MODULE 2: DJANGO TEMPLATES AND MODELS

Template-System Basics

Purpose of a Django Template:

- **Separation of Presentation and Data:** A Django template separates the presentation (HTML, text) from the data it displays.
- **Placeholders and Logic:** Templates contain placeholders for data and basic logic (called template tags) to manage the display of the document.

Common Use:

HTML Generation: Typically used to generate HTML documents, but can generate any text-based format.

Simple Example Template:

The example given creates an HTML page to thank a person for placing an order.

Example Template Explanation

Here is the example template with explanations:

```
<html>
<head><title>Ordering notice</title></head>
<body>

<h1>Ordering notice</h1>

<p>Dear {{ person_name }},</p>
<p>Thanks for placing an order from {{ company }}. It's scheduled to
ship on {{ ship_date|date:"F j, Y" }}.</p>
<p>Here are the items you've ordered:</p>
<ul>
{% for item in item_list %}
<li>{{ item }}</li>
{% endfor %}
```

```
</ul>
{% if ordered_warranty %}
<p>Your warranty information will be included in the packaging.</p>
{% else %}
<p>You didn't order a warranty, so you're on your own when
the products inevitably stop working.</p>
{% endif %}
<p>Sincerely,<br />{{ company }}</p>
</body>

</html>
```

Key Components Explained

HTML Structure:

- **<html>, <head>, <body>:** Basic HTML tags to structure the document.
- Dynamic Content with Placeholders:
    - **{{ person_name }}:** Placeholder for the person's name.
    - **{{ company }}:** Placeholder for the company's name.
    - **{{ ship_date|date:"F j, Y" }}:** Placeholder for the shipping date, formatted as "Month day, Year" (e.g., June 14, 2024).
- Looping through Items:
    - {% for item in item_list %}: Loop through each item in item_list.
    - <li>{{ item }}</li>: Display each item in a list item (<li>).
- Conditional Logic:
    - {% if ordered_warranty %}: Check if a warranty was ordered.
    - If the warranty was ordered, display a message about the warranty.
    - {% else %}: If no warranty was ordered, display a different message.

- Django Templates are used to generate dynamic content by separating data and presentation.
- Placeholders and Template Tags help manage and display dynamic data.

Example Template: Demonstrates placeholders for personalizing messages, loops for listing items, and conditions for displaying different messages based on certain criteria (like ordering a warranty).

Using the Django Template System

- Django's template system can be used independently of the rest of Django.
- It allows you to create dynamic text-based content, typically HTML, by separating presentation from data.

Basic Steps

- Create a Template Object: Provide raw template code as a string.
- Render the Template: Use the render () method with a set of variables (context) to get the final rendered string.

```
from django import template

# Step 1: Create a Template object
t = template.Template('My name is {{ name }}.')

# Step 2: Render the template with context
c = template.Context({'name': 'Adrian'})
print(t.render(c)) # Output: My name is Adrian.

c = template.Context({'name': 'Fred'})
print(t.render(c)) # Output: My name is Fred.
```

Detailed Steps

Creating Template Objects:

- Import the Template class from Django. template.
- Instantiate the Template class with raw template code.

- The template system compiles the code for efficient rendering

```
from django.template import Template
t = Template('My name is {{ name }}.')
print(t) # Output: <django.template.Template object at 0xb7d5f24c>
```

Handling Syntax Errors:

If there are syntax errors, a TemplateSyntaxError is raised.

```
from django.template import Template
t = Template('{% notatag %}') # Raises TemplateSyntaxError
```

Rendering a Template:

- Create a Context object with variable values.
- Pass the context to the render() method of the template object.

```
from django.template import Context, Template
t = Template('My name is {{ name }}.')
c = Context({'name': 'Stephane'})
print(t.render(c)) # Output: u'My name is Stephane.'
```

Complex Example:

- Use multiline strings for complex templates.
- Create and render the template with context.

```
from django.template import Template, Context
import datetime

raw_template = """<p>Dear {{ person_name }},</p>
<p>Thanks for placing an order from {{ company }}. It's scheduled to
ship on {{ ship_date|date:"F j, Y" }}.</p>
{% if ordered_warranty %}
<p>Your warranty information will be included in the packaging.</p>
{% else %}
<p>You didn't order a warranty, so you're on your own when
the products inevitably stop working.</p>
{% endif %}
```

