Let's create a **step-by-step beginner-friendly guide** to learning **Python Django** on a **Windows PC**.

1. What is Django?

- **Django** is a high-level **Python web framework** that encourages rapid development and clean, pragmatic design.
- It helps you build **websites and web applications** quickly, without reinventing the wheel.
- Key features:
  - Object-Relational Mapping (**ORM**) to interact with databases using Python code.
  - URL routing to map URLs to functions or views.
  - Templates for dynamically generating HTML.
  - Built-in admin interface.
  - Security features (against SQL injection, XSS, CSRF, etc.).

---

2. Setting Up Your Environment on Windows

Step 1: Install Python

1. Go to Python official website.
2. Download the latest **Python 3.x** installer for Windows.
3. During installation:
   - Check **"Add Python to PATH"**
   - Choose **Install Now**.

 Tip: Open Command Prompt and type:

```
python --version
```

You should see something like `Python 3.12.x`.

Step 2: Install a Code Editor

- Recommended: **Visual Studio Code (VS Code)**
- Download from VS Code website.
- Install Python extension in VS Code for code highlighting and debugging.

Step 3: Install Django

1. Open Command Prompt.
2. It's best to use a **virtual environment**:

```
python -m venv myenv
myenv\Scripts\activate
```

You should see `(myenv)` before the prompt.

3. Install Django:

```
pip install django
```

4. Verify installation:

```
django-admin --version
```

---

3. Creating Your First Django Project

1. Open Command Prompt in your desired folder.
2. Create a new project:

```
django-admin startproject mysite
```

3. Navigate to your project folder:

```
cd mysite
```

4. Run the development server:

```
python manage.py runserver
```

- Open your browser and go to `http://127.0.0.1:8000/`
- You should see the **Django welcome page**.

---

4. Understanding Django Project Structure

```
mysite/
├─ manage.py        # Command-line utility to interact with project
├─ mysite/
│  ├─ __init__.py
│  ├─ settings.py   # Project configuration
│  ├─ urls.py       # URL routing
│  ├─ asgi.py       # ASGI entry-point
│  └─ wsgi.py       # WSGI entry-point
```

 Tip: Treat `mysite/` as the **project container**, and you'll later create **apps** inside it.

---

5. Creating Your First App

1. Apps are like modules in Django.
2. Create an app:

```
python manage.py startapp blog
```

- Folder structure:

```
blog/
├─ admin.py
├─ apps.py
├─ models.py
├─ tests.py
├─ views.py
└─ migrations/
```

3. Register the app in `mysite/settings.py` under `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...,
    'blog',
]
```

---

6. Creating a Simple View and URL

1. In `blog/views.py`:

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello, Django Beginners!")
```

2. In `mysite/urls.py`:

```
from django.contrib import admin
from django.urls import path
from blog import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.home),  # Maps home page to our view
]
```

3. Run the server again:

```
python manage.py runserver
```

- Go to `http://127.0.0.1:8000/` → You should see **"Hello, Django Beginners!"**

---

7. Templates in Django

1. Create a folder `templates` inside the `blog` app:

```
blog/
└─ templates/
    └─ home.html
```

2. In `home.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title>My Django Site</title>
</head>
<body>
```

```
    <h1>Welcome to Django!</h1>
</body>
</html>
```

3. Update `views.py`:

```
from django.shortcuts import render

def home(request):
    return render(request, 'home.html')
```

- Refresh your browser to see the template rendered.

---

8. Using Models and Database

1. Open `blog/models.py`:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
```

2. Apply migrations:

```
python manage.py makemigrations
python manage.py migrate
```

3. Register model in `blog/admin.py`:

```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

4. Create a superuser to access admin:

```
python manage.py createsuperuser
```

- Login at `http://127.0.0.1:8000/admin/` to manage posts.

Let's dive into **Django Forms**. Forms are essential for accepting user input, like creating a blog post, registering a user, or submitting comments. I'll explain **step by step**, with examples that a beginner can follow on Windows.

---

1. What Are Django Forms?

Django forms are Python classes that let you:

- Render HTML form elements.
- Validate user input.
- Save data to the database (optional).

There are **two main types**:

1. **Form** – general-purpose forms not tied directly to a database.
2. **ModelForm** – a form automatically linked to a Django model, so it can create or update records easily.

---

2. Creating a Simple Form

Step 1: Create a basic form

In `blog/forms.py`:

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=50, label="Your Name")
    email = forms.EmailField(label="Email Address")
    message = forms.CharField(widget=forms.Textarea, label="Message")
```

