



► Django Models

COMP 8347

Usama Mir

Usama.Mir@unwindsor.ca

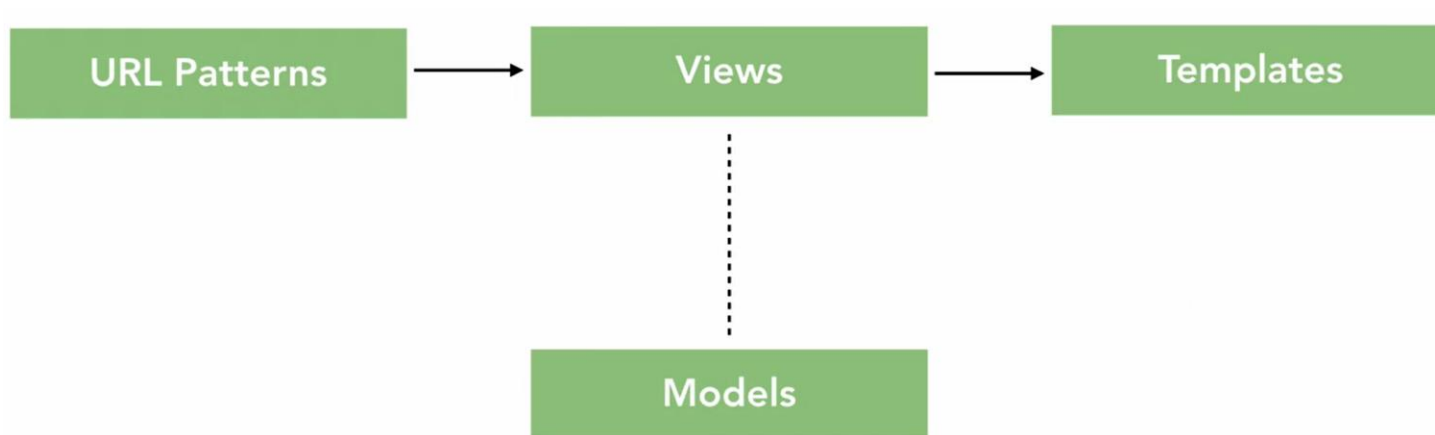
Django Models

- ▶ Topics
 - ▶ Creating simple models
 - ▶ Relationships between models
 - ▶ Advanced usage
 - ▶ Querying Django database



Review MTV Architecture

- ▶ When a Django application receives a request, it uses URL patterns to decide which view to select
- ▶ View manages the logic/control flow portion of a project
- ▶ Models are used to manage the database
- ▶ Templates deal with the view of HTML pages to be returned



More on Models

- ▶ Models:
 - ▶ Create the data layer of a Django app
 - ▶ Define database structure
 - ▶ Allow us to query from the DB
 - ▶ Contain “models.py” file
 - ▶ A model is inherited from `django.db.models.Model`



Why Use ORM?

- ▶ Django provides rich db access layer
 - ▶ Bridges underlying relational db with Python's object oriented nature
 - ▶ **Portability:** support multiple database backends
 - ▶ **Safety:** less prone to security issues (e.g. SQL injection attacks) arising from malformed or poorly protected query strings.
 - ▶ **Encapsulation:** Easy integration with programming language; ability to define arbitrary instance methods

