řádka vytvoří modrý obdélník, jehož levý horní roh bude na (50,25) a pravý dolní roh na (150,75).

Čelíme skutečnosti

Zkusme si namalovat něco zajímavějšího. Co třebas obličej? (Později to rozšíříme na celou osobu.) Bohužel, není jednoduché namalovat obličej s rovnými čárami. Obličej je přeci trochu kulatý, a tak ....

Chodíme do kruhu

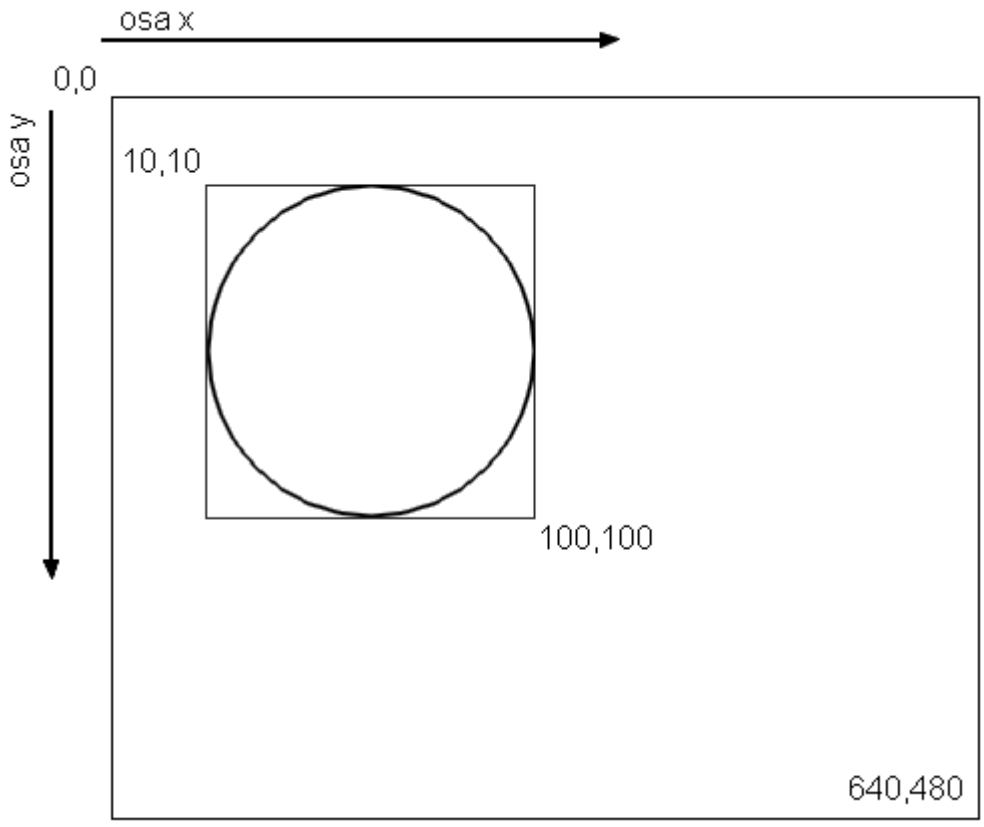
...potřebujeme umět namalovat věci, které jsou oblé. Například kruhy. Vyzkoušejte tyto příkazy rovnou jako program nebo nejdříve v příkazovém řádku:

```
from tkinter import *

platno=Canvas(width=640, height=480)  # plátno o rozměrech 640x480
platno.pack()
platno.create_oval((10,10),(110,110))

mainloop()
```

Ty čtyři čísla v závorce tvoří souřadnice dvou protilehlých rohů ohraničujícího obdélníku kolem oválu. Jinými slovy - levý horní roh a pravý dolní roh. Asi takto:



Vyzkoušejte spouštění tohoto programu s různými hodnotami (čísly) a ujistěte se, že víte, co dělá. Je možné zápis psát bez vnitřních

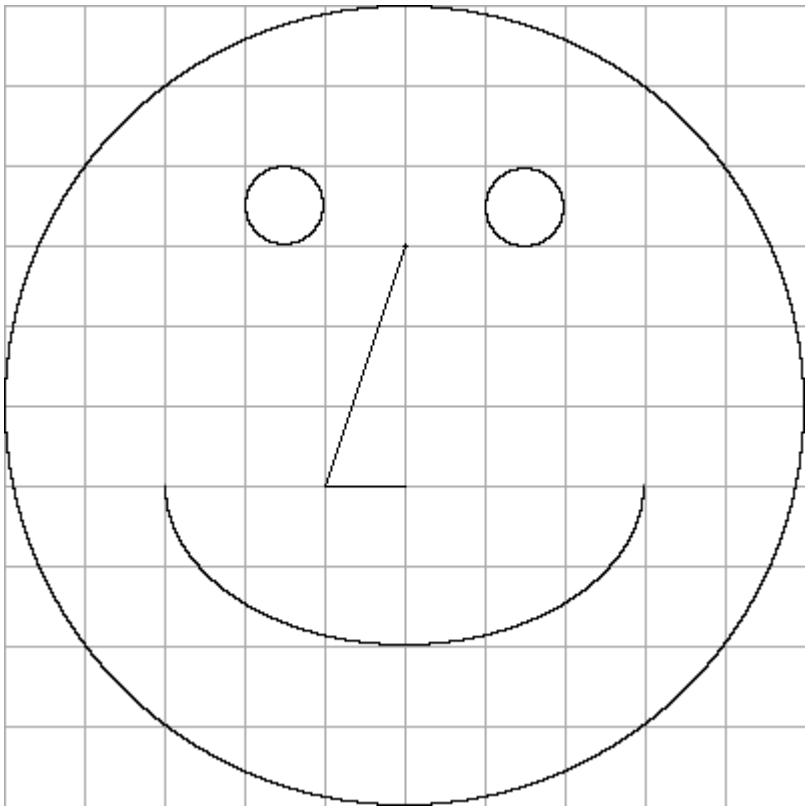
závorek:`platno.create_oval(10,10,110,110)`. A co dělá toto? (Zkuste to odhadnout předtím, něž ho spustíte.):

```
seznam=[(100,100),(80,80),(60,60),(40,40),(20,20), (-20,-20),(-40,-40)]
for souradnice in seznam:
    platno.create_oval((10,10), souradnice)
```

Obličej

Jdeme na to hlavní - zkusíme si namalovat obličej. Obličej se dá rozložit na několik tvarů:

- Jeden velký kruh (obvod celé hlavy)
- Dva malé kruhy (oči)
- Část kruhu (pusa)
- Dvě rovné čáry (nos)



Ta mřížka má 10 čtverečků na každé straně a každý čtvereček má 10 pixelů. Předpokládejme levý horní roh našeho obličeje souřadnicích (10,10). Na tom bodu není nic zvláštního, jen že je někde vlevo nahoře. Tak radši ještě jednou:

- obličej má levý horní roh (10,10) a pravý dolní roh na (110,110), (tedy rozměr obličeje je 100x100)
- levé oko má souřadnice (40,30),(50,40).
- Ostatní souřadnice nechávám vás.

Úkol: napište program, který namaluje celý obličej zatím kromě pusy (ta je trochu složitější). Na kruhy používejte metodu`create_oval(x1,y1,x2,y2,r)` a na rovné čáry `create_line(x1,y1,x2,y2)`, kde za x1,y1 musíte dosazovat souřadnice

levého horního rohu a za x2,y2 souřadnice pravého dolního rohu ohraničujícího obdélníku.

Otevíráme ústa

Jak jsem již řekl, ústa jsou těžší. Ústa jsou oblouk a oblouk se anglicky řekne arc. Hledáme tedy metodu create_arc(...), která má však poněkud složitější zadávání souřadnic. Vlastně ani ne tak souřadnic, jak spíše dalších argumentů, abychom dostali polokruh přesně tak, jak chceme. Souřadnice jsou celkem jasné: levý horní roh a pravý dolní roh ohraňičujícího obdélníku, v našem případě tedy přidáme do programu zatím toto:

```
platno.create_arc(30,70,90,90)
```

MEtoda create_arc umí kromě oblouku dělat i výseče a tětivy. Standardně dělá výseče, což je náš případ. Dále standardně dělá úhel 90°, což je také náš případ. Zkusíme to tedy opravit:

```
platno.create_arc(30,70,90,90, style=ARC, extent=180)
```

Teď již jen otočit, aby se nemračil :-(:

```
platno.create_arc(30,70,90,90, style=ARC, extent=180, start=180)
```

Úkol: Upravte váš pusu-kreslící-program tak, aby maloval pusu na jiném místě. Musíte změnit všechny čísla!

Úkol: Udělejte další změny na tváři, abyste plně pochopili smysl, jak tyto grafické příkazy fungují. Například:

- Zvedněte trochu oči.
- Přidejte obočí (čáry nad očima, snad trochu šikmo)
- Přidejte uši, vlasy.
- Změňte jemně tvar nosu.
- Použijte barvy

Více k metodě create_arc(), ale i jiným na http://tkinter.programujte.com/canvas.

Barvičky

Možná budete chtít udělat z černobílého obrázku barevný. Modul Tkinter umí mnoho barev, do začátku snad stačí pár základních: red, green, pink, yellow, atd. Použití je následující. Tento příklad kruh namaluje modře a červeně ho vybarví:

```
platno.create_oval((10,10),(110,110), fill="red", outline="blue")
```

Dvě hlavy je víc než jedna

Co kdybyste chtěli namalovat dvě takové hlavy?

Mohli byste vše zkopírovat, vložit na konec programu, změnit v kopii souřadnice a bylo by to, že? To by jistě fungovalo, ale bylo by to nudné a lehce byste se mohli splést. A pokud byste chtěli ještě jednu tvář, tak byste se asi zapotili.

Je zde lepší cesta.

Již víte, že Python vám umožňuje pojmenovávat čísla, řetězce, seznamy a další věci. Také vám dává možnost pojmenovat části programu. Část programu, která dostane jméno, se nazývá funkce (proč? - to je složitější a teď na tom nezáleží). Někdy se používá slovo procedura. Například jazyk Logo, který jste možná používali ve škole, je nazývá procedury. Nebo Baltík, ten je nazývá Pomocníci.

Funkce

Pokud se zde do toho zamotáte, vyzkoušejte Funkce. Zkuste následující příklad. Pythonýrský prompt, kterým začíná každá řádka se bude měnit podobně jako např. u for. Nezapomeňte na mezery na začátcích řádků!:

```
>>> def bajecne():
...     print ('Python je bajecny!')
...     print ('Stejne jako ja.')
...                             # Zde jen stiskněte Enter
```

Právě jste naučili Python nový trik. Co se stane, když napíšete?:

```
>>>  bajecne()
```

Zkuste to. Tímto jste nadefinovali nové slovo, kterému teď bude Python rozumět. (def je zkratka pro definovat.)

Úkol: Napište takovou funkci, že když napíšete hruza(), tak počítač vytiskne:

```
Python je pro kocku!
A stejne tak tyto listy.
```

Argument je dobrý

Funkce jsou však více než pouhá úspora znaků. Vyzkoušejte napsat následující definici a přemýšlejte o tom, co může znamenat:

```
>>> def dvakrat(x):
...     print (x, ' + ', x, ' = ', x+x)
...                             # Zde jen stiskněte Enter.
```

Pokud si myslíte, že tomu rozumíte (nebo jste se rozhodli, že tomu nebudete rozumět nikdy), experimentujte. Zkuste se zeptat Pythona nadvakrat(3), nebo dvakrat(99), nebo dokonce dvakrat('och').

Nyní je to snad již jasnější. Když řeknete `` dvakrat(7)``, počítač vykoná kód uvnitř definice dvakrat``(stejně jako to vykonal u funkce``bajecne a hruza již dříve) jen s tím rozdílem, že x je 7 . Takže je to tak, jako kdybyste řekli:

```
print (7, ' + ', 7, ' = ', 7+7)
```

Věci, které se takto dostanou dovnitř funkce, se z jistých důvodů nazývají argumenty. Takže když řeknete `dvakrat('superstar')`, tak řetězec `'superstar'` je argumentem funkce `dvakrat`. Pythonu se ale nebude líbit, když mu zadáte `dvakrat()`, tedy bez argumentu. Vyzkoušejte. Víte co se to děje? (Python hlásí chybu , že funkce `dvakrat()` očekávala 1 argument a nedostala žádný).

Úkol: Napište funkci, která se chová skoro jako `dvakrat`, ale narozdíl od tisku kolik je `x+x` (kde `x` je argumentem funkce), vytiskne, kolik je `x*x`. Vyzkoušejte. (Bohužel to nebude fungovat na řetězce!)

Funkce může mít více jak jeden argument. Jak to funguje snad pochopíte z příkladu:

```
def secti(x,y):
    print (x, ' + ', y, ' = ', x+y)
```

A můžete napsat `secti(7,5)` a počítač vám řekne co se stane, když sečtete 7 a 5.

Vestavěné funkce

Mimochodem je mnoho věcí, které jste doposud Pythonu říkali, a které jsou ve skutečnosti funkce. Když jste například říkali `print("ahoj")`tak jste používali funkci s jedním argumentem. Jediný rozdíl byl v tom, že jste nemuseli psát definici funkce sami.