- `CharField` → text input
- `EmailField` → validates email format
- `TextArea` → multi-line input

Step 2: Create a view to handle the form

In `blog/views.py`:

```
from django.shortcuts import render
from .forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():  # checks for correct input
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']
            # Here you could save to database or send an email
            return render(request, 'contact_success.html', {'name': name})
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})
```

Step 3: Create templates

contact.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Contact Us</title>
</head>
<body>
    <h1>Contact Us</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Send</button>
    </form>
</body>
</html>
```

- `{{ form.as_p }}` renders all form fields wrapped in `<p>` tags.
- `{% csrf_token %}` protects against CSRF attacks (important!).

contact_success.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Success</title>
</head>
<body>
    <h1>Thank you, {{ name }}! Your message has been sent.</h1>
```

```
</body>
</html>
```

Step 4: Add URL

In `mysite/urls.py`:

```
from blog import views

urlpatterns = [
    path('contact/', views.contact),
]
```

- Visit `http://127.0.0.1:8000/contact/` to see the form in action.

---

3. Using ModelForm to Submit Data to Database

If you want to save data directly to the database:

Step 1: Create a model

In `blog/models.py`:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
```

Run migrations:

```
python manage.py makemigrations
python manage.py migrate
```

Step 2: Create a ModelForm

In `blog/forms.py`:

```
from django.forms import ModelForm
from .models import Post

class PostForm(ModelForm):
    class Meta:
```

```
        model = Post
        fields = ['title', 'content']
```

Step 3: Create a view to handle form submission

In `blog/views.py`:

```
def create_post(request):
    if request.method == 'POST':
        form = PostForm(request.POST)
        if form.is_valid():
            form.save()  # saves to database
            return render(request, 'post_success.html', {'title':
form.cleaned_data['title']})
    else:
        form = PostForm()
    return render(request, 'create_post.html', {'form': form})
```

Step 4: Templates

create_post.html

```
<h1>Create a New Post</h1>
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Submit</button>
</form>
```

post_success.html

```
<h1>Post "{{ title }}" created successfully!</h1>
```

Step 5: Add URL

```
path('create-post/', views.create_post),
```

Visit `http://127.0.0.1:8000/create-post/` and try submitting a post.

---

4. Key Tips for Beginners

1. Always use `{% csrf_token %}` in forms.
2. Use `is_valid()` to validate input before saving.
3. Use `cleaned_data` to safely access form data.
4. Start with simple `Form` before moving to `ModelForm`.
5. Test your forms by submitting different kinds of inputs to see validation in action.

Right now, your **contact view** collects the data but doesn't save it anywhere. To store the submitted data in a **database**, you need a **model** that matches the form fields, and then you save the form data into it.

Here's a **step-by-step guide** to modify your code so that it saves to the database.

---

1. Create a Model for Contact Messages

In `blog/models.py` (or whichever app you're using):

```
from django.db import models

class ContactMessage(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()
    message = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)  # automatically
saves submission time

    def __str__(self):
        return f"{self.name} - {self.email}"
```

- `CharField` → stores text
- `EmailField` → validates email
- `TextField` → for longer text messages
- `auto_now_add=True` → automatically sets the date/time when saved

---

2. Apply Migrations

After creating the model, run these commands in Command Prompt:

```
python manage.py makemigrations
python manage.py migrate
```

This will **create the database table** for `ContactMessage`.

---

3. Save Form Data to the Database

Update your `contact` view in `views.py`:

```
from django.shortcuts import render
from .forms import ContactForm
from .models import ContactMessage  # import the model

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # Get cleaned data from the form
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']

            # Save data to database
            contact_message = ContactMessage(
                name=name,
                email=email,
                message=message
            )
            contact_message.save()  # this writes it to the database

            # Pass name to success page
            return render(request, 'contact_success.html', {'name': name})
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})
```

 What changed:

- Imported `ContactMessage` model.
- Created an instance with `ContactMessage(name=name, email=email, message=message)`.
- Called `.save()` to store it in the database.

---

4. Optional: Using ModelForm (Simpler Way)

Instead of manually saving, you can use a **ModelForm** which automatically connects your form to the model.