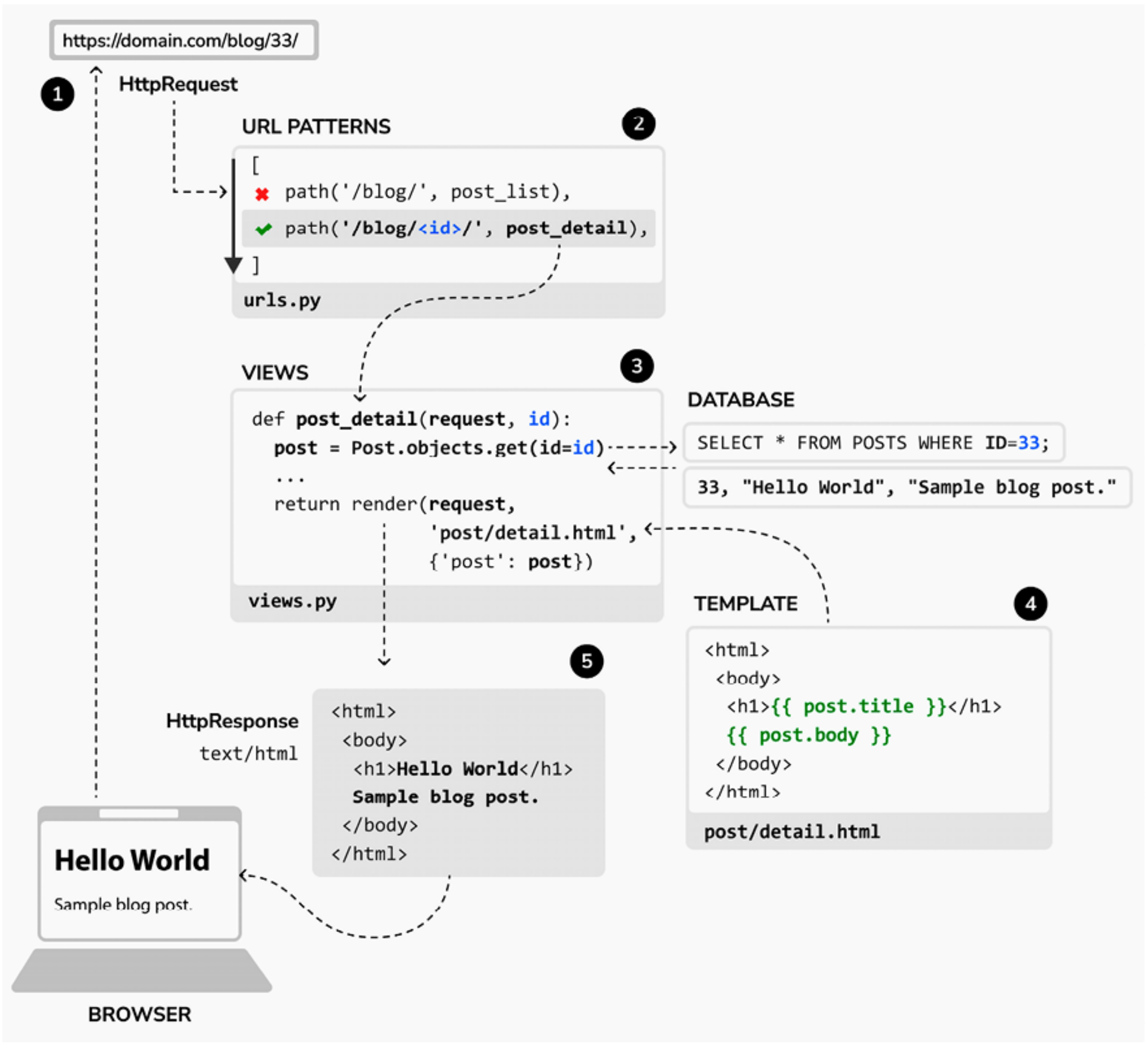
```
<p>Sincerely,<br />{{ company }}</p>"""

t = Template(raw_template)
c = Context({
'person_name': 'John Smith',
'company': 'Outdoor Equipment',
'ship_date': datetime.date(2009, 4, 2),
'ordered_warranty': False
})

print(t.render(c))
# Output:
# <p>Dear John Smith,</p>
# <p>Thanks for placing an order from Outdoor Equipment. It's scheduled to
# ship on April 2, 2009.</p>
# <p>You didn't order a warranty, so you're on your own when
# the products inevitably stop working.</p>
# <p>Sincerely,<br />Outdoor Equipment</p>
```

Django Templates: Used to separate presentation from data.