Návrat k hlavě

To všechno byla jen odbočka. Předtím, než jsem začal o funkcích, mluvili jsme o tom, jak namalovat dvě (nebo více) tváří. Možná již tušíte, co bude následovat: ta jediná správná věc je napsat funkci, která namaluje obličej. Funkce by měla přijímat argumenty, které jí budou říkat, kam obličej v grafickém okně patří. Jakmile to budeme mít, tak namalovat 5 obličejů bude prostě znamenat 5x použ9t naší funkci, namísto 5-ti kopírování našeho programu a děláním spousty změn v každé kopii. Takže jak by měla vypadat naše funkce na kreslení obličeje?

Musí být schopna přijmout kam má malovat obličej. Předávejme ji tedy 2 argumenty, které jí to budou říkat. Budou to tedy nejlépe snad souřadnice levého horního rohu celého obličeje. Zbytek si musí funkce dopočítat sama. Takže nyní již víme, že naše funkce bude vypadat asi takto:

```
def malujOblicej(x,y):
    blahblah    # vše, co potřebujete k namalování obličeje
```

Vyplnění `blahblah` bude zřejmě trochu obtížnější. ;-)

Co přesně znamenají argumenty `x` a `y`? Jak jsem již řekl, měly by to být souřadnice levého horního rohu celého obličeje. Takže např.`malujOblicej(300, 200)` bude znamenat namalování obličeje, který bude mít levý horní roh na souřadnicích (300,200).

Úkol: Pokud jste odvážní a cítíte se na to, vraťte se zpět k našemu programu a zkuste ho přeměnit na funkci. Pak ji hned otestujte. Pokud se na to necítíte, pokračujte ve čtení a já se vás pokusím postupem provést.

Pokud se tedy chcete pokusit udělat úkol sami, měli byste přestat číst hned teď.

Začneme tím, že opravíme program tak, aby namísto:

```
první řádka
druhá řádka
třetí řádka
atd
```

jsme tam měli:

```
def malujOblicej(x,y):
    první řádka
    druhá řádka
    třetí řádka
    atd
```

S úspěchem na odsazování používám Tabulátor. Nemusím počet mezer počítat a dokonce mohu odsadit několik řádek najednou, když si je předtím označím. Dovnitř funkce ale nepatří vše, patří tam jen samotné malování:

```
from tkinter import *

platno=Canvas()
platno.pack()

def malujOblicej(x,y):
    platno.create_oval((10,10),(110,110))
    blablabla

malujOblicej(300,200)
mainloop()
```

Nebo ještě lépe, pro větší přehlednost, přesuňte funkci hned na začátek programu za import:

```
from tkinter import *

def malujOblicej(x,y):
    platno.create_oval((10,10),(110,110))
    blablabla

platno=Canvas()
platno.pack()
malujOblicej(300,200)
mainloop()
```

Nyní tedy máte funkci, která sice přijímá dva argumenty, ale ignoruje je a maluje obličej stále na souřadnici (10,10). Takže teď to spravíme, aby byl obličej skutečně na souřadnicích (`x`, `y`).

Obrys obličeje

První věc, kterou jsme původně kreslili, byl kruh obličeje.. Ten bude asi nejjednodušší přizpůsobit naší funkci.Původně jsme měli kruhplatno.create_oval((10,10),(110,110)). První souřadnice levého horního rohu (10,10) je ta, kterou budeme chtít měnit - pokaždé budeme chtít hlavu namalovat jinde. Takže tam napíšeme místo konstant (10,10) lépe proměnné (x,y). No a pravý dolní roh je vždy o stovku větší, tedy (x+100,y+100). Všechny naše obličeje budou mít stejnou velikost, takže ho bychom měli tuto stovku nechat beze změny. (Abych nezapomněl - všechno bude snazší, pokud budete sledovat obličej-na-mřížce výše na tomto listu.)

Změněná řádka tedy bude platno.create_oval((x,y),(x+100,y+100)). Nyní by již vám měl program malovat kruh na různých místech okna, podle toho, jaká čísla zadáte v malujOblicej(300,200). Vyzkoušejte!

Oči

Dále jsme malovali oči. Původně levé oko bylo ((40,30),(50,40)). Tedy (40,30) je o (30,20) větší než základní bod (10,10). Jinými slovy, levé oko leží na:

```
platno.create_oval(x+30,y+20,x+40,y+30)
```

Pravé oko a noc nechávám na vás ;-).

Pusa

Asi si myslíte, že tohle bude těžší, než cokoliv jiného. Ne, skutečně nebude. Jen je třeba řádně si rozmyslet, jaké jsou všechny ty čísla ve vztahu základnímu bodu - tedy kolik musíte přičíst pixelů k levému hornímu rohu.

Levý horní bod pusy byl, například, na (30,70). To je tedy o (20,60) více než základní bod (10,10), takže výsledek levého horního bodu pusy by mohl být (x+20,y+60). Budu protivný a nechám vás samostatně vypracovat, jak přesně opravit řádku programu, která maluje pusu. Skutečně to není těžké.

Funguje to?

Funkci otestujte. Přidejte pod ní několik řádek:

```
malujOblicej(300,200)
malujOblicej(400,200)
malujOblicej(350,270)
```

a spusťte program. Vidíte tři obličeje? Nebo příšerný zmatek rozházených částí obličeje? Je tam vůbec něco? Je důležité umístit tyto řádky pod definici funkce malujOblicej, protože počítač neví co to je malujOblicej, dokud si nepřečte definici funkce.

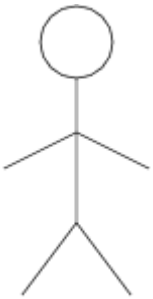
No a to je vše! Váš/náš program je hotov. Následují již jen takové speciality pro ty, kdo chtějí zkusit zažít ještě něco více.

Vylepšujeme program

Postava

Pokud již máte dost grafiky, následující pasáž přeskočte. Pokud vás to stále ještě baví, možná by stálo za to, použít vaši funkci kreslící obličej na vytvoření funkce kreslící celého člověka.

Tady je namalován jednoduchý obrázek. Nakreslete něco podobného na čtverečkovaný nebo milimetrový papír a použijte to pro design funkce, která bude malovat celou osobu. Tam, kde je namalovaná hlava, byste vlastně měli použít funkci `malujOblicej`.



Možná to bude chvíli trvat. Musíte si pořádně rozmyslet jaké argumenty předávat funkci `malujOblicej`, aby hlava a tělo seděli pěkně na sobě.

Ujistěte se, že vaše funkce fungují jak mají namalováním několika osob na různých místech obrazovky.

Možná bude jednodušší začít s malováním jednoduché postavy kdekoliv a pak ji přeměnit na funkci podobným způsobem, jako jste to udělali u tváře.

Co dál?

- Vymyslete, jak namalovat obličej a tělo v různých velikostech a/nebo na různých místech.
- Vymyslete, jak pootočit hlavu.
- Vymyslete, jak jednomu z vašich výtvorů nasadit klobouk.
- Namalujte domeček.

Také můžete prostě pokračovat na Větší nebo menší?, ve kterém si napíšete jednoduchou hru. (Bohužel, bez grafiky.)

Větší nebo menší?

V tomto listu si napíšeme jednoduchou hru Jaké číslo si myslím?: počítač (nebo další hráč) si myslí nějaké číslo a vy ho máte uhodnout. Po každém pokusu vám počítač řekne, jestli váš odhad je moc nebo málo.

Co potřebujete vědět

K dokončení tohoto listu potřebujete znát:

- Jak editovat a spustit program v Pythonu: viz Spouštíme Python
- Základy Pythona z Python se představuje a Zkoušení z násobilky.

Rozmyslíme si to

Typická hra bude vypadat asi nějak takto:

```
Dobrá. Myslím si číslo od 1 do 1000.

Jaké číslo si myslím?: 200
To je příliš málo.

Jaké číslo si myslím?: 500
To je příliš hodně.

Jaké číslo si myslím?: 345
To je příliš hodně.

Jaké číslo si myslím?: 300
```

```
To je moje číslo! Výborně!

Položil jsi 4 otázky..
Chceš hrát ještě jednou? ne

Dobře. Ahoj!
```

Program tedy musí umět následující věci:.

- Náhodně vybrat číslo a říci hráči, že už je připraven.
- Opakovaně (Dělej dokud hráč neuhádne odpověď):
  - Zeptat se hráče na číslo
  - Napsat, jestli je to málo nebo moc nebo trefa
- Počítat, kolikrát hráč hádal.
- Nakonec vypsat zprávu o hráčově výkonu a nabídnout možnost další hry.
- Pokud chce hráč další hru, má ji mít, začni od začátku.

Jednoduché

První věc na seznamu úkolů už znáte z Zkoušení z násobilky. Napište tedy první kousek našeho programu, který vybere náhodné číslo mezi 1 a 1000 a uloží ho do proměnné. Pak nám dá o tom vědět. (Nezapomeňte na `import random`!)

Měli byste znát i další věc - jak se zeptat hráče na číslo a odpovědět, jestli je to moc nebo málo nebo akorát. Něco podobného jste již dělali vZkoušení z násobilky. Budete potřebovat `if`.

Souhrn: Udělejte program, který vybere nějaké číslo, řekne vám o tom (‘Myslím si číslo .....”), zeptá se vás jednou na váš tip a řekne vám, co si o tom myslí.

Znovu a znovu a znovu dokola

V Zkoušení z násobilky byl odstavec nazvaný Znovu a znovu a znovu, který byl o cyklech `for`. Ty jsou velmi užitečné, když víte dopředu, kolikrát přesně chcete něco opakovat, ale již se méně hodí, když na začátku opakování ještě nevíte, kdy budete chtít skončit. To je náš případ: nevíme kolik bude hráč potřebovat správných odpovědí, než uhodne myšlené číslo.

Na tento druh úloh má Python jiný typ cyklu. Nazývá se cyklus `while`, protože (1) začíná slovem `while` a (2) říká počítači: prováděj tělo pořád dokola tak dlouho, dokud je pravdivá podmínka na vstupu.

Jednoduchý příklad, vyzkoušejte:

```
x=1    # takový malý trik, abychom se vůbec do těla cyklu dostali
while x < 100:
    print ('x je', x)
    x = x + 6
```

Asi odhadnete co to bude dělat. Co si myslíte, že se stane, když změníte první řádku na `x=101`, takže podmínka testovaná na začátku příkazem `while` je nepravdivá hned od počátku? Tipněte a pak teprv zkuste. Tipovali jste správně?

Úkol: Napište program, který se se vás opakovaně bude požadovat číslo a nepřestane, dokud ho nenakrmíte číslem 1234. Možná se vám bude hodit to, že v Pythonu `x != y` značí `x` se nerovná `y` .

Spousta pokusů

Nyní tedy již umíme opakovat něco znovu a znovu použitím cyklu `while`. Jak to můžeme využít v našem programu, když chceme, aby se ptal tak dlouho, dokud nedostane správnou odpověď?

Zkusme udělat jakoby nekonečný cyklus, který se bude přerušovat až v těle při nějaké podmínce. Toto řešení je dosti možná napoprvé zvláštní, přesto se hojně využívá. Nekonečný cyklus je třeba takto. Nekonečný znamená, že pokud ho nezarazíte pomocí CTRL+C nebo vypnutím počítače, bude probíhat pořád. Vyzkoušejte:

```
while True:
    print ("ahoj")
```

Složitější případ, pasovaný na naše podmínky je tento. V cykly nejdříve zvolíme náhodné číslo, vytiskneme ho, a pak testujeme, jestli to je pětky. Pokud ano, cyklus ukončíme pomocí `break`:

```
import random
while True:
    x = random.randint(1, 10)
    print (x)
    if x == 5:
        break
print ('Konečně konec! ;-)')
```

Jediná nová věc zde je předposlední řádka: `break`. `break` znamená: “jakmile na mě narazíš, přeruš okamžitě cyklus (vyskoč z něho na první příkaz za ním)”.

Když teď znáte příkazy `while` a `break`, měli byste být schopní upravit váš program tak, aby se ptal tak dlouho, dokud hráč nezadá správnou odpověď. Prostě vložte ten váš kousek programu, který již máte (která se jednou zeptá a vyhodnotí) do cyklu `while True:` a proveďte `break`, když hráč zadal správné číslo.

Kolikrát to bylo?

Sledování počtu otázek / odpovědí hráče je jednoduché.Je to hodně podobné sledování počtu správných odpovědí v testeru násobilky vZkoušení z násobilky. Založte pro to nějakou proměnnou, na začátku ji nastavte na 0 a pak do ní přičítejte 1 pokaždé, když hráč odpoví.

Tak jedeme, do toho..

Další hru?

Tahle hra je tak zajímavá, že když si ji zahrajete jednou, určitě si ji budete chtít zahrát znovu (možná). Takže když hra skončí, Python by se měl zeptat, jestli bychom si přáli hrát další hru.

Už znáte funkci `input()`, načte ve formě řetězce cokoliv, co jí zadáte:

```
znovu=input('Chcete hrat znovu? ')
if znovu=='a':
    print ("Ano, chcete")
```

Pokud vám není přesně jasné co funkce `input()` dělá, pohrajte si s ní chvíli: zkoušejte zadávat různé otázky i odpovědi (zkoušejte znaky, čísla, slova). Zkuste, aby se dalo odpovědět ‘ano’. Experimentujte.