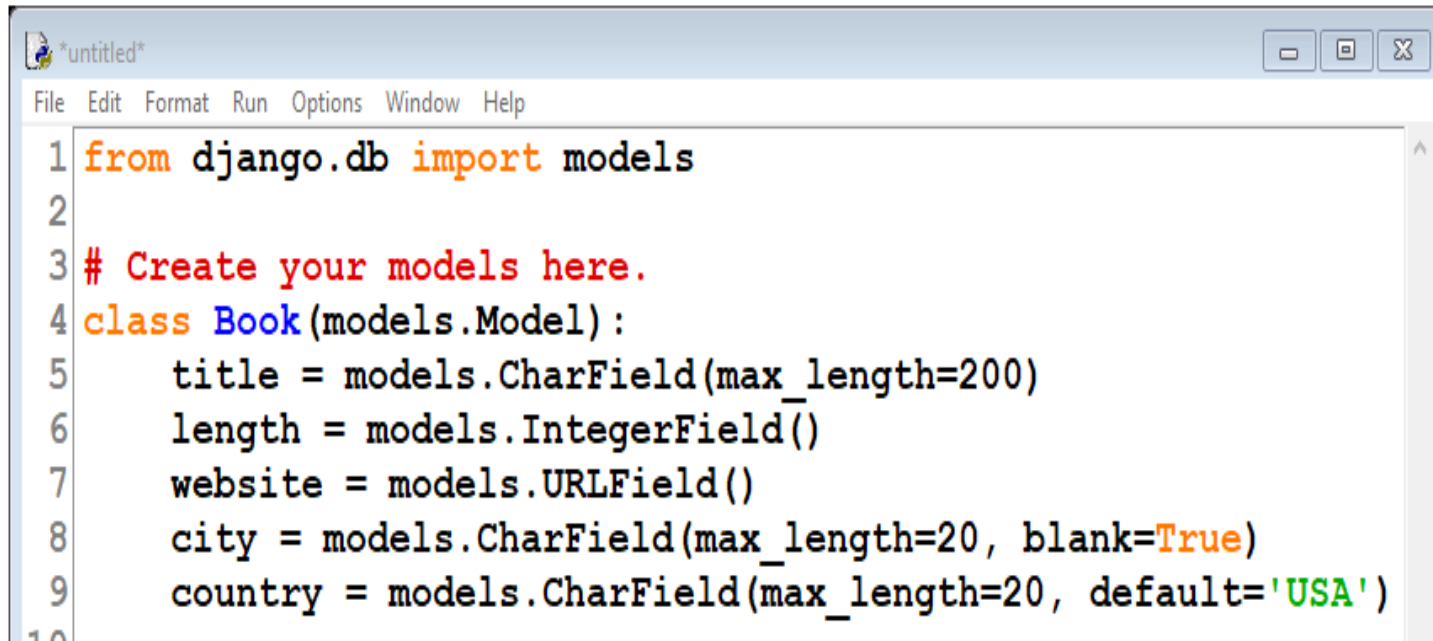


Defining Models

- ▶ **Model** is an object that inherits from **Model** class.
 - ▶ Model → represented by a table in the db
 - ▶ Field → represented by a table column
- ▶ Models are defined and stored in the APP's **models.py** file.
- ▶ **models.py** is automatically created when you start the APP
 - ▶ Contains one line: `from django.db import models`
 - ▶ This allows you to import the base model from Django



Defining Models: An Example



```
*untitled*
File Edit Format Run Options Window Help
1 from django.db import models
2
3 # Create your models here.
4 class Book(models.Model):
5     title = models.CharField(max_length=200)
6     length = models.IntegerField()
7     website = models.URLField()
8     city = models.CharField(max_length=20, blank=True)
9     country = models.CharField(max_length=20, default='USA')
10
```

Field Types - For Textual Data

Field	Example Values
CharField	"Product Name"
TextField	"To elaborate on my point..."
EmailField	<u>george@site.com</u>
URLField	<u>www.example.com</u>



Field Types - For Numeric and Miscellaneous Data

Field	Example Values
IntegerField	-1, 0, 1, 20
DecimalField	0.5, 3.14

Field	Example Values
BooleanField	True, False
DateTimeField	datetime(1960, 1, 1, 8, 0, 0)



Field Types - Null and Blank

null

If **True**, Django will store empty values as **NULL** in the database.
Default is **False**.

blank

If **True**, the field is allowed to be blank. Default is **False**.

```
models.CharField(max_length=10, blank=True)
```



Primary Keys

- ▶ By default Django automatically creates a primary key field.
 - ▶ All models without an explicit primary key field are given an `id` attribute (of type `AutoField`).
 - ▶ `id = models.BigAutoField(primary_key=True)`
 - ▶ **Autofield**: behaves like normal integers; incremented for each new row in table.
 - ▶ To define your own primary key:
 - ▶ specify `primary_key = True` for one of your model fields.
 - ▶ this field becomes the primary key for the table.
 - ▶ it is now your responsibility to ensure this field is unique.