Placeholders and Tags: Utilize placeholders for dynamic content and tags for logic.

Context: Pass variable values to templates via context.

Render: Generate the final string by rendering the template with context



```
[
  path('/blog/', post_list),
  path('/blog/<id>/', post_detail),
]
```

```
def post_detail(request, id):
  post = Post.objects.get(id=id)
  ...
  return render(request,
                'post/detail.html',
                {'post': post})
```

```
SELECT * FROM POSTS WHERE ID=33;
33, "Hello World", "Sample blog post."
```

```
<html>
 <body>
  <h1>{{ post.title }}</h1>
  {{ post.body }}
 </body>
</html>
```

```
<html>
 <body>
  <h1>Hello World</h1>
  Sample blog post.
 </body>
</html>
```

Basic Template Tags and Filters

1) Tags

A template tag is a special syntax used within templates to implement logic and control the rendering of content. Template tags are enclosed in {% %} and can perform various functions, such as looping over data, conditional statements, or creating variables. For example, the {% with %} tag allows you to define a variable for use within a template, which can help improve readability and efficiency, especially in loops.

The template tags are a way of telling Django that here comes something else than plain HTML. The template tags allows us to to do some programming on the server before sending HTML to the client.

a. if/else Tag:

- Evaluates a variable to determine if it is True.
- Displays content between {% if %} and {% endif %} if the variable is True.
- Supports {% else %} for alternate content.
- Allows and, or, and not for multiple conditions, but does not support combined logical operators.
- Nested {% if %} tags can be used for complex conditions.

Example:

```
{% if today_is_weekend %}
<p>Welcome to the weekend!</p>
{% else %}
<p>Get back to work.</p>
{% endif %}
```

b. for Tag:

- Iterates over each item in a sequence.
- Syntax: {% for X in Y %} where Y is the sequence and X is the variable for each iteration.

- Supports reversed for reverse iteration.
- Can nest {% for %} tags and supports {% empty %} for handling empty lists.
- Provides a **forloop** variable with attributes like counter, counter0, revcounter, revcounter0, first, last, and parentloop.

Example:

```
<ul>
{% for athlete in athlete_list %}
<li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

c. **ifequal/ifnotequal Tag:**

- Compares two values and displays content if they are equal or not equal.
- Supports an optional {% else %} tag.
- Accepts hard-coded strings, integers, and decimal numbers as arguments but not complex data types.

Example:

```
{% ifequal user currentuser %}
<h1>Welcome!</h1>
{% else %}
<h1>No News Here</h1>
{% endifequal %}
```

d. **Comments:**

- Single-line comments use {# #}.
- Multiline comments use {% comment %} {% endcomment %}.

Example:

```
{# This is a comment #}
```

```
{% comment %}
This is a
multiline comment.
{% endcomment %}
```

2) Filters

- Filters modify the value of variables before they are displayed.
- Use a pipe character (|) to apply a filter.
- Filters can be chained and some accept arguments.

Examples of Important Filters:

1) **addslashes:**

- Adds a backslash before backslashes, single quotes, and double quotes

Example

```
{{ text|addslashes }}
```

2) **date:**

- Formats a date or datetime object according to a format string.

Example:

```
{{ pub_date|date:"F j, Y" }}
```

3) **length:**

- Returns the length of a value (number of elements in a list or characters in a string).

Example:

```
{{ items|length }}
```

4) **lower:**

- Converts a string to lowercase.

Example:

```
{{ name|lower }}
```

5) **truncatewords:**

- Truncates a string to a specified number of words.

Example:

```
{{ bio|truncatewords:"30" }}
```

MVT Development Pattern

- The Model-View-Template (MVT) is an architectural pattern that separates an application into three interconnected components: the Model, the View, and the Template.
- This pattern helps in building maintainable, scalable, and secure web applications.

1) **Model:**

- The Model is responsible for managing the data of the application.
- It handles the logic for creating, reading, updating, and deleting data from the database.
- Models are Python classes that inherit from the django.db. models. Model class.

Example:

