What is model in Django? What is the purpose of 'models.Model' in defining a Django model?

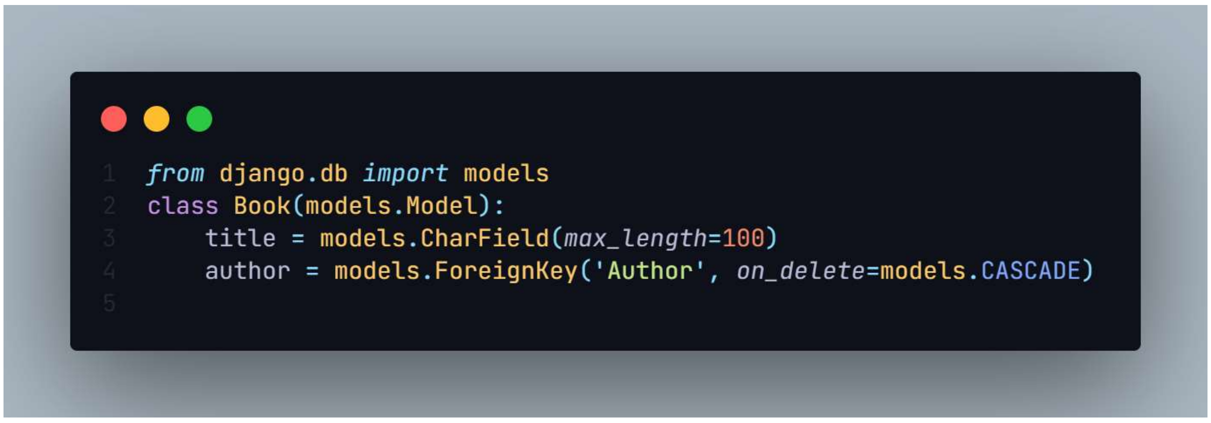
In Django, a **model** is a Python class that defines the structure and behavior of data stored in a database. It serves as a blueprint for creating database tables, where each attribute of the model represents a specific field (e.g., CharField, IntegerField) in the corresponding table. Models abstract database operations, allowing developers to interact with data using Python code instead of SQL.

Purpose of models.Model:
When defining a Django model, inheriting from models.Model is essential. This base class equips the model with Django's **Object-Relational Mapping (ORM)** capabilities. The ORM automates database interactions, enabling:

- **Database schema generation** (via migrations).
- **CRUD operations** (Create, Retrieve, Update, Delete) using Python methods.
- Database-agnostic code (works with SQLite, PostgreSQL, MySQL, etc.).

By subclassing models.Model, the class gains built-in methods and metadata to manage database relationships, validations, and queries efficiently.

For example:



```python
from django.db import models
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey('Author', on_delete=models.CASCADE)
```

Here, Book becomes a database table with fields mapped via the ORM, thanks to models.Model inheritance. This abstraction simplifies database management and promotes clean, maintainable code.

Give some common field types available in Django models?

- **CharField**:
  - Stores short text (e.g., names, titles).
  - Requires max_length (e.g., name = models.CharField(max_length=50)).
- **TextField**:
  - Stores large text (e.g., descriptions, paragraphs).
  - No max_length required (e.g., content = models.TextField()).
- **IntegerField**:
  - Stores integers (e.g., age, quantity).
  - Example: quantity = models.IntegerField().

- FileField:
  - Handles file uploads (e.g., document = models.FileField(upload_to='documents/')).

- ForeignKey:
  - Defines a many-to-one relationship (e.g., linking Book to Author).
  - Requires on_delete (e.g., author = models.ForeignKey(Author, on_delete=models.CASCADE)).