Example

► Person Model:

```
from django.db import models  
class Person(models.Model):  
    first_name = models.CharField(max_length=30)  
    last_name = models.CharField(max_length=30)
```

The above model is created in database as:

```
CREATE TABLE myapp_person (  
    "id" NOT NULL PRIMARY KEY,  
    "first_name" (30) NOT NULL,  
    "last_name" (30) NOT NULL );
```



Example

► Employee Model:

```
class Employee(models.Model):  
    emp_no = models.IntegerField(default=999,  
        primary_key = True)  
    name = models.CharField(max_length=50)  
    age = models.IntegerField()  
    email = models.EmailField(max_length=100)  
    start_date = models.DateField()
```



Field Types - Foreign Key and Many-to-Many

Field	Example Values
ForeignKey	1 (id of record in another table)
ManyToManyField	NA

```
class Company(models.Model):
```

```
    co_name = models.CharField(max_length=50)
```

```
class Car(models.Model):
```

```
    type = models.CharField(max_length=20)
```

```
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
```



Many-to-Many Relationship

- ▶ Uses the **ManytoManyField**.
- ▶ Syntax is similar to **ForeignKey** field.
- ▶ Needs to be defined on **one side** of the relationship only.
 - ▶ Django automatically grants necessary methods and attributes to other side.
 - ▶ Relationship is symmetrical by default → doesn't matter which side it is defined on.



Example

```
class Book(models.Model):  
    title = models.CharField(max_length=100)  
    length = models.IntegerField()  
    pub_date = models.DateField()
```

```
class Author(models.Model):  
    name = models.CharField(max_length=50)  
    books = models.ManyToManyField(Book)
```

NOTE: The Many-to-Many relation is only defined in one model.



Migrations

- ▶ **Migrations:** propagate changes to your models (adding a field, deleting a model, etc.) into your database schema.
 - ▶ Prior to version 1.7, Django only supported adding new models to the database; could not alter or remove existing models.
 - ▶ Used the syncdb command (the predecessor to migrate)



When do we need Migrations?



Adding a Model



Adding a Field



Removing a Field



Changing a Field

Migration Commands

- ▶ ***makemigrations*** : responsible for **creating new migrations** based on the changes made to your models.
- ▶ ***sqlmigrate***: displays the SQL statements for a migration.
- ▶ ***migrate***: **run all the migrations** that have not yet run.