```
# forms.py
from django.forms import ModelForm
from .models import ContactMessage

class ContactForm(ModelForm):
    class Meta:
        model = ContactMessage
        fields = ['name', 'email', 'message']
```

Then your `views.py` becomes simpler:

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            form.save()  # automatically saves to database
            return render(request, 'contact_success.html', {'name':
form.cleaned_data['name']})
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})
```

- No need to manually create the instance.
- Less code, easier to maintain.

Now, when someone submits the contact form, their **name, email, message, and timestamp** will be stored in the database!

Let's build a **complete beginner-friendly Django project** called **"Contact Manager"** where users can submit messages via a form, the messages are saved in the database, and you can view them in the Django admin. I'll give **step-by-step instructions** suitable for a Windows PC.

Django Contact Manager Project

---

1. Project Setup

1. **Create a virtual environment and activate it**:

```
python -m venv myenv
myenv\Scripts\activate
```

2. **Install Django**:

```
pip install django
```

3. **Create a Django project**:

```
django-admin startproject contact_manager
cd contact_manager
```

4. **Create a new app called `contact`**:

```
python manage.py startapp contact
```

5. **Register the app** in `contact_manager/settings.py`:

```
INSTALLED_APPS = [
    ...
    'contact',
]
```

---

2. Create the Database Model

In `contact/models.py`:

```
from django.db import models

class ContactMessage(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()
    message = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"{self.name} - {self.email}"
```

6. **Apply migrations**:

```
python manage.py makemigrations
python manage.py migrate
```

---

3. Enable Admin Access

1. In `contact/admin.py`:

```
from django.contrib import admin
from .models import ContactMessage

admin.site.register(ContactMessage)
```

2. **Create a superuser**:

```
python manage.py createsuperuser
```

- Follow prompts to set username, email, and password.

3. Start the server:

```
python manage.py runserver
```

- Visit `http://127.0.0.1:8000/admin/`
- Log in with your superuser account. You should see **Contact Messages** ready to view.

---

4. Create a ModelForm

In `contact/forms.py`:

```
from django.forms import ModelForm
from .models import ContactMessage

class ContactForm(ModelForm):
    class Meta:
        model = ContactMessage
        fields = ['name', 'email', 'message']
```

- This form is directly linked to your database model.

---

5. Create Views for the Form

In `contact/views.py`:

```
from django.shortcuts import render
from .forms import ContactForm

def contact_view(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            form.save()  # saves to database automatically
            return render(request, 'contact_success.html', {'name':
form.cleaned_data['name']})
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})
```

---

6. Create Templates

1. Inside `contact/` create a folder `templates/contact/`:

```
contact/
└── templates/
    └── contact/
        ├── contact.html
        └── contact_success.html
```

contact.html:

```
<!DOCTYPE html>
<html>
<head>
    <title>Contact Us</title>
</head>
<body>
    <h1>Contact Us</h1>
    <form method="post">
```

```
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Send</button>
    </form>
</body>
</html>
```

contact_success.html:

```
<!DOCTYPE html>
<html>
<head>
    <title>Success</title>
</head>
<body>
    <h1>Thank you, {{ name }}! Your message has been sent.</h1>
    <a href="/contact/">Send another message</a>
</body>
</html>
```

---

7. Add URL Patterns

1. In `contact/urls.py`:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.contact_view, name='contact'),
]
```

2. In `contact_manager/urls.py`:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('contact/', include('contact.urls')),
]
```

---

8. Run the Project

1. Start the server:

```
python manage.py runserver
```

2. Visit `http://127.0.0.1:8000/contact/`
    - Submit a message via the form.
    - You will see the **success page**.
3. Check the **admin panel** at `http://127.0.0.1:8000/admin/`
    - Your submitted messages are now in the database.