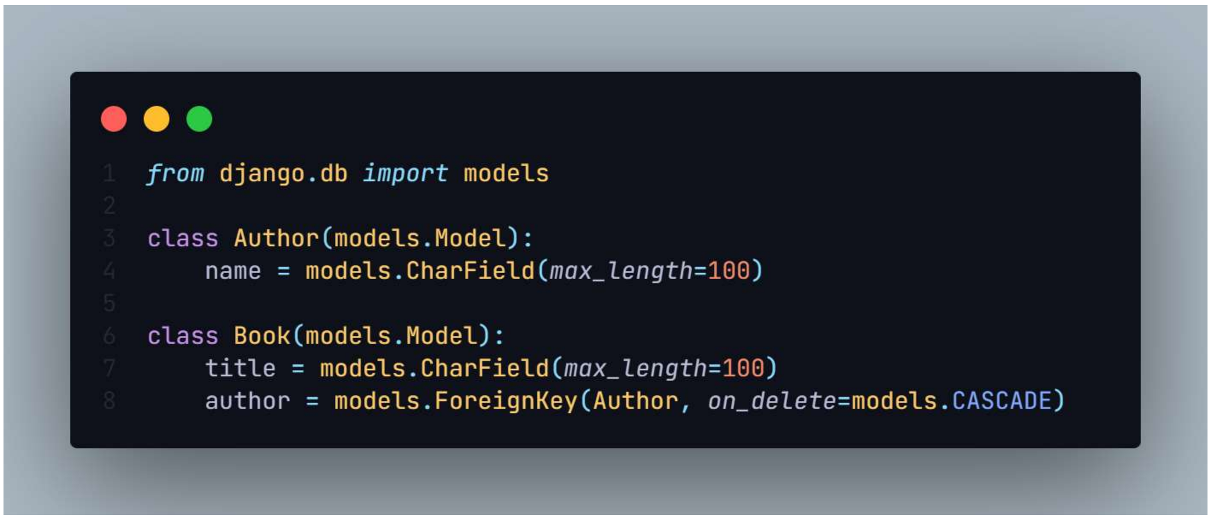
- OneToOneField:
  - Creates a one-to-one relationship (e.g., linking User to Profile).

- ManyToManyField:
  - Defines a many-to-many relationship (e.g., tags = models.ManyToManyField(Tag)).

Explain the purpose and use of the ForeignKey field in Django models.

The **ForeignKey** field in Django is used to establish a **many-to-one relationship** between two models, enabling one model to reference a single instance of another model. It is essential for creating relational database structures, where one object (the "child") is linked to another object (the "parent").

- **Define Relationships:** Connect two models (e.g., a Book belongs to one Author, but an Author can have many Books).
- **Database Integrity:** Enforce referential integrity at the database level.
- **Query Efficiency:** Enable easy querying across related models using Django's ORM.

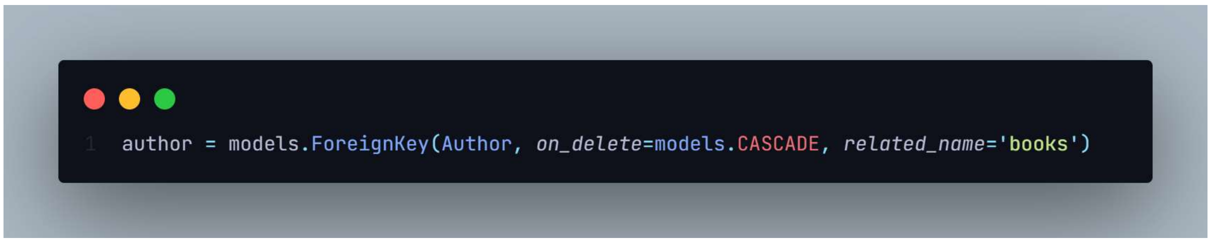


```python
from django.db import models

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

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
```

Here, each Book is associated with one Author, while an Author can have multiple Book entries.