```
# models.py
from django.db import models
class Book(models.Model):
title = models.CharField(max_length=200)
author = models.CharField(max_length=100)
publication_date = models.DateField()
```

2) **View:**

- The View handles the business logic and controls the flow of the application.

- It receives requests from the user, interacts with the Model to fetch or update data, and renders the appropriate Template.
- Views are Python functions or classes that receive HTTP requests and return HTTP responses.

Example:

```
        # views.py
        from django.shortcuts import render
        from .models import Book
        def book_list(request):
        books = Book.objects.all()
context = {'books': books}
return render(request, 'book_list.html', context)
```

3) Template:

- The Template is responsible for presenting the data to the user in an HTML format.
- It defines the structure and layout of the web page.
- Templates are written using Django's template language, which provides tags and filters to control the rendering of dynamic content.

Example:

```
<!-- book_list.html -->
<!DOCTYPE html>
<html>
<head>
<title>Book List</title>
</head>
<body>
<h1>Book List</h1>
<ul>
{% for book in books %}
<li>{{ book.title }} by {{ book.author }}</li>
{% endfor %}
</ul>
</body>
```

```
</html>
```

Template Loading

1) **Template Loading in Django:**

   - Django provides a powerful API for loading templates from the filesystem.
   - Templates are loaded using the get_template () function from django. template. loader.

2) **Setting Template Directories:**

   - In the settings.py file, you specify template directories using the TEMPLATE_DIRS setting.
   - Templates can be stored anywhere, but the directory must be readable by the web server user.
   - Absolute paths are recommended, but you can construct paths dynamically using Python code.

3) **Loading Templates Dynamically:**

   - Django settings files are Python code, so you can construct TEMPLATE_DIRS dynamically using Python functions.

4) **Using render_to_response ():**

   - render_to_response () is a shortcut function that loads a template, renders it, and returns an HttpResponse object in one line.
   - It's commonly used instead of manually loading templates and creating context.

5) **Using locals() to Simplify Context Creation:**

   - locals() is a built-in Python function that returns a dictionary mapping all local variable names to their values.

- It can be used to simplify passing variables to templates, reducing redundancy.

6) **Organizing Templates with Subdirectories:**

- Templates can be organized into subdirectories within the template directory.
- This is recommended for better organization, especially for larger projects.

7) **Using the {% include %} Template Tag:**

- {% include %} is a built-in template tag used to include the contents of another template.
- It's useful for reducing duplication when the same code is used in multiple templates.

8) **Behavior of {% include %} Tag:**

- Included templates are evaluated with the context of the template that includes them.
- If the included template isn't found, Django raises a TemplateDoesNotExist exception (in DEBUG mode) or fails silently (in production).

Template Inheritance

1) **Purpose of Template Inheritance:**

Template inheritance helps reduce duplication and redundancy in HTML pages by allowing common parts to be defined in a single location and reused across multiple templates.

2) **Server-Side Includes vs. Template Inheritance:**

While server-side includes (e.g., {% include %}) can be used to include common HTML snippets, template inheritance provides a more elegant solution by defining a base template with blocks that child templates can override.

3) **Defining Base Template:**

- The base template contains the overall structure of the page and defines blocks using {% block %} tags.
- Blocks represent areas of the template that can be customized or overridden by child templates.
- Each block can have a default content, which child templates can choose to override.

Example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
<title>The current time</title>
</head>
<body>
<h1>My helpful timestamp site</h1>
<p>It is now {{ current_date }}.</p>
<hr>
<p>Thanks for visiting my site.</p>
</body>
</html>
```

4) **Using {% extends %}:**

Child templates use the {% extends %} tag to indicate that they inherit from a specific base template. The child template overrides specific blocks defined in the base template using {% block %} tags.

Example:

```
{% extends "base.html" %}
{% block title %}The current time{% endblock %}
{% block content %}
<p>It is now {{ current_date }}.</p>
{% endblock %}
```

5) **Benefits of Template Inheritance:**

- Reduces redundancy by allowing common elements to be defined in a single location.
- Facilitates easier maintenance and updates, as changes made to the base template automatically reflect in all child templates.