□ What You Learned in This Project

- Setting up a Django project and app on Windows.
- Creating a **database model** (`ContactMessage`).
- Creating a **ModelForm** linked to the model.
- Submitting data via a **form** and saving it in the database.
- Displaying a **success page**.
- Viewing saved data in **Django admin**.

This is a **full beginner-friendly workflow** combining:

- URL routing
- Forms
- Models
- Templates
- Admin panel

Now that you have a working project with models and data, it's time to explore **Django ORM**—the tool Django provides to **query the database using Python code** instead of writing raw SQL. This is beginner-friendly and very powerful.

Let's go **step by step** using the `ContactMessage` model from your project.

---

1. What is Django ORM?

- **ORM** = Object-Relational Mapping
- Lets you interact with database tables as Python objects.
- Each Django **model** corresponds to a **table**, and each **model instance** corresponds to a **row**.

Example:

```
from contact.models import ContactMessage
messages = ContactMessage.objects.all()  # gets all messages
```

No SQL is needed!

---

2. Basic Queries

a) Get all records

```
from contact.models import ContactMessage

all_messages = ContactMessage.objects.all()
print(all_messages)
```

- Returns a **QuerySet**, like a Python list of model instances.
- QuerySets are **lazy**, meaning the database isn't actually queried until you iterate over it.

---

b) Get a single record

```
first_message = ContactMessage.objects.first()  # first record
last_message = ContactMessage.objects.last()    # last record
specific_message = ContactMessage.objects.get(id=1)  # record with ID = 1
```

- **Tip:** `get()` throws an exception if the record doesn't exist. Use `try/except` or `get_object_or_404()` in views.

---

c) Filter records

```
# messages by a specific person
john_messages = ContactMessage.objects.filter(name="John Doe")

# messages containing a keyword
keyword_messages = ContactMessage.objects.filter(message__icontains="hello")
```

- `__icontains` → case-insensitive match
- Other common lookups: `exact`, `startswith`, `endswith`, `gte`, `lte`

---

d) Exclude records

```
# exclude messages from John
not_john = ContactMessage.objects.exclude(name="John Doe")
```

---

e) Order records

```
# newest first
messages_ordered = ContactMessage.objects.all().order_by('-created_at')

# oldest first
messages_ordered = ContactMessage.objects.all().order_by('created_at')
```

---

f) Count records

```
total_messages = ContactMessage.objects.count()
john_count = ContactMessage.objects.filter(name="John Doe").count()
```

---

g) Get specific fields

```
names = ContactMessage.objects.values_list('name', flat=True)
for name in names:
    print(name)
```

- `values_list()` → returns a list of values instead of full objects.

---

3. Updating Records

```
# get a message
message = ContactMessage.objects.get(id=1)
message.message = "Updated message content"
message.save()  # save changes to the database
```

---

4. Deleting Records

```
message = ContactMessage.objects.get(id=1)
message.delete()  # removes it from the database
```

- Always be careful with `.delete()`, especially in production.

---

5. Using ORM in Views

You can use ORM queries in views to show dynamic data on web pages.

Example: Show all messages in a template.

```
# views.py
from django.shortcuts import render
from .models import ContactMessage

def view_messages(request):
    messages = ContactMessage.objects.all().order_by('-created_at')
    return render(request, 'view_messages.html', {'messages': messages})
```

Template: templates/`view_messages.html`

```
<h1>All Messages</h1>
<ul>
    {% for msg in messages %}
        <li><strong>{{ msg.name }}</strong>: {{ msg.message }} ({{
msg.created_at }})</li>
    {% empty %}
        <li>No messages found.</li>
    {% endfor %}
</ul>
```

URL pattern:

```
path('messages/', views.view_messages, name='view_messages')
```

- Visit `http://127.0.0.1:8000/messages/` to see all submitted messages.

---

6. Tips for Beginners Using ORM

1. Always use **filter()** instead of **get()** when unsure about multiple results.
2. Use **order_by()** to control the display of results.
3. Use **values()** or **values_list()** for efficiency when you only need specific fields.
4. Use **Django shell** for practice:

```
python manage.py shell
```

Then you can try queries interactively:

```
from contact.models import ContactMessage
ContactMessage.objects.filter(name="John")
```

5. Remember: ORM is **Pythonic**, readable, and safer than raw SQL.

Let's build a **mini-project called "Message Viewer"** where beginners can **view all messages, filter by name, and search keywords** using Django ORM. This will combine everything you've learned: **models, views, templates, URL parameters, and ORM queries**.