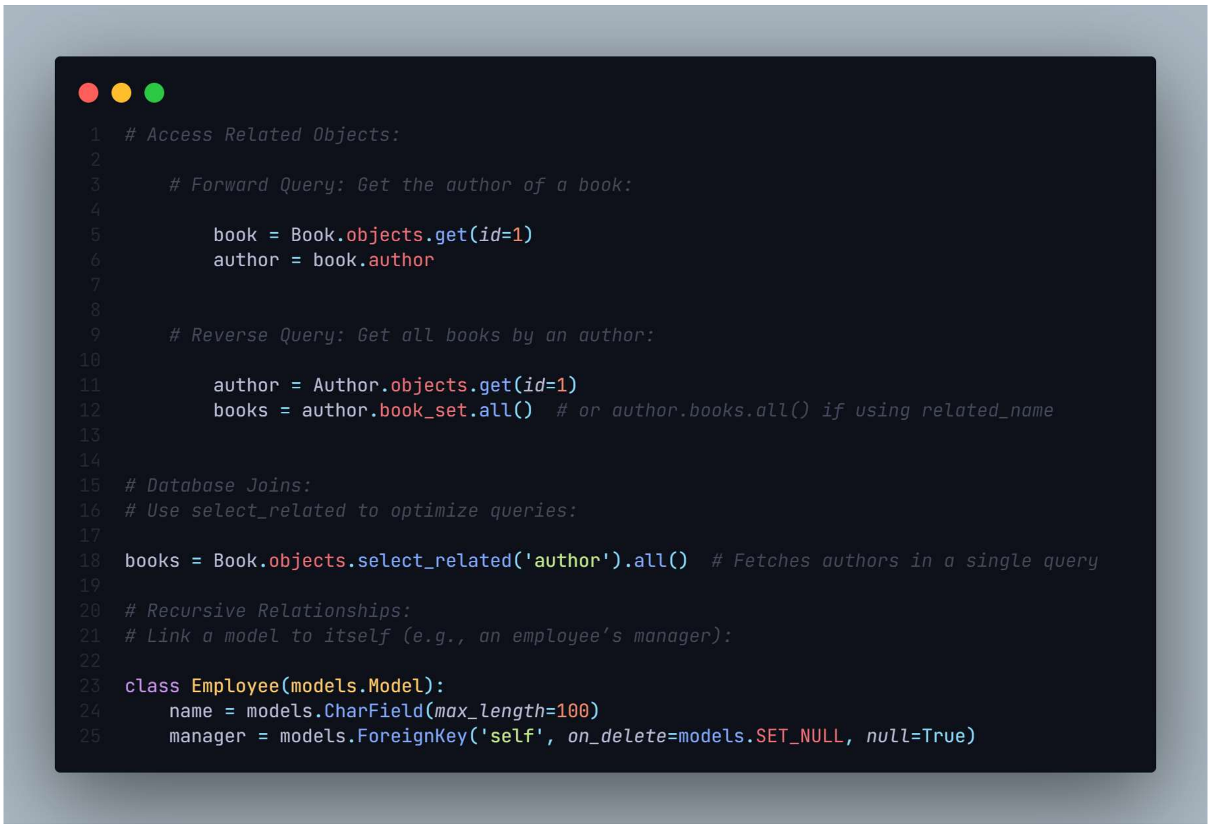
Key Parameters:

- **on_delete** (required): Specifies behavior when the referenced object is deleted. Common options:
    - CASCADE: Delete child objects when the parent is deleted (e.g., delete all books if the author is deleted).
    - PROTECT: Prevent deletion of the parent if child objects exist.
    - SET_NULL: Set the foreign key to NULL if the parent is deleted (requires null=True).
    - SET_DEFAULT: Set the foreign key to a default value.
    - DO_NOTHING: Take no action (use cautiously, as it may break referential integrity).
- **related_name**: Custom name for reverse relationships (e.g., author.books.all() instead of author.book_set.all()):



```
author = models.ForeignKey(Author, on_delete=models.CASCADE, related_name='books')
```