6) Guidelines for Working with Template Inheritance:

- {% extends %} must be the first template tag in a child template.
- It's better to have more {% block %} tags in base templates to provide flexibility for child templates.
- Duplicating code across templates indicates the need for moving that code into a {% block %} in the base template.
- Use {{ block.super }} to access and extend the content of a block from the parent template.
- Avoid defining multiple {% block %} tags with the same name in the same template.

  Example:

  ```
  {% extends "base.html" %}
  {% block title %}Future time{% endblock %}
  {% block content %}
  <p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>
  {% endblock %}
  ```

7) Dynamic Template Inheritance:

- The argument to {% extends %} can be a variable, allowing for dynamic selection of the parent template at runtime.

- Overall, template inheritance in Django provides a powerful mechanism for creating maintainable and reusable HTML templates across a web application.
- By defining a clear hierarchy of templates, developers can streamline the development process and ensure consistency in the site's appearance and behavior.

MVT Development Pattern

Overall Design Philosophy:

Django encourages loose coupling and strict separation between components to facilitate easier maintenance and updates.

Model-View-Controller (MVC) Pattern:

Django follows the MVC pattern, where "Model" represents the data access layer, "View" handles presentation logic, and "Controller" decides which view to use based on user input.

Breakdown in Django:

- Model (M): Handled by Django's database layer, including data access, validation, behaviors, and relationships.
- View (V): Handled by views and templates, responsible for selecting and displaying data on the web page or document.
- Controller (C): Managed by the framework itself, following URLconf and calling appropriate Python functions based on user input.

MTV Framework:

Django is sometimes referred to as an MTV framework, where "M" stands for "Model," "T" for "Template," and "V" for "View."

- Model (M): Represents the data access layer, covering everything about data and its relationships.
- Template (T): Represents the presentation layer, handling how data should be displayed on a web page.
- View (V): Represents the business logic layer, acting as a bridge between models and templates.

Comparison with Other MVC Frameworks:

- In Django, views describe the data presented to the user, including which data is displayed, not just how it looks.

- Contrary to frameworks like Ruby on Rails, where controllers decide which data to present, Django views are responsible for accessing models and deferring to appropriate templates.

Underlying Concepts:

Both interpretations of MVC (Django's and others like Ruby on Rails) have their validity, and the key is to understand the underlying concepts rather than adhering strictly to one interpretation.

Configuring Databases

Initial Configuration:

- Django requires configuration to connect to the database server.
- Configuration settings are stored in the settings.py file.

Database Settings:

DATABASE_ENGINE: Specifies the database engine to use. Must be set to one of the available options (e.g., PostgreSQL, MySQL, SQLite).

DATABASE_NAME: Specifies the name of the database. For SQLite, specify the full filesystem path.

DATABASE_USER: Specifies the username to use for database access.

DATABASE_PASSWORD: Specifies the password for database access.

DATABASE_HOST: Specifies the host for the database server. If using SQLite or the database is on the same computer, leave this blank.

DATABASE_PORT: Specifies the port for the database server.

Testing Configuration:

- After configuring the database settings, it's recommended to test the configuration.
- Use the Django shell (python manage.py shell) to test the connection.
- Import connection from django.db and create a cursor.

- If no errors occur, the database configuration is correct.

Common Errors and Solutions:

- Errors may occur if settings are incorrect or if required database adapters are missing.
- Solutions include setting correct values for settings, installing required adapters, and ensuring database existence and user permissions.

Defining and Implementing Models in Django

Introduction to Django Models

A model in Django is a representation of a database table, defined as a Python class. It provides an abstraction layer that allows developers to interact with the database using Python code instead of writing raw SQL queries. Django models help define the structure, behavior, and relationships of data stored in a database.

Why Use Django Models?

- They allow developers to define database schema using Python classes.
- They integrate with Django's Object-Relational Mapping (ORM) to provide a high-level database interface.
- They simplify data handling with built-in methods for querying, inserting, updating, and deleting records.
- They provide automatic migration support, making database schema modifications seamless.

Reasons to Use Python Models in Django

1. **ORM (Object-Relational Mapping)**

Django models use an ORM that allows interaction with the database using Python instead of SQL. This means developers don’t need to write database-specific SQL queries. The ORM automatically translates model definitions into SQL queries behind the scenes.