Django Mini-Project: Message Viewer

---

1. Project Setup

Assuming you already have the **Contact Manager project** with the `ContactMessage` model:

```
python -m venv myenv
myenv\Scripts\activate
pip install django
```

If not, create a new app called `message_viewer` inside your project:

```
python manage.py startapp message_viewer
```

Add it to `INSTALLED_APPS` in `settings.py`:

```
INSTALLED_APPS = [
    ...
    'message_viewer',
]
```

---

2. URL Patterns

In `message_viewer/urls.py`:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.all_messages, name='all_messages'),
    path('search/', views.search_messages, name='search_messages'),
    path('filter/<str:name>/', views.filter_by_name, name='filter_by_name'),
]
```

Include it in `project/urls.py`:

```
path('messages/', include('message_viewer.urls')),
```

---

3. Views with ORM Queries

In `message_viewer/views.py`:

```
from django.shortcuts import render
from contact.models import ContactMessage  # reuse existing model

# 1 View all messages
def all_messages(request):
    messages = ContactMessage.objects.all().order_by('-created_at')
    return render(request, 'message_viewer/all_messages.html', {'messages':
messages})

# 2 Filter messages by name using URL parameter
def filter_by_name(request, name):
    messages = ContactMessage.objects.filter(name=name).order_by('-
created_at')
    return render(request, 'message_viewer/all_messages.html', {'messages':
messages, 'filter_name': name})

# 3 Search messages by keyword
def search_messages(request):
    query = request.GET.get('q', '')  # get 'q' from URL ?q=keyword
    messages =
ContactMessage.objects.filter(message__icontains=query).order_by('-
created_at') if query else []
    return render(request, 'message_viewer/search_results.html', {'messages':
messages, 'query': query})
```

---

4. Templates

Create folder: `message_viewer/templates/message_viewer/`

all_messages.html

```
<h1>All Messages</h1>
{% if filter_name %}
    <h3>Filtered by: {{ filter_name }}</h3>
{% endif %}
<ul>
    {% for msg in messages %}
        <li><strong>{{ msg.name }}</strong>: {{ msg.message }} ({{
msg.created_at }})</li>
    {% empty %}
        <li>No messages found.</li>
    {% endfor %}
</ul>

<h2>Search Messages</h2>
<form method="get" action="{% url 'search_messages' %}">
    <input type="text" name="q" placeholder="Search keyword">
```

```
    <button type="submit">Search</button>
</form>
```

search_results.html

```
<h1>Search Results for "{{ query }}"</h1>
<ul>
    {% for msg in messages %}
        <li><strong>{{ msg.name }}</strong>: {{ msg.message }} ({{
msg.created_at }})</li>
    {% empty %}
        <li>No messages found for "{{ query }}"</li>
    {% endfor %}
</ul>

<a href="{% url 'all_messages' %}">Back to all messages</a>
```

---

5. Examples of Using the Mini-Project

1. **View all messages**
   Visit: `http://127.0.0.1:8000/messages/`
2. **Filter messages by name**
   Visit: `http://127.0.0.1:8000/messages/filter/John/`
   Only messages from John are shown.
3. **Search messages by keyword**
   Visit: `http://127.0.0.1:8000/messages/search/?q=hello`
   Only messages containing “hello” are shown.

---

6. Key Features for Beginners

- **`objects.all()`** → get all records
- **`filter()`** → narrow results with conditions
- **`__icontains`** → case-insensitive search
- **`order_by('-created_at')`** → newest messages first
- **Dynamic URLs** → filter by name
- **GET parameters** → search by keyword

□ What This Mini-Project Teaches

1. Using **Django ORM** to query the database.
2. Creating **dynamic pages** with URL parameters (`filter/<name>/`).
3. Using **GET parameters** to search for text.
4. Rendering **query results in templates**.
5. Combining **forms, views, models, templates, and ORM**.