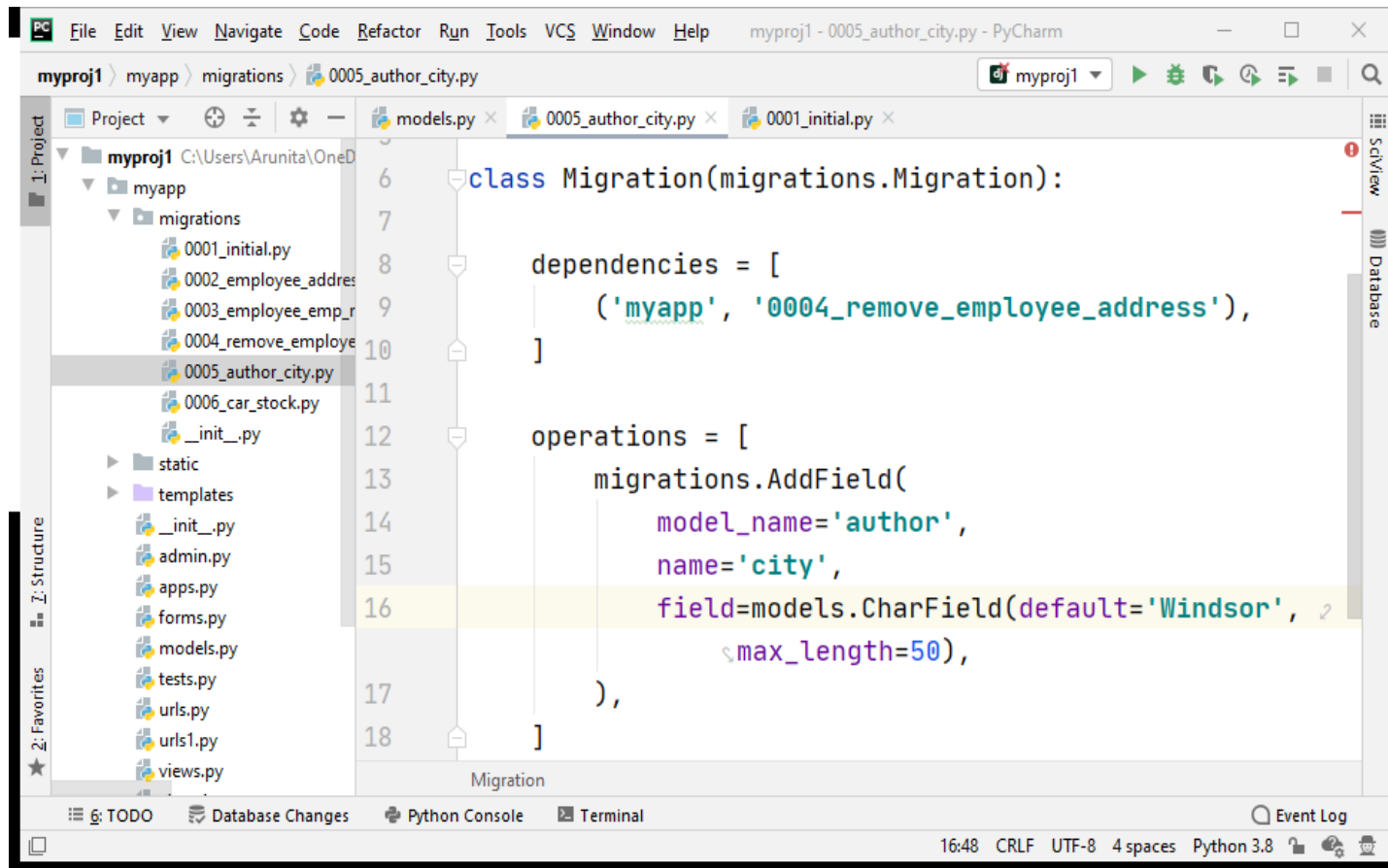


Example of Migrations

```
class Author(models.Model):
```

```
    name = models.CharField(max_length=50)
```

```
    city = models.CharField(max_length=50, default='Windsor')
```



The screenshot shows the PyCharm IDE interface. The main editor window displays the file `0005_author_city.py` within the `migrations` directory of a project named `myproj1`. The code defines a `Migration` class that inherits from `migrations.Migration`. It specifies dependencies on `'myapp', '0004_remove_employee_address'` and a list of operations. The operation `migrations.AddField` is used to add a `city` field to the `author` model, with a `CharField` type, a `default` value of `'Windsor'`, and a `max_length` of `50`. The left sidebar shows the project structure, and the bottom status bar indicates the file encoding is UTF-8 and the Python version is 3.8.

```
6 class Migration(migrations.Migration):
7
8     dependencies = [
9         ('myapp', '0004_remove_employee_address'),
10    ]
11
12    operations = [
13        migrations.AddField(
14            model_name='author',
15            name='city',
16            field=models.CharField(default='Windsor',
17                                   max_length=50),
18        ),
19    ]
```

Model Inheritance

- ▶ Models can inherit from one another, similar to regular Python classes.

- ▶ Previously defined Employee class

```
class Employee(models.Model):
```

```
    name = models.CharField(max_length=50)
```

```
    age = models.IntegerField()
```

```
    email = models.EmailField(max_length=100)
```

```
    start_date = models.DateField()
```

- ▶ Suppose there are 2 types of employees
 - ▶ **programmers** and **supervisors**



Model Inheritance

- ▶ Option 1: Create 2 different models
 - ▶ duplicate all common fields and violate DRY principle.