Example of how ORM simplifies database interaction:

```
# Without ORM (SQL Query)
SELECT * FROM Book WHERE author = 'J.K. Rowling';

# With ORM (Django)
Book.objects.filter(author='J.K. Rowling')
```

This abstraction makes database interactions safer, more readable, and database-independent.

2. **Maintainability**
    - Since Django models use Python, they integrate well with version control systems.
    - Changes in the database structure can be handled using migrations without manually modifying tables.
    - The ORM automatically ensures data consistency and relationships.
3. **Security**

Using Django models helps prevent SQL injection attacks by automatically sanitizing database queries.
Example:

```
# Raw SQL (Vulnerable to SQL injection)
cursor.execute("SELECT * FROM users WHERE username = '%s'" % user_input)

# Django ORM (Safe)
User.objects.get(username=user_input)
```

Django ORM automatically escapes dangerous characters, reducing security risks.

4. **Portability**

Django models work across multiple database backends (SQLite, PostgreSQL, MySQL, Oracle). You can switch databases without changing application code.

Key Features of Django Models

1. **ORM Integration**

Django models eliminate the need for raw SQL queries. You can create, retrieve, update, and delete records using simple Python methods.

2. **Auto-Generated Admin Interface**

Once a model is defined, Django’s admin interface provides a user-friendly way to manage database records without writing any extra code.

3. **Built-in Data Validation**

Django model fields come with built-in validation rules such as:

- `max_length=100` → Restricts character length.
- `unique=True` → Ensures unique values in a column.
- `blank=False` → Prevents empty values.
- `default="Default Value"` → Sets a default value if no input is provided.

Example:

```
class UserProfile(models.Model):
    username = models.CharField(max_length=50, unique=True)
    email = models.EmailField(unique=True)
    age = models.IntegerField(default=18)
```

4. Relationships between Models

Django provides relationship fields to model database associations:

- One-to-One (`OneToOneField`) → A user has one profile.
- One-to-Many (`ForeignKey`) → An author can write multiple books.
- Many-to-Many (`ManyToManyField`) → A book can have multiple authors, and an author can write multiple books.

Example:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
```

This ensures data consistency and helps maintain logical relationships between different tables.

5. Migrations for Database Schema Evolution

Django provides a migration framework to apply changes to the database schema effortlessly. Instead of writing manual SQL commands, developers can use:

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

Migrations track changes automatically, making it easy to add or modify tables without losing existing data.

Defining a Model in Django

A model in Django is defined as a Python class that inherits from `models.Model`. Each class attribute represents a database field, with its type and constraints specified.

Example:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    publication_date = models.DateField()
    price = models.DecimalField(max_digits=6, decimal_places=2)

    def __str__(self):
        return self.title
```

Explanation:

1. `from django.db import models` → Imports Django's database module.
2. `class Book(models.Model)` → Defines a `Book` model that inherits from `models.Model`.
3. `title = models.CharField(max_length=200)` → A string field for storing book titles (max 200 characters).
4. `author = models.CharField(max_length=100)` → A string field for storing author names.
5. `publication_date = models.DateField()` → A date field for book publication dates.
6. `price = models.DecimalField(max_digits=6, decimal_places=2)` → A decimal field for storing book prices, ensuring precision.

7. `def __str__(self):` → Defines how model instances are represented as strings. This helps when displaying objects in the Django admin panel.

Registering Models in Django Admin

To make models accessible in the Django Admin panel, register them in `admin.py`:

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

admin.site.register(Book)
```

After running the development server, you can manage book records through the Django admin interface.

Creating and Applying Migrations

What are Migrations?

Migrations in Django are used to apply changes to the database schema when models are created, updated, or deleted. They ensure the database structure stays in sync with the model definitions.

Steps to Create and Apply Migrations

Step 1: Create a Model

Define a Django model in `models.py`.

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    email = models.EmailField(unique=True)
```

Step 2: Create Migrations

Run the following command to generate migration files:

```
python manage.py makemigrations
```

This creates a migration file inside the migrations/ folder.

Step 3: Apply Migrations

To apply the migration and update the database schema, run:

```
python manage.py migrate
```

Step 4: Check Migration Status

To see the status of migrations, use:

```
python manage.py showmigrations
```

Step 5: Rollback a Migration (Optional)

If needed, you can undo the last migration using:

```
python manage.py migrate app_name 0001
```

Here, `0001` is the migration number you want to roll back to.