Nebo něco zajímavějšího:

```
odpoved=input('Dali byste si cokoladu? (a/n)')
if odpoved=='a':
    print ('Lituji, ale nemam.')
elif odpoved=='n':
    print ('To je dobre, protoze stejne nemam.')
else :
    print ('Tuto odpoved neznam.')
```

Zpět k našemu programu a otázce, jestli chceme hrát ještě jednu hru. Tentokrát to neuděláme pomocí `while True:` a `break`, ale zkusíme jiný trik, také hojně v programátorské praxi používaný:

```
znovu="a"
while znovu=="a":
    print ("tak tedy znovu")
    znovu=input("Ještě jednou?:")
```

Cyklus while se bude opakovat tak dlouho, dokud mu budete psát “a”. Jakmile napíšete cokoliv jiného, skončí. Abychom se vůbec do cyklu poprvé dostali, musíme před vstupem do něho nastavit proměnnou `znovu` tak, aby podmínka u while byla pravdivá. Vyzkoušejte.

Úkol: dotvořte podle tohoto návodu hru tak, aby se na konci ptal, jestli chceme hrát ještě jednou.

Hotovo! Jsme na konci tohoto listu. Váš program je u konce. Následuje jen odstavec pro ty, kdo chtějí zkusit něco navíc.

Vylepšujeme program

Je spoustu cest, jak vylepšit program. Tady je pár návrhů.

- Určitě se vám několikrát stalo, že jste místo čísla zadali omylem nějaký znak nebo jen Enter a program skončil chybovou hláškou. Je to velmi nepříjemné, zvláště když už jsme u konce hry. Měli bychom tedy nějak “ošetřit vstup”, aby když uživatel omylem zadá něco jiného než číslo, aby

počítač položil otázku znovu. Jinými slovy, potřebujeme zde funkci, která by dokázala zachytit naší chybu. Taková funkce existuje a jmenuje se try: ... except:.... Pro bližší vysvětlení se podívejte na Vstup a výstup. A také na list o chybách v PythonuChyby.

- Nechte si hráče zvolit z jakého rozsahu čísel se chce hádat. Není nic magického na číslech od 1 do 1000, hráč by možná rád hrál rychlejší hru (1-20) nebo pomalejší (1-100000000000?).
- Udělejte z toho hru pro dva hráče. Jeden hráč si číslo myslí a zadá a druhý hráč se ho snaží pak uhodnout. Smažte obrazovku, když první hráč zadá číslo.
- Pokud to zvládnete, můžete pak udělat, aby si hráči v příštím kole vyměnili místa, takže ten co zadával bude hádat a naopak.
- A pokud zvládnete i to, zeptejte se na začátku na jméno hráče, takže při dalším kole již počítač může sám počítač psát, kdo z hráčů volí a kdo hádá.
- Program také může sledovat, kolik v průměru potřeboval každý z hráčů na uhodnutí čísla.
- (Toto je složitější!) Obraťte hru – myslete si číslo a nechte počítač, ať ho uhodne, ať se vás ptá. Musíte si dobře rozmyslet strategii, jak vysvětlit počítači jak má hádat.....

Z tohoto místa můžete pokračovat na Roboti přichází!, kde si napíšete zajímavější grafickou hru za pomoci znalostí z Pěkné obrázky.

Roboti přichází!

Toto je poslední číslovaný list v naší pythonýrské učebnici. Na konci, pokud ho ovšem dokončíte, budete mít funkční program, který hraje překvapivě návykovou hru: jste obklopeni roboty, kteří se vás snaží chytit a vy je musíte přelstít a dosáhnout toho, aby jeden do druhého naráželi a tím je vlastně eliminovat.

Bude to celkem obtížné také proto, že vám nebudu dávat tolik nápovědy, jako v předcházející listech. Předpokládám, že věci budete zkoumat a objevovat a studovat v jiných listech sami. Přesto tento list vám ukáže spoustu nových věcí, které jste ještě neviděli, takže se nebojte kdykoliv zeptat. Při výkladu budu často odskakovat k jiným příkladům a programům, abyste pochopili smysl toho co ukazuji, a pak se zase vrátím k našemu hlavnímu programu.

Co musíte znát

- Základy Pythona (z listu Python se představuje a Pěkné obrázky)
- Jednoduchou pythonýrskou grafiku (z Pěkné obrázky)
- Funkce (Pěkné obrázky a snad také Funkce)
- Cykly

Pravidla

Radši začnu hned tím, jak přesně hra funguje.

Honí vás roboti. Jsou ozbrojeni a když vás chytí, tak jste mrtvi. Nemáte žádné zbraně. Roboti jsou naštěstí hloupí. Jdou pořád za vámi, i když jim něco stojí v cestě. Když se srazí dva roboti, umírají oba a zůstává po nich hromada šrotu. Takže vy se snažíte stát tak, aby když vás roboti honí, aby do sebe naráželi.

Z toho vyplývá, že to bude trochu těžké; je snadné se dostat do místa, kde nemůžete dělat nic jiného, než jen čekat, až do vás roboti narazí. Hráč tedy bude mít schopnost teleportace na libovolné místo na obrazovce.

Nejjednodušší bude hru umístit jakoby na čtvercové pole. Hráč se může hýbat v 8 azimutových směrech a robot se pohne vždy v jednom cyklu v jednom z těch 8 směrů o jeden čtverec; pokud je robot například na severovýchodě od hráče, tak se pohne jeden čtverec jižně a jeden čtverec západně.

Kostra programu

Plánujeme

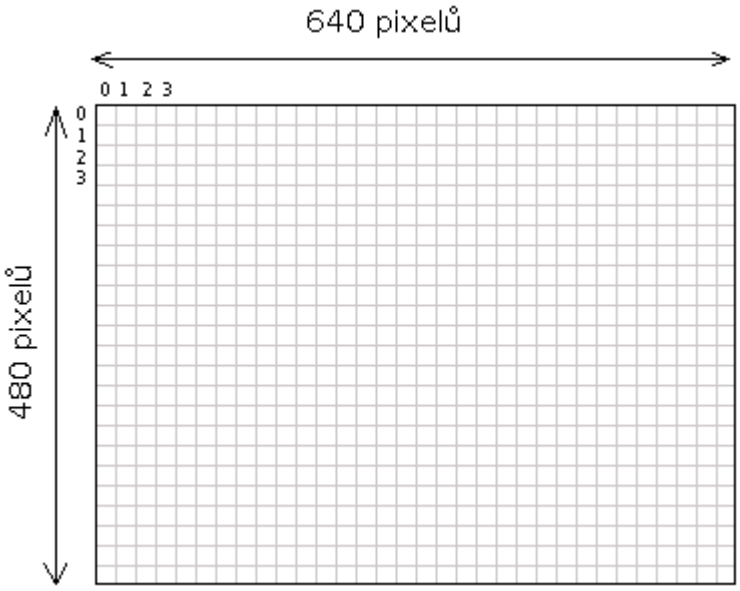
Začněme tím, co bude při hře potřeba zajistit.

- Umístit hráče a roboty na obrazovku.
- Opakovaně
    - posunout hráče, případně teleportace
    - posunout roboty směrem k hráči
    - kontrolovat - srážky robotů a robotů s hromadami šrotu. - jestli hráč vyhrál - na ploše je samý šrot - jestli hráč prohrál - hra končí

Spousta věcí. Začneme jako obvykle nějakými lehkými kousky.

Kostra

Všechny akce budou probíhat na mřížce. Naše grafické okno bude 640 x 480 pixelů. Vcelku vhodná velikost.



Poznámka

to be continued sooner or later ....

Spouštíme Python

Instalace Python

- Windows: Stáhněte si z http://www.python.org/download/ Python a nainstalujte. V menu Start by vám měla vzniknout nová položka a v ní několik odkazů.
- Linux: liší se distribuce od distribuce. Na Ubuntu např. stačí: sudo aptitude install python, ale většinou Python již všude je.

Python se může spouštět ve dvou režimech. Níže si popíšeme práci s oběma z nich.

- v režimu Příkazovém řádku: něco napíšete a hned dostanete odpověď. Někdy se tomu také říká prompt, command line nebo shell a řádek vždy začíná >>>. Může to být IDLE (Python GUI) nebo Python (command line)
- v režimu editoru pro psaní programů: Python (command line) - File - New Window nebo jakýkoliv textový editor (např. PSPad či gedit)

Hraní si na Příkazové řádce

Pusťte IDLE (Python GUI) nebo Python (command line). Počítač píše `>>>` , značí to, že je připraven přijímat příkazy, které mu dáte. Na konci každé řádky, kterou napíšete, zmáčkněte klávesu Enter. Tento >>> se nazývá “výzva”, někdy také zůstává anglicky “prompt”. Okno se většinou nazývá Python Shell nebo Python (command line).

Je to vhodné na zkoušené jednotlivých příkazů, ne však na psaní opravdových programů, scriptů.

Je příjemné moci zadávat příkazy jeden za druhým a hned vidět výsledek. Ale když chcete stvořit něco komplikovanějšího, možná budete si to chtít někam uložit, abyste mohli to pak lehce upravovat a spouštět to znovu a znovu a nemuset to znovu a znovu psát.

IDLE editor

V pythonýrském příkazovém okně (“Python shell”), myší klikněte na File (Soubor) vlevo nahoře. Objeví se menu. Zvolte New window (Nové okno). Mělo by se vytvořit nové okno, skoro jako Python shell, ale s titulkem Untitled (bez názvu) a bez promptu. Zde se píší pythonýrské programy.

Nastavení

Před vlastním psaním programů doporučuji si trochu pohrát s nastavením prostředí v IDLE: Option - Configure IDLE - General, a zde nastavit následující volby:

- No Prompt (At Start of Run (F5)) - aby se IDLE při každém spuštění programu neptalo, jestli chcete uložit
- UTF8 (Default source encoding) - nastavit utf8 jako standardní kódování
- dále si můžete nastavit co je libo ...

Psaní a ukládání

Pokud si chcete napsaný program uložit, jděte do “File” a tentokrát zvolte “Save”. Mělo by se objevit okno “Uložit jako”. Pokud jsme vše udělali správně, většina okna by měla být prázdná, nebo obsahovat programy, které jste již dříve uložili.

Pokud jste se ujistili, že jste na správném místě, klikněte do rámečku vpravo od slov Název souboru a zadejte jméno vašeho programu. Měli byste naspat něco, co bude končit na `.py` jako například `kruhy.py` nebo `hra.py` . Bude potom pro počítač jednodušší poznat později, že je to program pro Python.

Takže nyní máte uloženou kopii vašeho programu na disku počítače. Zůstane tam dokonce i když počítač vypnete. Pokud uděláte v programu změny, a chcete aktualizovat uložený soubor (a nahradit cokoliv tam bylo předtím tím, co máte v programu teď), znovu zvolte Save z File nabídky.

Otevření programu

Pokud jste program uložili a chcete se na něj znovu podívat (abyste ho upravili, nebo jen znovu spustili), zvolte Open (Otevřít) z File menu. Dostanete okno podobné tomu “Save As”. Dvakrát klikněte na souboru, který chcete nahrát.

Spuštění programu

Tak máte program v okně a chcete vyzkoušet, jestli funguje nebo ne. Jinými slovy chcete “spustit” (“run”) program.” Myší klikněte na menu Run (na vršku okna). Jedna z položek se nazývá Run module; klikni na ní. Tak se spustí tvůj program.

Spustit samostatně program můžete ve Windows v Průzkumníkovi, když ho zde najdete a dvakrát na něj klikněte. Nebo z příkazové řádky v Linuxu (i ve Windows, pokud stojíte ve správné složce), zadejte `python mojehra.py` , kde namísto `mojehra.py` zadáte jméno uloženého programu.

Při spouštění programů ve Windows přes Průzkumníka se může se někdy vyskytnout následující problém. A to, když v programu bude nějaká chyba a program skončí chybovým hlášením, nebo dokonce i při normálním ukončení vašeho programu. Windows okno s vaším programemokamžitě zavře a vy neuvidíte žádný výsledek. Řešení tohoto problému můžete nalézt na http://www.py.cz/PythonInstalacePodWindows.

Čeština v programech

Pro absolutní začátečníky je silně doporučeno česká písmena s háčky a čárky v programech vůbec nepoužívat. Pokud to ale přesto budete chtít, upozorním na

následující nepříjemnost: IDLE editor je bohužel zatím přímo nepřátelský k češtině. Následující program:

```
print ("ěščřžýáíé")
```

zcela spolehlivě celé IDLE shodí :-(. Pokud se tak stane, máte dvě možnosti:

- nepoužívat v programech češtinu
- přejít na jiný editor - pod Windows např. PSPad. Ten, při dodržení několika základních principů správného kódování, funguje s českými háčkami a čárkami naprosto bezchybně.

PSPad

PSPad je skvělý editor pro Windows, kde můžete také psát své programy. Po instalace je třeba jej trochu doladit, aby byl užitečným pomocníkem a pomáhal, tak jak má: http://www.py.cz/PSPad. Oproti IDLE má výhodu, že zde funguje dobře v programech i čeština, tedy háčky a čárky.

Spouštění her

IDLE a hry i grafika někdy spolu moc dobře nefungují. Někdy vám může IDLE zamrznout nebo spadnou, když se budete snažit hrát hry. Je to proto, že IDLE i vaše grafická hra běží pod stejným prostředím - Tkinter. Nejlepší cesta je přestat IDLE používat a přejít na jiný editor.

Také u grafiky (a tedy i u grafických her) není vhodný na zkoušení příkazů “Python Shell” z IDLE, ale - mnohem raději - “Python (command line)”, který byste měli najít v menu Start na podobném místě, jako jste našli IDLE. Důvod je stejný jako výše.