- ▶ Option 2: Inherit from **Employee** class

```
class Supervisor(Employee):
```

```
    dept = models.CharField(max_length=50)
```

```
class Programmer(Employee):
```

```
    boss = models.ForeignKey(Supervisor, on_delete=models.CASCADE)
```



Adding Methods to Models

- ▶ Since a model is represented as a class, it can have *attributes* and *methods*.
- ▶ One useful method is the `__str__` method which is a dunder method
 - ▶ It controls how the object will be displayed.

```
class Book(models.Model):  
    title = models.CharField(max_length=100)  
    length = models.IntegerField()  
    pub_date = models.DateField()  
    def __str__(self):  
        return self.title
```

Other dunder methods = del, lt, add, sub, mul, abs, len and so on

<https://holycoders.com/python-dunder-special-methods/>



Meta Inner Class

- ▶ **Meta class:** Used to inform Django of various **metadata** about the model.
 - ▶ E.g. display options, ordering, multi-field uniqueness etc.

```
class Employee(models.Model):  
    name = models.CharField(max_length=50)  
    age = models.IntegerField()  
    email = models.EmailField(max_length=100)  
    start_date = models.DateField()
```

```
class Meta:  
    ordering = ['name']
```

Other examples: verbose_name, unique_together, etc



Query Syntax

- ▶ Querying makes use of two similar classes: **Manager** and **QuerySet**
- ▶ **Manager**: Interface through which database query operations are provided to Django models
 - ▶ At least one Manager exists for every model
 - ▶ By default, Django adds a **Manager** with the name **objects** to every Django model class.



Manager Class

- ▶ Manager class has the following methods:
 - ▶ ***all***: returns a **QuerySet** containing all db records for the specified model
 - ▶ ***filter***: returns a **QuerySet** containing model records matching specific criteria
 - ▶ ***exclude***: inverse of filter; return records that don't match the criteria
 - ▶ ***get***: return a single record (model instance) matching criteria
 - ▶ raises error if no match or multiple matches.



Query Examples

```
class Company(models.Model):  
    co_name = models.CharField(max_length=50)
```

```
class Car(models.Model):  
    type = models.CharField(max_length=20)  
    company = models.ForeignKey(Company,  
                                on_delete=models.CASCADE)
```

- ▶ Get all cars in the db.
`car_list = Car.objects.all()`
- ▶ Get the car of type 'Lexus'.
`car1 = Car.objects.get(type='Lexus')`
- ▶ Get the name of the company that made car1.
`name = car1.company.co_name`
- ▶ Get all the cars made by 'Ford'
`company = Company.objects.get(co_name='Ford')`
`cars = company.Car_set.all()`



QuerySet

► **QuerySet:** Can be thought of as a list of model class instances (records/rows)

► above is a simplification - actually much more powerful

► QuerySet examples:

► List of all books:

```
all_books = Book.objects.all()
```

► List of books with the word “Python” in title:

```
python_books = Book.objects.filter(title__contains=“Python”)
```

► The book with id == 1:

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



QuerySet

- ▶ **QuerySet as container:** QuerySet implements a partial list interface and can be iterated over, indexed, sliced, and measured.

- ▶ Example 1:

```
python_books = Book.objects.filter(title__contains='Python')  
for book in python_books:  
    print(book.title)
```

- ▶ Example 2:

```
all_books = Book.objects.all()  
  
How many books in db?  
num_books = len(all_books)  
  
Get the first book:  
first_book = all_books[0]  
  
Get a list of first five books:  
first_five = all_books[:5]
```



QuerySet

- ▶ QuerySet as **building blocks**: QuerySets can be *composed* into complex or nested queries.

- ▶ Example:

```
python_books = Book.objects.filter(title__contains="Python")  
short_python_books = python_books.filter(length__lt=100)
```

- ▶ Equivalently:

```
short_python_books =  
    Book.objects.filter(title__contains="Python").filter(length__lt=  
    100)
```



References

- ▶ <https://docs.djangoproject.com/en/4.1/intro/tutorial02>
- ▶ <https://docs.djangoproject.com/en/4.1/topics/db/models/>
- ▶ <https://docs.djangoproject.com/en/4.1/topics/db/managers/>
- ▶ https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/
- ▶ <https://levelup.gitconnected.com/python-dunder-methods-ea98ceabad15>
- ▶ <https://holycoders.com/python-dunder-special-methods/>
- ▶ Python Web Development with Django, by J. Forcier et al.
- ▶ Slides from Dr. Arunita and Dr. Saja