Querying Data with Models (Django ORM)

Django provides an Object-Relational Mapper (ORM) that allows developers to interact with the database using Python code instead of SQL queries.

1. Creating and Saving Objects

To add a new record to the database:

```
student = Student(name="John Doe", age=22, email="john@example.com")
student.save()  # Saves the record to the database
```

2. Retrieving Data

a) Get All Records

```
students = Student.objects.all()  # Retrieves all records
```

b) Filter Data

```
students = Student.objects.filter(age=22)  # Retrieves all students aged 22
```

c) Get a Single Record

```
student = Student.objects.get(name="John Doe")
```

If the record doesn't exist, `.get()` will raise an exception, while `.filter()` will return an empty queryset.

d) Get First or Last Record

```
first_student = Student.objects.first()
last_student = Student.objects.last()
```

e) Sorting Data

```
students = Student.objects.order_by('name')  # Ascending order by name
students = Student.objects.order_by('-age')  # Descending order by age
```

f) Limiting Query Results

```
students = Student.objects.all()[:5]  # Get first 5 students
```

g) Excluding Records

```
students = Student.objects.exclude(age=22)  # Get all students except those
aged 22
```

h) Counting Records

```
count = Student.objects.count()
```

i) Updating Records

```
student = Student.objects.get(name="John Doe")
student.age = 23
student.save()
```

j) Deleting Records

```
student = Student.objects.get(name="John Doe")
student.delete()
```

Relationships in Models

Django supports relationships between models using different field types:

1. **One-to-One Relationship (`OneToOneField`)**

Used when one record in a model corresponds to exactly one record in another model. Example:

```
class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField()
```

2. **One-to-Many Relationship (`ForeignKey`)**

Used when one record in a model is related to multiple records in another model. Example:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
```

Here, an author can write multiple books, but each book has only one author.

3. **Many-to-Many Relationship (`ManyToManyField`)**

Used when multiple records in one model are related to multiple records in another model. Example:

```
class Student(models.Model):
    name = models.CharField(max_length=100)

class Course(models.Model):
    title = models.CharField(max_length=100)
    students = models.ManyToManyField(Student)
```

Here, a student can enroll in multiple courses, and a course can have multiple students.

Advanced Model Features

1. Meta Options in Models

The `Meta` class inside a model is used to define additional options. Example:

```
class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

    class Meta:
        ordering = ['name']  # Default ordering by name
        verbose_name_plural = "Students"  # Custom name in Django admin
```

2. Custom Model Managers

A model manager allows customization of database queries. Example:

```
class StudentManager(models.Manager):
    def adults(self):
        return self.filter(age__gte=18)

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

    objects = StudentManager()  # Custom manager
```

Now, we can query:

```
adult_students = Student.objects.adults()
```

3. Model Methods

Define custom methods inside models.

Example:

```
class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

    def is_adult(self):
        return self.age >= 18
```

Usage:

```
student = Student.objects.get(name="John")
print(student.is_adult())  # True if age >= 18
```

4. **Signals (Pre-Save and Post-Save Actions)**

Django signals allow executing code when an event occurs (e.g., before saving an object). Example:

```
from django.db.models.signals import pre_save
from django.dispatch import receiver

@receiver(pre_save, sender=Student)
def before_saving_student(sender, instance, **kwargs):
    print(f"About to save: {instance.name}")
```

5. **Abstract Models**

Abstract models allow creating base models that other models can inherit from. Example:

```
class BaseModel(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True  # This model will not be created in the database

class Student(BaseModel):
    name = models.CharField(max_length=100)
```

Basic Data Access in Django ORM

Django provides a built-in Object-Relational Mapper (ORM) to interact with the database using Python instead of raw SQL queries.

1. Accessing the Django ORM

To use Django's ORM, first, open the Django shell:

```
python manage.py shell
```

Then, import the model:

```
from myapp.models import Student
```

2. Retrieving All Objects

```
students = Student.objects.all()
for student in students:
    print(student.name, student.age)
```

3. Filtering Data

```
students = Student.objects.filter(age=20)  # Get all students with age 20
students = Student.objects.filter(name__icontains="John")  # Name contains
'John'
```

4. Retrieving a Single Object