Vstup a výstup

Téměř všechny zajímavé programy musí odněkud informace získávat a posílat někam odpovědi. Tomu se říká “vstup” a “výstup” Tento list popisuje některé způsoby, jak Python nakládá se vstupy a výstupy.

Vstup

Pokud chcete, aby se počítač na něco zeptal, a čekal, než uživatel zadá odpověď (včetně Enteru), pak použijte funkci input(). Funkce input()očekává cokoliv: můžete zadat jen nějaké číslo, nebo jeden znak, nebo celou větu, nebo i nic (ale musíte vždy dát Enter na konci). Ať zadáte cokoliv, počítač to převede na text (řetězec), a to i když zadáte číslo. To je třeba mít na paměti a počítat s tím:

```
>>> cislo=input("Zadej číslo:")
54
>>> cislo+5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> cislo + " ahoj"
'54 ahoj'
>>> int(cislo)+5              # musíme nejdříve číslo-řetězec převést na skutečné číslo
59
```

Je to vidět hlavně na 3.-6. řádku, protože text-řetězec s číslem skutečně Python sečíst nedokáže. Vzorec cislo + 5 vlastně znamená “54”+5. To jak tento součet udělat správně je na předposledním řádku.

Jak ošetřit vstup?

Chcete, aby program nezhavaroval, když program chce číslo a uživatel zadá nějaký znak nebo jen Enter? Aby se ho ptal donekonečna pořád dokola, dokud nezadá číslo nebo to co chcete?

K tomu, abyste plně pochopili jak na to, je třeba znát příkaz `while` z Cykly. Nabízím zde jedno z možných řešení, kde se objevuje kouzelný příkaz `try:... except:....`

```
while 1:          # delej donekonecna, nebo dokud nenarazis na break
    vek=input("Kolik je ti let: ")
    try:          # zkus vykonat, co je uvnitr
        vek=int(vek)
        break     # vyskakuje z nekonecneho cyklu while
    except:       #a pri chybe pokracuj zde
        pass      #a nedelej nic

print (vek)
```

Výstup může být takový:

```
Kolik je ti let: nevim
Kolik je ti let: málo
Kolik je ti let: nula
Kolik je ti let:
Kolik je ti let:
Kolik je ti let: 14
14
```

Pokud uživatel zadá něco jiného než číslo, příkaz `vek=int(vek)` vygeneruje chybu, program však nezhavaruje (protože patřičná část kodu je umístěna v těle příkazu `try:`), ale okamžitě pokračuje na `except:` a k příkazu `break`, nedorazí. A protože je smyčka téměř nekonečná, pokračuje zase tam, kde začal, příkazem `try:`, následovaným otázkou. Jakmile uživatel zadá správně číslo, program dorazí k příkazu `break`, který okamžitě cyklus `while` přerušuje a pokračuje hned za ním, tiskem proměnné `vek`. Slovo pass znamená Projdi. Nezastavuj se. Nemám co bych ti řekl. Je to příkaz v Pythonu a používá se i v příkazech `if` a `while`.

Příkaz `try:` tedy přemýšlí asi takto: pokud se kdekoliv mém těle vyskytne jakákoliv chyba, při které by se normálně ukončil program chybovou hlášku, tak já nezhavaruji, ale předám řízení mé sestře, příkazu `except:`. Pokud se ale nevyskytne ani jedna chyba, sestry si nebudu všímat, a budu pokračovat v běhu programu dál za ní.

Celé by se to dalo ještě vylepšit:

- vyhodnocení vstupu na smysluplnost (vek nemůže být přeci menší než 0 nebo větší než, dejme tomu 150)
- omezit počet dotěrných stejných otázek na, dejme tomu 5, pak skončit (protože uživatel asi neumí číst)
- první otázka normální, další již jiné, upozorňující na to, že se někde stala chyba
- uzavřít vše do Funkce vracející hodnotu toho, co přečetla ze vstupu

Výstup

Nejdůležitější věcí, kterou potřebujete o výstupu znát, je funkce `print()`. Tiskne jakýkoliv objekt:

```
>>> x=[1,2,3] # Seznam,
>>> y='zog'   # řetězec,
>>> z=99      # číslo,
>>> f=repr    # funkci
>>> print (x,y,z,f)
[1, 2, 3] zog 99 <built-in function repr>
```

Všimněte si, že mezi jednotlivými tisky jsou vloženy mezery.

Když napíšete dva příkazy `print`, jeden za druhým, uvidíte, že ten druhý tiskne na novou řádku a nepokračuje tam, kde první skončil, na té samé řádce. Pokud to ale chcete změnit, zadejte jako druhý argument end=” “:

```
print (123, end=" ")
print (456)
```

To vytiskne `123 456` na jedné řádce.

Řetězce

Řetězec je kus textu. Řetězce jsou sestaveny ze “znaků”; znak je jednotlivé písmeno, symbol, nebo cokoliv. Python poskytuje mnoho “nářadí” pro práci s textem: tento list vám o některých z nich řekne.

Zadáváme řetězce

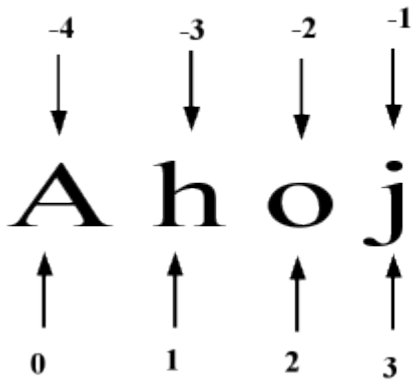
Zadávat řetězce do programu můžete pomocí uvozovek nebo pomocí funkce input(). Nezáleží, jestli jednoduché nebo dvojité uvozovky, ale musíte použít stejný typ na začátku i na konci! Zde častěji používám jednoduché uvozovky. V pohodě ale můžete používat dvojité, pokud jim dáváte přednost:

```
jedna="ahoj"
dva=input("co chceš?")
tri='nazdar'
ctyry='Řekl: "je to nuda."'
```

Řetězce jako posloupnosti

Pár věcí funguje stejně na seznamy jako na řetězce, protože jak seznamy tak řetězce jsou “posloupnosti”. Stručný souhrn: (Chcete-li, porovnejte s Seznamy a entice.):

```
>>> mroz = 'Jsem velmi pekny mroz'     #Řetězec, se kterým budeme pracovat
>>> pokus = "Neco jineho"              #A další
>>> mroz+pokus                         #Spojení řetězců...
'Jsem velmi pekny mrozNeco jineho'     #funguje jak asi čekáte
>>> 2*mroz                             #Násobení řetězců...
'Jsem velmi pekny mrozJsem velmi pekny mroz'   #také funguje jak asi    čekáte
>>> mroz[0]                            #Začínáme počítat u 0
'J'                                    #takže to je první písmeno
>>> mroz[4:11]                         #Znaky od 4 (včetně) do 11 (vyjma)
' velmi '                              #tedy celkem 7 znaků
>>> len(mroz)                          #Kolik má mroz znaků?
21                                     #Tolik!
```



Operace s řetězci

Nad řetězci se dá dělat spousta operací (metod). Uvedu jen pár nejužitečnějších z nich:

```
>>> honza='mrOZ'
>>> honza.capitalize()                #První písmeno velké
'Mroz'
>>> honza.lower()                     #Všechna písmena malá
'mroz'
>>> 'MroZ'.upper()                    #Všechna písmena velká
'MROZ'
>>> 'Jsem velmi pekny mroz'.split()     #Vytvoř seznam ze slov
['Jsem', 'velmi', 'pekny', 'mroz']
>>> 'Jsem velmi pekny mroz'.replace('e', 'oho')   #Nahraď 'e' za 'oho'
'Jsohom voholmi pohokny mroz'
```

Je daleko více věcí, které můžete s řetězci dělat. Pokud byste rádi a nevíte si rady, zeptejte se.

Seznamy a entice

Seznamy

Chleba, chleba, chleba

Seznamy jsou kolekce věcí. Děláte si seznamy, když chodíte nakupovat, seznamy úkolů nebo seznamy robotů, které máte ve vaší hře. A jak udělat seznam v Pythonu?:

```
>>> jidelnicek = ['chleba', 'vajicka', 'chipsy', 'fazole']
>>> cisla = [1, 2, 3]
>>> mismas = [2, "nevim", "kudy", 4, "zlato"]
>>> prazdne = []
```

Ano, jen uzavřete cokoliv, co chcete zahrnout do seznamu, do hranatých závorek, dokonce i když to něco je nic! Mimochodem, to není hloupý nápad, jak to snad vypadá; všechno musí odněkud začít, tak proč nezačít s prázdným seznamem a pak přidávat věci, jak budete potřebovat.

Pokud jste již někdy programovali v jiných jazycích, možná vás hned napadlo, že pythonýrské seznamy jsou jako pole. Blahopřeji, jsou to pole, povětšinou. Nicméně čtěte dál; přijdou jistě nějaké fintičky.

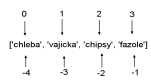
Kousek chleba, pane?

Jak zjistíme, co je uvnitř seznamu? Python jednoduše všechny položky seznamu čísluje a umožňuje nám zjistit jednotlivé položky napsáním pořadového čísla do hranatých závorek za jméno seznamu. Zkuste toto:

```
>>> jidelnicek[1]              # První věc na seznamu?
'vajicka'                  # To není 'chleba'.  Oops!
```

Python ve skutečnosti čísluje seznamy od 0 a ne od 1, takže náš `chleba` bude `jidelnicek[0]`. Dokonce můžete používat i záporná čísla, abyste četli od konce seznamu, pokud se vám to bude hodit. Poslední položka seznamu má číslo `-1` , takže snad to funguje tak, jak byste čekali!:

```
>>> jidelnicek[-2]
'chipsy'
>>> cisla[-1]
3
```



Se seznamy se dají dělat i další věci - můžete je “krájet” (slice) - vytvoření nového seznamu jako části toho starého. Prostě řekněte Pythonu, odkud chcete začít a kde chcete skončit a oddělte tyto hodnoty dvojtečkou `:`

```
>>> jidelnicek[1:3]
```

```
['vajicka', 'chipsy']     # Vezmi polozky 1 a 2, u 3 zastav.
>>> jidelnicek[:2]
['chleba', 'vajicka']     # Pokud neřeknu, začni od začátku.
>>> jidelnicek[2:]
['chipsy', 'fazole']      # Pokud neřeknu, skonči na konci.
>>> jidelnicek[:]
['chleba', 'vajicka', 'chipsy', 'fazole']       #Zbytečnost!
```

Se seznamy můžeme dělat ještě i další věci, které jsme si ještě neukazovali:

```
>>> jidelnicek + cisla    # "Spojení" funguje jako u řetězců.
['chleba', 'vajicka', 'chipsy', 'fazole', 1, 2, 3]
>>> ['x'] * 3             # "Opakování", zase jako u řetězců.
['x', 'x', 'x']
>>> cisla.append(100)     # Přidej něco na konec seznamu.
>>> print (cisla)         # Vytiskni to!
[1, 2, 3, 100]
>>> len(cisla)            # Jak dlouhý je můj seznam?
4
>>> cisla.reverse()       # Otoč seznam vzhůru nohama.
>>> print (cisla)         # A znovu ho tiskni.
[100, 3, 2, 1]
>>> cisla.sort()          # Seřaď seznam.
>>> print (cisla)
[1, 2, 3, 100]
>>> cisla.index(3)        # Na kterém místě je v seznamu 3?
2                         # Posice 2 (počítáme od 0)
>>> cisla[2]              # Zkontrolujme, jestli je to skutečně na pozici 2.
3                         # Ano!
>>> cisla.index(1234)     # A co když tam není?
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  ValueError: list.index(x): x not in list
  # Eeeeech. Takže to radši nedělejte.

>>> del cisla[2]          # Vymaž položku ze seznamu.
>>> cisla
[1, 2, 100]               # Je pryč.
>>> list(range(3))        # Vytváření užitečných seznamů.
[0, 1, 2]
>>> list(range(1,4))      # Trochu jako krájení (slicing) naopak.
[1, 2, 3]
```

(Některé z příkazů snad vypadají dosti zvláštně – např. `cisla.sort()` . Zatím to nechte tak!) Funkce `range` je velmi užitečná pro cykly `for` – viz Cykly. Je spousta dalších věcí, které můžete dělat se seznamy.

Seznamy ze seznamů ze seznamů ...

Někdy jeden list nestačí. Mysleme si, že si chcete psát seznam toho, co jste měli na snídani; je jednoduché napsat:

```
>>> snidane = ['kava', 'kukuricne lupinky', 'toast', 'marmelada']
```

a pokračovat. Ale co když budete chtít seznam toho, co jste měli ke snídani každý den? Co uděláte teď?

Python nám naštěstí může s tímto dost pomoci. Vzpomeňte si, že jsme si říkali, že seznamy jsou jen kolekce věcí. A seznam, to je také věc, takže udělat seznam seznamů je jako udělat seznam z čehokoliv jiného!:

```
>>> snidane = [
...   'Pondeli', ['kava', 'rohliky'],  # Pamatujte na mezery na začátku řádku!
...   'Utery', ['dzus', 'toast', 'marmelada'],
...   'Streda', ['kava'] ]
```

(Podívejte se, ve středu jsem spěchal!). Pythonu nevadí, když to pro přehlednost rozdělíme na víc řádků. Prostě si počká, až mu napíšeme uzavírací závorku.