- **related_query_name**: Custom name for reverse filter queries.



```python
# Access Related Objects:

    # Forward Query: Get the author of a book:

        book = Book.objects.get(id=1)
        author = book.author


    # Reverse Query: Get all books by an author:

        author = Author.objects.get(id=1)
        books = author.book_set.all()  # or author.books.all() if using related_name


# Database Joins:
# Use select_related to optimize queries:

books = Book.objects.select_related('author').all()  # Fetches authors in a single query

# Recursive Relationships:
# Link a model to itself (e.g., an employee’s manager):

class Employee(models.Model):
    name = models.CharField(max_length=100)
    manager = models.ForeignKey('self', on_delete=models.SET_NULL, null=True)
```

- A ForeignKey field creates a **database index** by default (improves query speed).
- Use db_index=False to disable indexing if not needed.
- Avoid circular dependencies by referencing models as strings (e.g., ForeignKey('app.Model')).

By using ForeignKey, you can model complex relationships while leveraging Django’s ORM for clean, Pythonic database interactions.

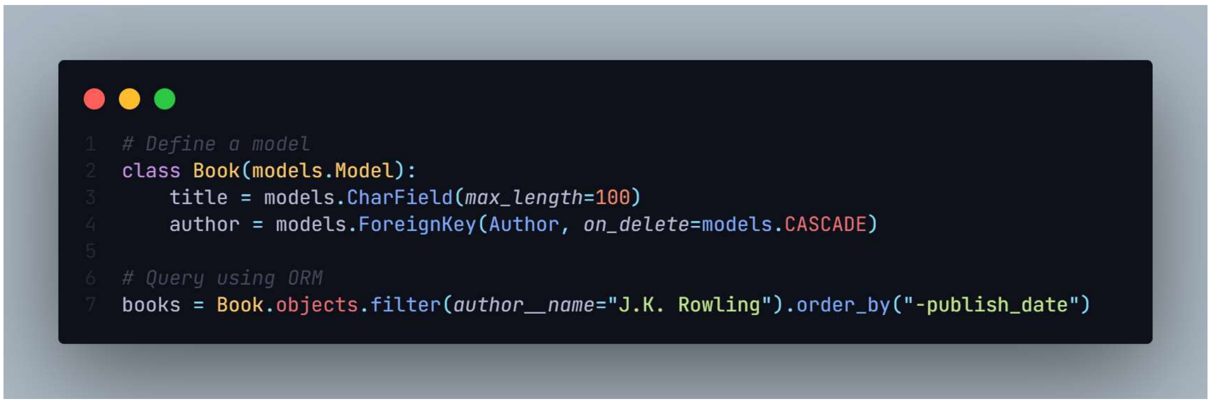
Explain ORM Feature in Django.

Django ORM (Object-Relational Mapper) is a core feature that bridges Python code and relational databases, enabling developers to interact with databases using Python instead of SQL.

Key Features & Benefits:

1. **Database Abstraction**:
   - Define database tables as **Python classes** (models), with fields as class attributes.
   - No SQL required for CRUD operations (e.g., save(), filter(), delete()).
2. **Cross-Database Compatibility**:
   - Write database-agnostic code (works with SQLite, PostgreSQL, MySQL, etc.).
3. **Query Building**:
   - Use **QuerySets** to chain filters, joins, and aggregations (e.g., Book.objects.filter(author__name="Rowling")).
   - Lazy evaluation optimizes performance (queries execute only when needed).
4. **Relationships**:
   - Define **ForeignKey**, **OneToOneField**, and **ManyToManyField** to model database relationships.
5. **Migrations**:
   - Automatically generate and apply schema changes via makemigrations and migrate.
6. **Security**:
   - Prevents SQL injection by sanitizing inputs.

7. **Performance Tools**:
   - Optimize queries with select_related (JOINs) and prefetch_related (batch fetching).



```python
# Define a model
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)

# Query using ORM
books = Book.objects.filter(author__name="J.K. Rowling").order_by("-publish_date")
```

In short: Django ORM simplifies database interactions, promotes clean code, and ensures security while abstracting SQL complexities.