```
student = Student.objects.get(id=1)  # Get student with ID 1
```

Note: `.get()` raises an error if no match is found. Use `.filter()` to avoid exceptions.

Adding Model String Representations

Django models should have a human-readable representation. This is done by defining the `__str__` method.

1. Example Without `__str__` Method

```
class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
```

Without `__str__()`, printing a student object will output:

```
<Student: Student object (1)>
```

2. **Example With `__str__` Method**

```
class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

    def __str__(self):
        return f"{self.name} ({self.age} years old)"
```

Now, printing the object gives:

```
<Student: John Doe (20 years old)>
```

Inserting and Updating Data

1. **Inserting Data**
   a. **Method 1: Creating and Saving Objects**

```
student = Student(name="Alice", age=22)
student.save()  # Saves to the database
```

   b. **Method 2: Using `create()`**

```
Student.objects.create(name="Bob", age=25)
```

2. **Updating Data**
   a. **Method 1: Modify and Save**

```
student = Student.objects.get(name="Alice")
student.age = 23
student.save()  # Updates the record
```

b. **Method 2: Using `update()`**

```
Student.objects.filter(name="Bob").update(age=26)
```

Note: `update()` works on querysets and doesn't require `.save()`.

Selecting and Deleting Objects

1. **Selecting Data (Queries)**

- **Get First Record**

```
student = Student.objects.first()
```

- **Get Last Record**

```
student = Student.objects.last()
```

- **Sorting Data**

```
students = Student.objects.order_by('name')  # Ascending
students = Student.objects.order_by('-age')  # Descending
```

- **Limiting Results**

```
students = Student.objects.all()[:5]  # First 5 students
```

2. **Deleting Objects**

- **Method 1: Deleting a Single Object**

```
student = Student.objects.get(name="Alice")
student.delete()
```

- **Method 2: Bulk Delete**

```
Student.objects.filter(age__lt=18).delete()  # Delete all underage students
```

Schema Evolution (Changing Database Structure Over Time)

Schema evolution refers to modifying a database schema while preserving existing data. Django migrations handle this process.

1. Steps for Schema Changes

Step 1: Modify `models.py`

For example, adding a new field:

```
class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    email = models.EmailField(unique=True, null=True)  # New field
```

Step 2: Create Migrations

```
python manage.py makemigrations
```

Step 3: Apply Migrations

```
python manage.py migrate
```

Step 4: Check Migration Status

```
python manage.py showmigrations
```

2. Handling Schema Changes

a) Adding a New Field

- Add a new field in `models.py`
- Run `makemigrations` and `migrate`
- If the field is `NOT NULL`, provide a default value.

b) Renaming a Field

```
class Student(models.Model):
    full_name = models.CharField(max_length=100)  # Renamed from `name`
```

Run:

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

c) Removing a Field

Simply remove the field from `models.py` and apply migrations.

d) Changing Field Type

Modify the field type in `models.py`, then run `makemigrations` and `migrate`.
Example:

```
age = models.PositiveIntegerField()  # Changed from IntegerField
```

e) Rolling Back a Migration

If an issue arises after migration:

```
python manage.py migrate myapp 0001  # Reverts to migration 0001
```

Combined Summary

- Django models simplify database interactions using Python classes instead of SQL, ensuring security, maintainability, and portability across databases.
- Django ORM allows CRUD operations (Create, Read, Update, Delete) without writing SQL queries, making data access and manipulation efficient.
- Models define database schema, relationships (One-to-One, One-to-Many, Many-to-Many), and constraints, providing structured data handling.
- Migrations manage database schema changes efficiently, enabling schema evolution without losing data.
- Querying data is simplified with ORM methods like `.get()`, `.filter()`, `.save()`, `.create()`, `.update()`, and `.delete()`.
- String representation of models using `__str__` improves readability in Django admin and shell.

- Advanced features like model methods, signals, and custom managers enhance flexibility and extend model functionality.

Topic	Key Points
Basic Data Access	Django ORM lets you retrieve, filter, and manipulate data easily.
Adding Model String Representation	Use `__str__` for human-readable object representations.
Inserting/Updating Data	Use `.save()`, `.create()`, and `.update()` for CRUD operations.
Selecting/Deleting Objects	`.get()`, `.filter()`, `.delete()` provide efficient queries.
Schema Evolution	Use Django migrations (`makemigrations`, `migrate`) to update the database structure.