Někteří lidé říkají seznamům ze seznamů “2-rozměrné matice” nebo “tabulky”, protože je můžete zapisovat do řádků a sloupců (2 rozměry) jako tabulky. Narozdíl od skutečných tabulek (a mnoha jiných počítačových jazyků) v Pythonu nemusí mít každá řádka stejnou délku).

Pokud přidáte pořadí položky (počítáno od nuly!) v hranatých závorkách za jméno seznamu, získáte jednu položku seznamu. Pro seznam v seznamu potřebujete 2 sady hranatých závorek a tak dále:

```
>>> snidane[3]
['dzus', 'toast', 'marmelada']
>>> snidane[3][0]
'dzus'
```

Lstivá past

Doposud jsme mluvili o seznamech jako o kolekci věcí. Je však přesnější říkat, že pythonýrské seznamy jsou ve skutečnosti kolekcereferencí nebo ukazatelů nebo odkazů na věci. Pro většinu případů zde není žádný rozdíl a můžete mě naštěstí ignorovat, když jsem takový puntičkář. Když však vytváříte seznam z proměnných nebo jiných seznamů, tak je to již důležité.

Když chcete například udělat kopii seznamu, získáte kopii naprosto stejného seznamu. Je to pak vše vlastně jeden a tentýž seznam, i když má jiné jméno. Takže když změníte jeden, změní se obsah druhého. Následující tedy nefunguje tak, tak jak byste snad očekávali:

```
>>> pondeli=['matematika','fyzika','pocitace']
>>> utery=pondeli
>>> utery.append('hudebka')
>>> print (pondeli)
['matematika', 'fyzika', 'pocitace', 'hudebka']
```

Jsou cesty, jak to vyřešit. Nejsnazší je udělat něco, co udělá nový seznam (skutečnou kopii) ze staré a to je `utery=pondeli[:]` (pamatujete, že jsem vám to již ukazoval výše?), nebo napsání každého seznamu odděleně anebo ukrojením části seznamu. (Pokud jste ničemu z předchozího nerozuměli, nezoufejte.)

Entice

Entice je jako seznam. Místo hranatých závorek píšete obyčejné. Narozdíl od seznamů jsou však neměnné - prostě nemůžete měnit obsah.:

```
>>> entice=(5,4,"hroch")
>>> entice=(5,4,"hroch")
>>> entice[0]
5
>>> entice[0]=5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Entice můžete používat téměř všude, kde seznamy. Můžete například napsat `for x in (1,2,3,4):` namísto `for x in [1,2,3,4]` (viz Cykly, když nevíte o čem je řeč).

Slovníky

Pracovní listy, které tvoří většinu této učebnice, nepokrývají vše. Tento list vám řekne něco o slovnících.

Slovníky

Nejjednodušší způsob na vysvětlení slovníků je příklad. Tak tady je jeden:

```
>>> slov = {}           #Prázdný slovník
>>> slov[1] = 'boo!'     #Vložíme nějakou informaci.
>>> slov['och'] = 99     #A další.
>>> slov                #Podívejme se, co je v něm.
{1: 'boo!', 'och': 99}   #Dvě "spojení"
>>> slov['och']         #Co je spojeno s 'och'?
99                    #Tohle.
```

Takže slovník je podobný seznamu až na to, že seznam má položky označené čísly, zatímco položky ve slovníku mohou být značeny i jinými věcmi než jsou jen čísla. (Nejčastěji se označují čísly nebo řetězci). Někdy pomáhá přemýšlet o slovnících, jakože jsou složeny ze spojení. Spojení (asociace) je kousek informace, jako například: `'och'` je spojeno s `99`. Takže slovník, který jsme sestavili výše, má dvě spojení.

Slovník se dá používat jako záznamník adres, kde každé spojení páruje jméno člověka s jeho adresou. Nebo může obsahovat informaci o všech hráčích v multiplayerové hře nebo o všech počítačích připojených do sítě. Ty dvě části spojení se někdy nazývají klíč (key) a hodnota (value): hodnotu naleznete, když vyhledáte klíč. Takže ve slovníku vytvořeném výše jsou klíče `1` a `'och'`, a odpovídající hodnoty jsou `'boo!'` a`99`.

Jak jsme již viděli, použití slovníků je stejné jako u seznamů: pokud `d` je slovník, pak `d[k]` je hodnota spojená s klíčem `k`. Když chcete změnit spojení ve slovníku, řekněte jednoduše něco jako `slov['och'] = 1000`. Pro úplné smazání položky použijte `del slov['och']`. (`del` je zkratka pro anglické delete (smazat).) Pokud chcete otestovat slovník jestli vůbec má klíč `k`, proveďte něco jako:

```
>>> slov = {'a': 123, 'b': 987}   #Založíme slovník.
>>> 'a' in slov                  #Je tam klíč 'a' ?
True
>>> 'z' in slov                  #Je tam klíč 'z' ?
False                         #Ne.
```

Cykly

Počítače jsou dobří v opakování věcí; často po nich chceme, aby něco dělali pořád dokola (snad s malými změnami z času na čas):

- Sečti všechna čísla v seznamu
- Přesuň všechny Ďábelské cizí vetřelce blíže k Zemi
- Vytiskni všechna čísla od 1 do 100
- Ptej se tak dlouho, dokud nedostaneš správnou odpověď

a tak dále. Někdy se tomu říká opakování nebo procházení cyklem. Python má dva druhy cyklů. Tento list vám o nich řekne více.

Dva druhy cyklů (smyček)

Máme dva druhy smyček, mezi kterými jsou podstatné rozdíly. První smyčka nazývaná `for` se používá, když dopředu víte kolikrát budete chtít tu- kterou-věc opakovat. Ta druhá smyčka nazývaná `while` se používá, když to dopředu nevíte.

Cyklus for: Když víte kolikrát

Cyklus `for` se nazývá `for` protože první věc, kterou musíte psát, když ji zakládáte, je slovo `for`. Nejsnadnější forma vypadá asi takto:

```
mujSeznam = [1,2,3,4,5,1000]
for x in mujSeznam:
    print ("Tady je cislo:", x)
```

Takže je třeba zadat jméno proměnné `x` a seznam věcí (`1,2,3,4,5,1000`) a nějaké tělo, co dělat s každou položkou v seznamu. To “tělo” dostane jednou `x` s hodnotou 1, jednou s hodnotou 2, a tak dále.

Seznam mujSeznam může samozřejmě získat svůj obsah složitějším způsobem. Může to být například výsledek nějakého výpočtu nebo po načtení ze souboru.

Řady (Range)

Nejobyčejnější druh cyklů se v Pythonu dělá dosti nešikovně. Často chcete udělat něco 10x (nebo 93x, nebo kolikrát chcete). Mohli byste říci`for x in [1,2,3,4,5,6,7,8,9,10]:` , ale určitě by vás to brzy přestalo bavit: ta spousta psaní a zabraného místa – a co když byste dokonce potřebovali 1000 opakování? Nebo když se počet opakování může měnit?

Naštěstí (uf!) můžete říci:

```
for x in range(10):
    print (x)
```

Toto vytiskne 10 čísel. Asi to ale neudělá způsobem, jaký byste očekávali. Posloupnost čísel `x` nebude 1,2,3,...,10; ale bude 0,1,2,...,9. Nicméně pořád je to 10x..

Mimochodem, range se nepoužívá jen ve smyčkách for. Dá se použít i jinak::

```
>>> print (list(range(5)))
[0, 1, 2, 3, 4]
```

Všechny cykly for mají tu vlastnost, že když začnou, tak počítač musí přesně vědět, jaký seznam bude procházet, musí znát, kde bude končit. Ale co když se dozvíme, kdy skončit až po startu cyklu?

Cykly while: Když nevíte kolikrát

Python má pro tento účel jiný druh cyklů. Nazývá se while (česky dokud):

```
cislo = 1
while cislo < 1000:
    print (cislo)
    cislo = 2*cislo
print ("tak to je konec")
```

Jakmile počítač uvidí while cislo < 1000:, udělá toto:

- Podívá se jestli cislo < 1000 je pravda nebo ne.
- Pokud ne, opustí cyklus: bude pokračovat čímkoliv, co následuje za koncem cyklu.
- Pokud je, půjde zpracovat obsah cyklu...
- ...a vrátí se zpět na začátek zjistit, jestli je již cislo > 1000 nebo ještě ne. Jinými slovy, vykonává obsah cyklu pořád dokola, ale jendokud (while) je podmínka number < 1000 pravdivá.

Můžete se podívat na Podmínky a podmíněné příkazy, kde najdete více informací o podmínkách, o pythonýrských představách o tom, co je to pravda a co lež (nepravda).

Opouštíme cyklus předčasně

Někdy potřebujete opustit cyklus "předčasně". Například máte cyklus for sčítající několik čísel ze seznamu, ale vy potřebujete přestat sčítat, když narazíte na nulu. (Proč? Nevím, je to jen příklad.)

Zde potřebujete příkaz break. Značí asi toto: skonči smyčku, ať si kde si. Takže výše uvedené zadání bude vypadat asi takto:

```
soucet = 0
seznam=[5,2,-5,155,0,36,1,2]
for x in seznam:
    if x == 0:
        break
    soucet = soucet+x

print ("Soucet je", soucet)
```

Příkaz `break` je obzvláště užitečný, když máte cyklus `while`, ale testovací podmínka se nevyskytuje hned na začátku cyklu. Dejme tomu, že chcete sečíst spoustu náhodných čísel a zastavit, když kterékoliv z těch čísel bude číslo 3. (Ano, je to divný příklad. Existuje spousta užitečnějších příkladů, ale všechny jsou delší a složitější.) Mohli byste to udělat třebas takto:

```
from random import randint
soucet = 0
while 1:
    nahodneCislo = randint(1,10)
    print (nahodneCislo)
    if nahodneCislo == 3:
        break
    soucet = soucet + nahodneCislo

print ("Soucet je", soucet)
```

Jediná podivná věc na tomto kousku kódu je ta `1` v příkazu `while 1:`. Jak vám vysvětluje Podmínky a podmíněné příkazy, Python považuje všechna nenulová čísla za pravdu. Takže `while 1:` značí: následující obsah dělej donekonečna nebo dokud nenarazíš na `break`.

Nedělejte si těžkou hlavu, jestli jste z toho zmateni. Asi to nebudete potřebovat.

Podmínky a podmíněné příkazy

Podmínky jsou věci, které mohou být pravdivé nebo nepravdivé. Například výraz `x <3` je pravdivý pro čísla menší než 3, jinak je nepravdivý.

Podmíněné příkazy jsou věci, které závisí na podmínkách. Nejdůležitějším podmíněným příkazem v Pythonu je příkaz `if`, který vám umožňuje dělat jednu nebo druhou věc v závislosti na tom, jestli podmínka je pravdivá nebo ne.

Tento list vám poví o podmínkách a o podmíněných příkazech.

Podmínky

Zkuste zadat následující příkazy. Neukazuji výsledek, protože se to lépe naučíte, když si to sami vyzkoušíte:

```
>>> 1<2   # 1 je menší než 2, proto je tato podmínka pravdivá
[CENZUROVÁNO]
>>> 1>2   # 1 není větší než 2, proto je tato podmínka nepravdivá
[CENZUROVÁNO]
```

True znamená pravda, False znamená nepravda, čili lež.

V podstatě můžete použít jakékoliv věci pro porovnávání. Můžete si to ostatně vyzkoušet. Zkuste mezi sebou porovnat texty, seznamy. Zjistíte sami princip, podle čeho Python rozhoduje co je pravda? Pokud nevíte, nevadí, někdy je to matoucí, pokračujeme dál..

Porovnání

Většina podmínek je porovnáním jedné věci s jinou. Zde je stručný seznam způsobů, jak v Pythonu porovnávat věci.

- a < b - pravda když a je menší než b
- a <= b - pravda když a je menší nebo rovno b
- a > b - pravda když a je větší než b
- a >= b - pravda když a je větší nebo rovno b
- a == b - pravda když a se rovná b
- a != b - pravda když a není rovno b

Je vcelku zřejmé, co tyto věci znamenají, když `a` a `b` jsou čísla. Ale oni dávají smysl i pro další druhy objektů. Řetězce se například porovnávají podle abecedy. Mimochodem, dávejte pozor na rozdíl mezi `=` a `==`. Znak `=` používáme pro nastavení proměnné (tj. objekt dostane jméno), a `==` pro testování, jestli dvě věci jsou stejné.

Kombinace porovnávání

Pro podmínku můžeme použít i něco jako `1 < x < 2`. Tato podmínka bude pravdivá, když číslo x bude větší než 1 a zároveň menší než 2.

Další podmínky

Zde je několik dalších užitečných podmínek:.

- 0 - vždy nepravda - odpovídá False, nebo také prázdnému řetězci “”
- 1 - vždy pravda - odpovídá True, nebo také jakémukoliv neprázdnému řetězci či číslu
- x in y - pravda, když x je rovno některé položce z y
- x not in y - pravda, když x se nerovná žádné položce z y

U posledních dvou podmínek, `in` a `not in`, by proměnná `y` měla být posloupnost: tedy seznam nebo entice nebo řetězec.

Kombinace podmínek

Podmínky můžete spojovat slovy `and`, `or` a `not` (česky: a zároveň, nebo, není). Tak například `x<3 and y>6` je podmínka, která je pravdivá, když číslo x je menší než 3 a zároveň číslo y je větší než 6..

Příkaz if (když)

Tak nyní již víme, co jsou to podmínky. Jsou velmi důležité, když je použijeme v příkazu `if`. Je to přímočaré:

```
if x < 3:
    print ("x je mensi než 3. Zmenim x na 3.")
    x = 3
```

Často potřebujeme udělat jednu věc, když je podmínka pravdivá a jinou věc, když není. K tomu máme kouzelné slovo `else`:

```
if x<3:
    print ("x je mensi než 3.")
else:
    print ("x neni mensi než 3.")
```

Méně často potřebujeme otestovat celou skupinu podmínek a něco dělat podle toho, která z nich se ukáže býti správná jako první. K tomu tu zase máme podivné slovo `elif`, což je zkratka pro `else if` (česky: jinak když):

```
if x<3:
    print ("x je mensi než 3")
elif x<4:
    print ("x neni mensi než 3, ale je mensi než 4.")
else:
    print ("x neni ani mensi než 4.")
```

Cykly¶

Podmínky se často používají v cyklech `while`: Chcete-li se něco o nich dozvědět něco bližšího, čtěte Cykly.

Chyby

(také nazývané výjimky, errory, bugy)

Když se něco pokazí, Python odpoví nějakou drsnou hláškou, která může vypadat i takto:

```
Traceback (innermost last):
  File <stdin>, line 1, in ?
  File <stdin>, line 1, in f
TypeError: illegal argument type for built-in operation

český překlad:

Zpětné trasování (poslední vnitřní)
  Soubor "<stdin>", řádka 1, v ?
  Soubor "<stdin>", řádka 1, v f
TypeError: neplatný argument pro vestavěnou funkci
```

Většinou můžete ignorovat vše kromě poslední řádky, ačkoliv předchozí řádky – pokud se na ně podíváte pozorně – dávají obyčejně nápovědu o tom, kde se tak chyba stala, což je často také velmi důležité. Tento list je stručný průvodce po běžnějších věcech, které můžete vidět na posledním řádku chybového hlášení a o tom, co jste udělali špatně, že jste ji vyprovokovali.

Chybové hlášky

Chyby atributu, Chyby klíče, Chyby indexu

Tyto hlášky začínají na `AttributeError:`, `KeyError:` nebo `IndexError:`. Všechny mají něco společného: snažili jste se získat neexistující části objektů (například položku seznamu). Když například `x` je seznam `[1,2,3]` a vy se budete ptát na `x[100]`, výsledkem bude Chyba indexu: `IndexError`.

Chyby jména

Všechny tyto začínají na `NameError:`. Znamenají: “Nikdy jsem neslyšel nic o této věci”. Ta “věc”, o které Python nikdy neslyšel, následuje za `NameError:`. Může to znamenat, že jste se přepsali, například: `imput (x)` namísto `input (x)`. Může to také znamenat, že jste použili proměnnou (tj. jméno), aniž byste ji předtím nadefinovali - např. `print (cislo)`, když jste Pythonu nikdy předtím neřekli, kolik je `cislo`. Nebo že jste použili funkci předtím, než byla nadefinovaná pomocí `def`.

Toto se stává také, když opomenete uvozovky kolem řetězce, například `print (Ahoj svete)`, protože Python se snaží vytisknout proměnnou `Ahoj`, která však neexistuje.

Syntaktické chyby

Všechny začínají na `SyntaxError:`. Pro tyto hlášky je společné, že Python ani nemohl začít číst příkazy, protože mu prostě nedávají smysl. Když někdo řekne Já jablko jíst rád to pro, jedná se syntaktickou chybu! Může to znamenat spoustu věcí. Pár příkladů:

```
+ + +          #A co sečíst s čím?
1z6           #Je to číslo? Nebo proměnná? JE to chyba?
for           #Po for by mělo něco následovat
while if:     #if není věc, která by mohla být pravda nebo nepravda!
def 1(x):     #1 je číslo. Nemůže se použít pro jméno funkce
f([)          #Znak [ začal seznam, ale nikdy jste ho neskončili
if 1==2       #Opomněli jste na : na konci řádku
if x=y:       #Mělo by tam být ==, ne =
```

Pokud zmotáte odsazování (mezery na začátcích řádků), můžete také dostat snad překvapující hlášku syntaktické chyby, takže pokud pokud se zdá být vše v pořádku, zkontrolujte ještě jestli sedí odsazování.

Typové chyby

Všechny tyto začínají na `TypeError:`. Pro tyto hlášky je společné to, že Python očekával nějaký druh objektu (dejme tomu že číslo) a vy jste mu dali jiný (například řetězec nebo seznam).

Mnoho z těchto chyb se stává ne zcela zřejmým způsobem. Pokud například předáte něco, co není číslo, některé z grafických funkcí –`create_oval()`, atd. – pak pravděpodobně dostane `TypeError`.

TypeError: illegal argument type for built-in operation

Žádali jste Python o vykonání nějaké vestavěné funkce se špatnými druhy objektů.

TypeError: len() of unsized object

Když chcete délku něčeho, co délku nemá. `len()` je smysluplné pro seznamy, řetězce a entice (není nyní nutno vědět co “entice” je) a to je skoro všechno.

TypeError: f() takes exactly 1 argument (0 given)

Snažíte se volat funkci, ale ta potřebuje více nebo méně argumentů, než jí posíláte.

TypeError: unsupported operand type(s) for +: ‘int’ and ‘str’

Tuto hlášku můžete také dostat, pokud se pokusíte sčítat čísla řetězce, např. `4+'och'`.

TypeError: unsubscriptable object

Snažíte se udělat něco jako `a[3]`, když `a` není ta správná věc. To se může stát třebas takto: `a=3; print (a[3])`.

Ošetření chyb

Python nám nabízí velmi mocný nástroj na zpracování chyb. Prostě něco, co nám umožní chybu v programu zachytit a zpracovat. A ne aby program skončil a vypsal, že se mu něco nelíbí. Příkaz , který toto zvládá má jméno try:.... except:..... Zkuste si jednoduchý příklad, dělení nulou:

```
>>> print (5/0)
ZeroDivisionError: integer division or modulo by zero # chyba při
dělení nulou
```

A nyní chybu zkusíme zachytit a ošetřit:

```
try:            # zkus vykonat, co je uvnitř
    print (5/0)
except:         #a při chybě pokračuj zde
    print ("Asi jste delili nulou, to neumim")
```

Užitečnější příklad na ošetření vstupu od uživatele, tedy když chceme, aby nám uživatel zadal skutečně číslo a ne nějaký znak nebo jen Enter, najdete na Vstup a výstup.

Funkce

Funkce je kousek pythonýrského programu, který má jméno. Funkce jsou velmi důležité; tento list vám o nich řekne více.

Výhody funkcí

Funkce je jako recept. Je to sada instrukcí, na které se odkazujete jménem. Použití funkcí usnadňuje přemýšlení ve stejném smyslu, jako je to u receptů; Když plánujete oběd, je o mnoho snadnější jednoduše říci Budeme mít boloňské špagety a jablečný koláč, než přesně do detailu říkat, jak se to vaří nebo co všechno je tam za přísady.

Funkce jsou užitečné, když potřebujete jednu věc udělat několikrát: jednou funkci nadefinujete (jako když napíšete recept) a pak ji několikrát použijete (jako upečení několika koláčů podle toho samého receptu).

Jsou ještě užitečnější, když chcete udělat téměř tu samou věc několikrát. Můžete napsat funkci, jejíž jednotlivé kousíčky nebudou úplně přesně nadefinované a detailně se vyplní, až když je funkce použita. Snad by se to dalo přirovnat k receptu, podle kterého se dají upéct různé druhy koláčů.

Funkce jsou také užitečné, přesto když každá z nich bude použita jen jednou. Váš program nebude rozhodně kratší, ale bude přehlednější, snadnější na pochopení. (Tady mi asi nebudete věřit, dokud si sami nenapíšete několik větších programů a pak to objevíte sami!)

Definice a použití funkcí

Definice jednoduché funkce (když ji zavoláte, dělá pokaždé to samé) může vypadat takto:

```
def mojeMalaFunkce():
    print ("7x8= ",7*8)
    print ('-'*40)
    print ("A to je konec")
```

Pokud ji chcete zavolat, řekněte jednoduše:

```
mojeMalaFunkce()
```

V podstatě se stane přesně to samé, jako kdybyste spustili samotné řádky uvnitř funkce.

Lokální proměnné

Řekněme, že napíšete:

```
def oblibenaHra():
    hra = 'Doom'
    print (hra, hra)
```

```
hra = 'StarCraft'
oblibenaHra()
print (hra)
```

Na konci tohoto prográmku v proměnné hra bude stále 'StarCraft' – a ne 'Doom', jak byste snad čekali. (Je to tedy úplně něco jiného, než kdybyste funkci vypustili, její obsah prostě zkopírovali do prográmku a spustili ho.) Vypadá to, jakoby zde existovali dvě proměnné se stejným jménem hra a v každé bylo uloženo něco jiného. Ano, v podstatě to tak je. Ta jedna platí jen uvnitř funkce, někdy říkáme jen po dobu života funkce. Ta druhá hra se nepřepíše, je prostě jinde, v jiném "prostoru", a proměnná hra z funkce na ní nemá vliv.

Je pro to dobrý důvod. Značí to, že funkce nemohou mít šílené neočekávané důsledky. Kdo by četl jen druhou (hlavní) část programu, neměl by žádný důvod očekávat, že funkce oblibenaHra() změní nějak hodnotu proměnné hra . Takže jakékoliv "lokální" změny proměnných uvnitř funkce nemají vliv na hlavní část programu, a je tak jednodušší mu rozumět: nepotřebujte přesně vědět, co se děje uvnitř funkce, abyste mohli pochopit hlavní část programu.

Proměnná hra uvnitř funkce oblibenaHra() se nazývá "lokální proměnná".

Globální proměnné

Vlastně jsem lhal, když jsem říkal, že funkce nemůže mít neočekávané důsledky na hodnoty proměnných. Může, ale nejdřív o to musíte požádat. Když přidáme k výše uvedenému prográmku řádku:

```
def oblibenaHra():
    global hra          # Tato řádka je navíc
    hra = 'Doom'
    print (hra, hra)

hra = 'StarCraft'
oblibenaHra()
print (hra)
```

pak se hodnota proměnné hra změní. Příkaz global hra znamená: Proměnná hra není lokální, bude ukazovat na stejné místo, jako ta hra z 'vyššího patra'. (Globální je opakem k lokální.). Takže když změníme proměnnou hra uvnitř funkce, změníme i tu druhou proměnnou hra, kterou tím vlastně přepíšeme.

Neměli byste ale toto používat hodně často hlavně proto, že platí následující pravidlo: Proměnná se považuje za globální, pokud ji funkce jen používá (čte),ale nemění její hodnotu (nezapisuje). Takže vaše funkce může používat proměnou z hlavní části programu bez starostí se psaním global ; pouze pokud tuto proměnnou budete chtít uvnitř funkce změnit, musíte říci, že je global . Obvykle by to ale neměla být moc potřeba.

Funkce s argumenty

Nejzajímavější funkce mají “argumenty”. Argument je kousek informace, kterou funkci předáváte při volání a která méně nebo více mění průběh funkce. Když máte recept, který vám říká, jak upéct libovolný počet palačinek (použijte 0,5 litru mléka na 12 palačinek , apod.), pak počet palačinek je argumentem pro recept. (ačkoliv kuchařky to tímto způsobem neuvádějí!).

Tady je, jak nadefinovat funkci s argumenty:

```
def tiskniDveVeci(x,y):
    print ('x je', x)
    print ('y je', y)
```

Pravděpodobně uhodnete, že když řeknete `tiskniDveVeci(1,'bing!')`, tak počítač vytiskne:

```
x je 1
y je bing!
```

Argumenty funkcí jsou ve skutečnosti speciálním druhem lokálních proměnných: uvnitř funkce `tiskniDveVeci()` jsou `x` a `y` lokální proměnné, jejíž hodnoty jsou zadány, když zavoláte funkci `tiskniDveVeci(1,'bing!')`.

Funkce vracející hodnotu

Některé funkce nejen že věci jen dělají; některé také vracejí výsledek. Můžete například napsat funkci, která přijímá 2 argumenty a dává, jako výsledek součet těch 2 argumentů. (Není žádný zvláštní důvod, proč byste to měli dělat, je to jen jednoduchá ukázka.)

Uděláme to takto:

```
def secti(a,b):
    celkem=a+b
    return celkem
```

Kouzelné slovo `return` znamená vrať jako výsledek, to co je za mnou. Takže funkci můžete použít následovně:

```
print ('3 + 4 =', secti(3,4))
```

Výsledek bude překvapivě `3 + 4 = 7` .

Dobrovolné argumenty (pojmenované argumenty)

Někdy jsou argumenty předávané funkci často při voláních stejné, jen zřídka je chcete měnit. Pomocí “dobrovolných argumentů” můžete ušetřit psaní kódu. Jak to udělat:

```
def tiskni(x, y=999):
```

```
    print (x,y)
```

Funkci můžete použít třemi způsoby.

- Jako obyčejnou funkci se dvěma argumenty. `tiskni(5,6)` funguje úplně stejně, jako kdyby definice zněla `def tiskni(x,y):` .
- Jako funkci s jedním argumentem. Pokud neřeknete, kolik má být hodnota `y`, tak bude nastavena na 999 po dobu trvání funkce. Takže`tiskni(1)` je to samé jako `tiskni(1,999)` .
- Jako funkci s jedním argumentem a jedním "pojmenovaným argumentem" (keyword argument), který vyvoláte jménem. Například`tiskni(8, y=123)`. Vypadá to, že to moc k ničemu není, ale když `tiskni` má spoustu dobrovolných argumentů, dává vám toto použití možnost přehledně změnit jen hodnotu vybraného argumentu, přičemž ostatní zůstávají beze změny.

Proč se tím trápit?

Je možné psát dosti složité programy bez trápení se s funkcemi. Co vám tedy použití funkcí přináší?

- Vaše programy budou jasnější . Je snadnější rozumět programu, který je rozdělen do zvládnutelných dávek. A funkce jsou dobrým způsobem, jak toho dosáhnout.
- Nebudete muset tak usilovně přemýšlet. Pokud je váš program citlivě rozdělen do funkcí, stačí si vám pamatovat, jak přesně funkci použít, bez nutnosti do detailu znát vnitřek funkce. Když uvidíte funkci `aktualizujNejvyssiSkore()`, budete snad již z názvu vědět, co to dělá, bez nutnosti probírat se kódem funkce, která aktualizaci provádí, takže bude rychlejší pochopit ten kousek programu, který`aktualizujNejvyssiSkore()` volá.
- Ušetříte psaní kódu.. Když musíte dělat tu samou věc (nebo téměř tu samou věc) několikrát, dejte ji do funkce. Pak vám stačí vysvětlit pouze jednou, jak to udělat.
- Váš program bude více přizpůsobitelný. Pokud zjistíte, že potřebujete v programu něco opravit, je to o mnoho snadnější, když je úhledně zapakován do funkcí. Jinak budete mít noční můry z toho, jestli jste skutečně našli každé místo v programu, kde se pracuje se skóre hráče a nastavovat ho podle nových pravidel.
- Budete moci znovu používat co jste napsali. S kouskem štěstí zjistíte, že některé vaše funkce jsou použitelné pro různé věci, takže můžete použít tu samou funkci (nebo nepatrně upravenou) v několika programech. To vám ušetří námahu oproti tomu, kdybyste to museli psát několikrát! Například funkce ošetření vstupu, viz Vstup a výstup.

Pravděpodobně nevidíte smysl spousty věcí z tohoto listu, dokud vy sami si nenapíšete (nebo nepřečtete!) nějaké složité programy. Jakmile to ale uděláte, objevíte že funkce jsou základem pro udržování velkých programů.

Moduly

Python nabízí ohromné množství užitečných funkcí a dalších podobných věcí. (Tyto listy popisují snad něco kolem 1% toho, co je dostupné. Možná 2%.) Tak mnoho rozdílných funkcí může být matoucí, proto byly rozkouskovány do věcí, kterým se říká “moduly”. Moduly jsou vlastně vyšší forma funkcí.

Importování modulů

Moduly si můžete představit jako košík plný užitečných věcí. (Skutečná definice je trochu složitější a nemusíte ji znát.) Když se chcete dostat k těm Užitečným věcem z modulu, musíte ho “naimportovat”, “vtáhnout” do programu. Takhle:

```
import vetes
```

se zpřístupní pro váš program věci z modulu `vetes`. Ty věci pak budou mít jména jako `vetes.kolo` proto, aby nebylo nebezpečí, že u dvou věcí se stejnými jmény ze dvou různých modulů nastane “kolize” jmen. Je to jako s telefonními čísly: určitě existuje spousta lidí v Evropě , kteří mají stejné telefonní číslo; proto musí existovat “předvolby států”, kdy se před číslem vytočí číslo toho příslušného státu a dovoláte se správně. Jméno modulu je tedy jako předvolba státu.

Žití je nebezpečné

Někdy z toho můžete být dost otrávení, psát pořád dokola jméno modulu. Python vám dává možnost získat věci z modulu přímočařeji. Když řeknete:

```
from vetes import napinacek, zarovka
```

potom věci, které by se normálně jmenovali `vetes.napinacek` nebo `vetes.zarovka` jsou dostupné pod kratšími jmény `napinacek` a `zarovka`. (Tímto nenaimportujete z modulu nic jiného, žádné jinou věc.)

Může to být prospěšné, když některou část modulu budete používat velmi často.

Žití ještě nebezpečnější

A co když používáte spousty věcí z modulu často? Pak můžete říci:

```
from vetes import *
```

čímž naimportujete všechno (dobře, téměř všechno) z modulu `vetes` do vašeho programu, aniž byste pak při používání těchto věcí museli psát předponu `vetes.` na začátku každého jména. Toto je obecně špatný nápad; mohou existovat dva moduly, které obsahují různé věci pod stejnými jmény a

pak se dějí skutečně ošklivé věci. S výjimkou grafických modulů ala Tkinter se toto nedoporučuje dělat.

Co je to v tom modulu?

Jakmile napíšete `import vetes`, můžete získat seznam všech jeho funkcí (všech věcí) napsáním `dir(vetes)` nebo dokonce dokumentaci, nápovědu (pokud taková v modulu existuje) příkazem `help(vetes)`. Asi to nebude obvykle moc k užitku, obzvlášť, když je všechno v angličtině, ale když chcete prozkoumávat ....

Pár užitečných modulů

Jak jsem již řekl v úvodu, tam někde venku je spousty modulů. Je mnoho modulů, které přichází v samotné instalaci Pythona! Pokud jste odvážní a zvládáte angličtinu, můžete se na některé podívat. Zde nabízíme jen letmý přehled. Podívejte se na ně v oficiální dokumentaci k Pythonu. Abyste rozuměli některým věcem z dokumentace, bude asi třeba něco vědět o Objektové programování.

sys

: Pár většinou užitečných proměnných o systému

os

: Téměř vše, co souvisí s OS: např. kopírování souborů a složek, cesty k souborům

math

: Matematické funkce a konstanty

random

: Náhodná čísla

time

: Datum a čas.

Objektové programování

Když používám v těchto listech slovo objekt, tak většinou ve smyslu jakákoliv věc, se kterou Python může pracovat. Takže číslo je objekt, a taky řetězec, seznam nebo slovník. Vše je objekt.

Ale existuje ještě jiný význam slova objekt. Možná jste slyšeli slova objektově orientované programování. Jedná se o jiný způsob psaní programů, jiný způsob přemýšlení, který je obzvlášť efektivní pro velké a složité programy. Python vám umožňuje dělat objektově orientované programování a jak je zvykem, jeho objekty jsou vhodné dokonce i pro celkem malé programy.

Tento list je celý o těchto objektech v Pythonu. Pro začátečníky je to velmi dobrovolné. Pro tvorbu her v Pythonu je samozřejmě základem vědět, jak tyto objekty fungují.

Instance a třídy

Podstatou objektu je v Pythonu (a v některých dalších programovacích jazycích také) třída. Věci, které se z těchto tříd vyrábí, se nazývajíinstance.

Když v Pythonu vytváříte třídu, říkáte Pythonu co mohou dělat instance, zrozené z té třídy. Třída je jako forma na lisování instancí, které dělají určité věci. Věci, které instance z určité třídy může dělat, se nazývají metody. Metody jsou jako funkce, které žijí uvnitř instance.

Vyzkoušejte následující script:

```
class Tiskarna:
    def tiskniNeco (self, co):
        print ("Tento objekt říká", co)
```

Právě jste vytvořili třídu s názvem Tiskarna. Instance, které budou patřit do třídy, budou zatím umět dělat jen jednu věc - tiskniNeco, neboť jsme vytvořili jen jednu metodu s názvem tiskniNeco. Program spusťte a na následně na promptu vyzkoušejte vytvořit:

```
>>> mujTisk = Tiskarna ()
```

To bylo jednoduché. Nyní je mujTisk instancí ze třídy Tiskarna. Věci ze třídy Tiskarna umí dělat tiskniNeco. Vyzkoušejte toto:

```
>>> mujTisk.tiskniNeco ("Ahoj vy tam")
```

Tečka se používá, když se chcete dostat k věcem, které žijí uvnitř určitého objektu, takže mujTisk.tiskniNeco používá metodu tiskniNecouvnitř instance mujTisk. Instance mujTisk má metodu tiskniNeco, protože jsme ho vytvořili ze třídy Tiskarna. Zkuste přimět mujTisk, aby říkal i jiné věci. Zkuste vytvořit více instancí ze třídy tiskárna s různými jmény a přimět je, aby také mluvili.

Proměnné v metodách

Zkusme něco jiného. Vytvořte další třídu s názvem LepsiTiskarna:

```
class LepsiTiskarna:        #definice třídy s názvem    LepsiTiskarna
    def nastavTisk (self, co):
        self.vytisk = co
    def tiskniTo (self):
        print (self.vytisk)
```

Úkol: Následně zkuste vytvořit instanci této třídy LepsiTiskarna. Nazvěte ji razitko. Rada: je to podobné jako příklad, který jsme již psali, když jsme vytvářeli instanci patřící třídě Tiskarna.

razitko patří třídě LepsiTiskarna, takže ví, jak udělat nastavTisk, a také tiskniTo, protože to jsou dvě metody, které jsme vytvořili pro Instance z LepsiTiskarna. Zkuste toto:

```
>>> razitko.nastavTisk ("Ahoj vy tam")
```

Co si myslíte že se stane, když řekneme razitku tiskniTo? Zkuste to a uvidíte:

```
>>> razitko.tiskniTo ()
```

Zkuste změnit co razitko tiskne pomocí metody nastavTisk. Můžete říci razitku tiskniTo více než jednou bez nutnosti měnit to, co má tisknout.

Zkuste vytvořit některé další instance patřící třídě LepsiTiskarna. Když změníte, co má tisknout jedna instance, změní to i u jiné instance? Vyzkoušejte to.

Když Python spouští metodu, nastavuje self na instanci, které ta metoda patří, takže se snadno dostaneme k metodám a proměnným instance, které žijí uvnitř něho. Všechno ostatní co dáváme do závorek, když spouštíme metodu, se uloží do proměnné, kterou jsme uvedli zaself, když jsme definovali metodu. Takže když řekneme:

```
>>> razitko.nastavTisk ("Ahoj vy tam")
```

spustí se metoda nastavTisk a přiřadí do self razitko a do proměnné co přiřadí Ahoj vy tam. (Podívejte se na místo, kde jsme definovali třídu LepsiTiskarna: uvidíte tam ty jména).

Každá instance patřící ke třídě LepsiTiskarna ukládá, co jí řeknete, do proměnné vytisk, aby to později vytiskla. self.vytisk znamená: všechny instance, které vzniknou ze třídy LepsiTiskarna mají každá svůj vytisk`, takže:

```
self.vytisk = co  # Toto je řádka z metody nastavTisk.
```

znamená:

```
razitko.vytisk = "Ahoj vy tam"
```

Můžete si to prověřit sami, protože Python vás neomezuje, abyste se dívali na jednotlivé objekty vytisku přímo. Zkuste toto:

```
>>> print (razitko.vytisk)
```

a uvidíte, jestli se vám podařilo dostat k razitkovu vytisku bez použití metody tiskniTo. Zkuste nastavit proměnnou vytisk bez použití metody nastavTisk, a pak použít metodu tiskniTo, abyste se ujistil, že jste skutečně změnili to, co razitko tiskne.

Můžeme tedy přistupovat a dokonce i měnit proměnné instance přímo bez použití metod nastavTisk a tiskniTo Obecně to ale není dobrý nápad, přistupovat takto přímo k proměnný třídy z vnějšku. Představte si, že vytváříte třídu, kterou budou používat také další programátoři: budete asi chtít, aby uměli používat vaši třídu, i když změníte její vnitřní fungování, přičemž navenek, přes svoje metody se nezmění.

Dá se to udělat tak, že stanovíte jména metod, argumentů, které přijímají a co vracejí. Tyto věci se pak předkládají ostatním uživatelům k dispozici a často se jim také říká rozhraní (interface), API. (ve spisovné angličtině je interface prostě místo, kde se dvě různé věci stýkají, takže to dává i nějaký smysl).

Některé lidé si ale myslí, že mohou fušovat do vnitřku vaší třídy - pak jim ale přestanou fungovat jejich programy, pokud ty vnitřky změníte. Představte si, že jsme se rozhodli změnit jméno vytisk na kopie. Program, který používal vytisk místo přístupu přes metody nastavTisk atiskniTo najednou přestane fungovat! Naproti tomu program, který ty dvě metody používal (a nesnažil se používat vytisk přímo), bude fungovat normálně dál, protože jsme schovali řízení třídy dovnitř těch dvou metod. Toto ještě získává na důležitosti, když pracujete na skutečně velkých programech nebo spolupracujete s ostatními na velkém programu.

Dědičnost

Zkusme něco jiného:

```
class LepsiTiskarna:
    def nastavTisk (self, co):
        self.vytisk = co
    def tiskniTo (self):
        print (self.vytisk)

class JesteLepsiTiskarna (LepsiTiskarna):
    def pridejTo (self, co):
        self.vytisk = self.vytisk + co
```

Právě jste vytvořili novou třídu nazvanou JesteLepsiTiskarna.

Úkol: Řekněte Pythonu, aby vytvořil instanci z této třídy s názvem erazitko. Teď zkuste toto:

```
>>> erazitko.nastavTisk ("zdravime pozemstany")
>>> erazitko.tiskniTo ()
```

Co se stalo? Mělo by vám fungovat, že obě dvě metody - nastavTisk a tiskniTo - mají instanci erazitko. Ale vždyť jsme je ve třídě JesteLepsiTiskarna nedefinovali, tak co se tu děje? Odpověď leží v prvním řádku definice třídy JesteLepsiTiskarna

```
class JesteLepsiTiskarna (LepsiTiskarna):
    ....
```

To říká Pythonu, že JesteLepsiTiskarna je částí třídy LepsiTiskarna, a může dělat vše, co dokáže LepsiTiskarna. Takže erazitko umí metody nastavTisk a tiskniTo, protože LepsiTiskarna je definuje. Ale ve třídě JesteLepsiTiskarna je více než ve třídě LepsiTiskarna. Zkuste toto:

```
>>> erazitko.pridejTo (", zaved mne ke svemu vedoucimu.")
>>> erazitko.tiskniTo ()
```

Co se stalo? Podívejte se nahoru na definici metody pridejTo třídy JesteLepsiTiskarna a zkuste na to přijít sami, co se stalo. Zeptejte se, pokud si nevíte rady. Zkuste vytvořit další instance ze třídy JesteLepsiTiskarna a hrajte si s našimi třemi metodami, které instance mají.

A co LepsiTiskarna? Vědí instance této třídy o metodě pridejTo? Zkuste vytvořit ještě jednu instanci z LepsiTiskarny a zkuste, jestli bude znát pridejTo. Náš výtvor vypadá asi takhle (používám obdélníky pro třídy a ovály pro metody):

JesteLepsiTiskarna je podtřídou LepsiTiskarny (na obrázku nahoře, je pod ní: představte si to snad jako ponorku pod vodou). Nebo jinak:LepsiTiskarna je nadtřídou (superclass) JesteLepsiTiskarny (Snad byste si mohli představit Supermana, létajícího nad ní! A nebo taky ne).

Viděli jsme, že instance z JesteLepsiTiskarny mají metody, která má i LepsiTiskarna, a extra metodu pridejTo, kterou jsme přidali do třídyJesteLepsiTiskarna. Příhodné jméno pro toto je dědičnost, protože JesteLepsiTiskarna je jako dítě třídy LepsiTiskarna a dědí tyto metody od ní.

Zkuste vytvořit další podtřídy LepsiTiskarny a instance patřící těmto třídám. Co se stane, když vytvoříte třídu, která bude dědit zLepsiTiskarny, ale která má svojí vlastní metodu tiskniTo, která dělá něco jiného než ta z LepsiTiskarny? Říkáme, že metody dětí jsou vždy přednější než metody rodičů, takže se vždy zavolá první třída toho jména, která je nejblíž z místa, odkud tuto metodu voláme.

Takže co z toho plyne? Jak jsme již řekli dříve, dědičnost je cesta, jak říci že A je částí B. Možná budeme psát hru, kde všechny instance na obrazovce budou mít něco společného: všechny mají rychlost a pozici, musí reagovat na zásah od jiných instancí a tak dále. Ale také jsou mezi nimi rozdíly: loď hráče muže například vystřelovat kulky, ale asteroidy střílet neumí.

Můžeme tedy udělat třídu pro objekt na obrazovce, který bude umět tyto běžné věci, abychom si ušetřili vypisování těch samých metod pořád dokola. Všechny objekty na obrazovce ale nejsou stejné, takže uděláme podtřídy naší třídy objekt na obrazovce, abychom mohli zpracovávat věci, které jsou pro každý druh objektů rozdílné. Hráč, kulka a asteroid mají za předka, za

rodiče objekt na obrazovce, takže našimi slovy jsou podtřídou objektu na obrazovce.

Kouzelné metody

Je ještě jedna věc, kterou potřebujeme znát, abychom pochopili, jak třídy v Pythonu fungují. Porozumíme pak i pracovním listům, které je používají. Udělejme další podtřídu naší staré známé: LepsiTiskarna:

```
class SZTiskarna (LepsiTiskarna):
    def __init__ (self, co):
        self.vytisk = co
        print ("Narodila se instance ze třídy SZTiskarna.")
        print ("Hurááááááá, jsem živá!")
```

Zkuste toto:

```
>>> mojeTiskarna = SZTiskarna ("Bonjour")
```

Co se stalo? Funguje to jak to funguje, protože metoda __init__ má v Pythonu speciální význam (init je zkratka pro inicializaci, což je oblíbené jméno pro věci, které děláme) když je se děje něco úplně poprvé. Metoda __init__ se spouští pokaždé (pokud ji tedy třída obsahuje), když se vytváří nová instance. Takže když jsme vytvořili instanci mojeTiskarna ze třídy SZTiskarna, metoda __init__ se spustila a vytiskla svou zprávu.

Asi jste si všimli, že tentokrát je něco jinak, než tomu bylo předtím. Když jsme vytvářeli mojeTiskarna, umístili jsme cosi do vnitřku závorek. Když to takto uděláme, metoda __init__ převezme tuto hodnotu a vloží ji do proměnné, kterou jsme definovali v __init__ v závorce za self. Uvnitř __init__ se tedy do proměnné co přiřadil řetězec Bonjour. Podívejte se na definici __init__. Co dělá s proměnnou co?

__init__ je užitečný, protože nám dovoluje nastavit věci v instanci tak, jak je potřebujeme k dalšímu využití v programu. Kdybychom například psali na vesmírnou hru, dali bychom do __init__ startovní umístnění instancí na obrazovce.

Existují také další speciální metody v Pythonu, ale zatím vám nemusí dělat starosti. Jen abyste věděli, že vždy začínají na __. Bude to vše OK, pokud nebude začínat jakákoliv z vašich metod na __, jestli to ovšem neučiníte záměrně.

Závěr

Co jsme se naučili? Měli byste znát:

- Jak vytvořit třídu a definovat jí metody.
- Jak vytvořit instanci té třídy a použít její metody.
- Jak nastavit proměnné, které žijí uvnitř instance.
- Jak vytvořit podtřídu z jiné třídy.

- jak použít metodu `__init__`.

Pokud si nejste jisti jakékoliv z těchto věcí, vraťte se a hrajte si s vytvářením tříd a instancí a zeptejte se, pokud potřebujete. Jakmile to pochopíte, můžete začít používat třídy a instance ve svých vlastních programech.

Hantýrka

S počítači přichází mnoho nezvyklých slov a mnoho normálních slov se používá nezvyklým způsobem. Snažili jsme se v těchto listech nepoužívat mnoho pravé počítačové mluvy, ale ne vždy se nám to dařilo. Tento list má tedy poskytnout krátký přehled o tom, co tyto nezvyklá slova znamenají. Pokud narazíte na něco, čemu nerozumíte, stojí za to rychle mrknout na tento list.

Něco z této hantýrky je o programování obecně. Něco je speciálně o Pythonu. Mnoho z těchto záznamů se odkazuje na jiné záznamy, takže když zde narazíte na slovo, kterému nerozumíte, ihned si ho vyhledejte!

Hantýrka

argument

Objekt předávaný funkci, když ji voláte. Objevuje se uvnitř funkce jako lokální proměnná.

asociace

Kousek informace, který říká Pokud dostaneš x, vrať y. Slovník je vytvořen z asociací.

tělo

Kód uvnitř cyklu nebo funkce nebo jiné komplikovanější věci. Takže tělo funkce je to, co se bude provádět, když zavoláte funkci a podobně "tělo cyklu" je to, co se provádí pokaždé při průchodu smyčkou.

volat

"Volat" funkci znamená použít ji. Volání funkce je trochu jako zkopírování její definice a provedení jejího těla.

třída

Třídy jsou druhy objektů. (Ve skutečném světe věci jako "počítač" a "žena" a "kniha" jsou třídy.)

znak

Písmeno, číslice nebo jiný symbol. Řetězce jsou složeny z posloupnosti znaků.

kód

Hmota, ze které je program udělán. "Kousek kódu" je část programu.

komentář

Kousek programu, který nic nedělá, ale je jen na čtení člověkem. Obvyklý smysl komentářů je vysvětlovat, co dělá kód v blízkosti.

podmínka

Něco, co může buď pravda nebo nepravda, jako `1>2`.

podmíněné příkazy

Něco, co závisí na podmínce, jako příkaz `if`.

definice

Kousek kódu, který říká počítači, co je něco: jakou hodnotu má proměnná, co má funkce dělat nebo vše o třídě.

slovník

něco jako adresář nebo encyklopedie. Slovník se skládá z asociací.

položka

Člen v posloupnosti.

výjimka

Neobvyklá situace, kdy se něco pokazí. Honosný způsob, jak jinak říci “chyba” nebo “error”.

výraz

Výpočet, který plodí hodnotu. Nejjednoduššími výrazy jsou věci jako konstanty (`123`, `'eek'`) a proměnné (`mojeJmeno`). Mohou být také velmi komplikované: `17*a[a.index(func(99))]>93 and a/(b+c)==2` .

cyklus for

Cyklus, jejíž tělo se provádí jednou pro každou položku v posloupnosti.

funkce

Sada příkazů, která má jméno. Počítač vykoná všechny instrukce funkce, když napíšete jméno funkce včetně závorek a případných argumentů.

globální proměnná

Proměnná, která existuje kdekoliv ve vašem programu, a ne je uvnitř jedné funkce.

grafika

Obrázky malované počítačem.

neměnné

Není dovolena změna.

import

Umožnit použití věcí nějakého modulu.

instance

Objekt popsaný třídou. V reálném světě to jsou konkrétní počítače, konkrétní ženy a konkrétní, jednotlivé knihy.

opakování (iterace)

Jeden průchod tělem cyklu. Když se vykonává cyklus, obvykle se několikrát opakuje.

klíč

Jedna polovina asociace ve slovníku: Ta polovina, podle které můžeme pak věci vyhledávat. Když řeknete `slovnik['zog']=123`, pak klíč je `zog`.

seznam

Objekt vytvořen z několika jiných objektů, popořadě. Získat je můžete podle čísla (pořadí).

lokální proměnná

Proměnná, která existuje jen uvnitř konkrétní funkce, a která neovlivňuje nic vně funkce

cyklus (smyčka)

Kousek programu, který se může vykonávat pořád dokola.

modul

Hromada objektů se jmény, které můžete “naimportovat” do vašeho programu.

objekt

Kousek informace, se kterým může Python pracovat, jako třebas čísla nebo řetězce nebo seznamy.

návratová hodnota

To, co vrátí funkce, když skončí. Když vrací hodnotu, může se použít ve výrazech.

posloupnost

Seznam nebo entice. (Ve skutečnosti jsou ještě jiné posloupnosti, ale není nyní třeba se jimi zabývat.)

řetězec

Posloupnost znaků, obvykle ve formě kousku textu.

entice

Zvláštní druh posloupnosti, zapisovaný jako: `(1,2,3)`. Narozdíl od seznamů jsou entice neměnné.

proměnná

Jméno pro objekt. Po napsání `x=6`, `x` je jméno pro objekt obvykle známý jako `6`, a použití `x` znamená obvykle to samé jako by znamenalo použití `6`. `x` se říká “proměnná”, protože se může proměňovat její obsah. Tedy místo `6` tam může být za chvíli třeba `8`, když o to Python požádáme.

Poznámky překladatele

Tento tutoriál s cizokrajným názvem Buwralug je postaven na základech původního překladu Livewires. Hlavní změny jsou dvě: původně použitý modul Livewires byl nahrazen obyčejným Tkinterem a je použit Python 3.0 narozdíl od původního Python 2.5. Pojetím se zaměřuje na mládež a absolutní začátečníky v programování. Odpovídá tomu “hantýrka” i použité příklady.

První vydání/překlad byl vytvořen o Vánocích 2004. O těch Vánocích, kdy se mírně naklonila zemská osa a změnil se čas. V létě 2008 jsem převedl dokumentaci z obyčejného html na Sphinx, kdy byly provedeny také mírné částečné úpravy. Na Vánoce 2008 jsem přešel na Python 3.0 a Tkinter a nový název Buwralug

With kind permission based on the LiveWires package.

Python

Python můžete stahovat z http://www.python.org/download/.
České webové stránky věnující se Pythonu: http://www.py.cz
Originální oficiální tutoriál: http://docs.python.org/3.0/tutorial/index.html
Plná originální pythonýrská
dokumentace: http://docs.python.org/3.0/library/index.html

Tkinter

Tkinter je modul (knihovna) pro programovací jazyk Python, která umožňuje v Pythonu vytvářet okna. Prostě grafické okna jak je na ně každý zvyklý z Windows či KDE/GNOME. Tkinter se dodává společně s instalací Pythona a je tak dostupná všude, kde je Python. Funguje stejně na na Linuxu i Windows, takže váš program funguje bez jakýchkoliv změn tam i tam.

Dokumentace v češtině: http://tkinter.programujte.com. Jedno malé upozornění: tato dokumentace platí i pro Python 3.0 přesto, že byla vytvořena pro Python 2.5. Jen si např. u příkazu print () musíte všude doplnit závorky. Např:

```
print x, y    ---------->    print (x, y)
```

Home

Tato učebnice žije na http://www.geon.wz.cz. Vítám podnětné připomínky, návrhy či opravy. Pavel Kosina (geon at post dot cz).

Dotazy

Dotazy směřujte např. do konference http://www.py.cz/KonferenceDiskuze.

Licence