ITE3162 – DJANGO FUNDAMENTALS

PELIN

OUTLINE

In this course, you will learn:

1. How to create routes (or views) with Django.
2. How to serve static content and files using Django.
3. How to connect templates with models to serve data dynamically.
4. How to create Models and how to connect them with Templates and Views.
5. How to work with databases using SQLite & PostgreSQL.
6. How to handle and validate forms in Django.
7. How to create Relative URLs with templates and how to check out Template and Custom Filters.
8. How to create User models and forms and implement login and registration.
9. Finally, the most rewarding outcome of this course is that you will learn:
10. How to build a Django application, hands-on.

INTRO TO DJANGO

WHAT IS DJANGO

free and open-source web application framework written in Python.

Used for rapid web development and clean, pragmatic design.

Focus on writing apps instead of reinventing the wheel.

Created in 2003 at the Lawrence Journal-World newspaper started using Python for their web development.

In 2005 Django became open source

It has a large community that contributes a lot to its development by adding features or developing packages. (djangopackages.org)

Django is considered an **"opinionated framework"**

DJANGO FEATURES

Super fast: Django development is extremely fast.

Fully loaded: Django has dozens of projects that can be integrated to carry out common tasks such as user authentication, authorization, and content administration.

Versatile: Django can be used for almost any kind of project, from CMSs to e-commerce apps to on-demand delivery platforms.

Secure: Django also has support to prevent common security issues, including cross-site request forgery, cross-site scripting, SQL injection, and clickjacking.

Scalable: Django websites can scale fast to meet high traffic demands.

Portable: runs on many platforms (Linux, Windows, Mac)

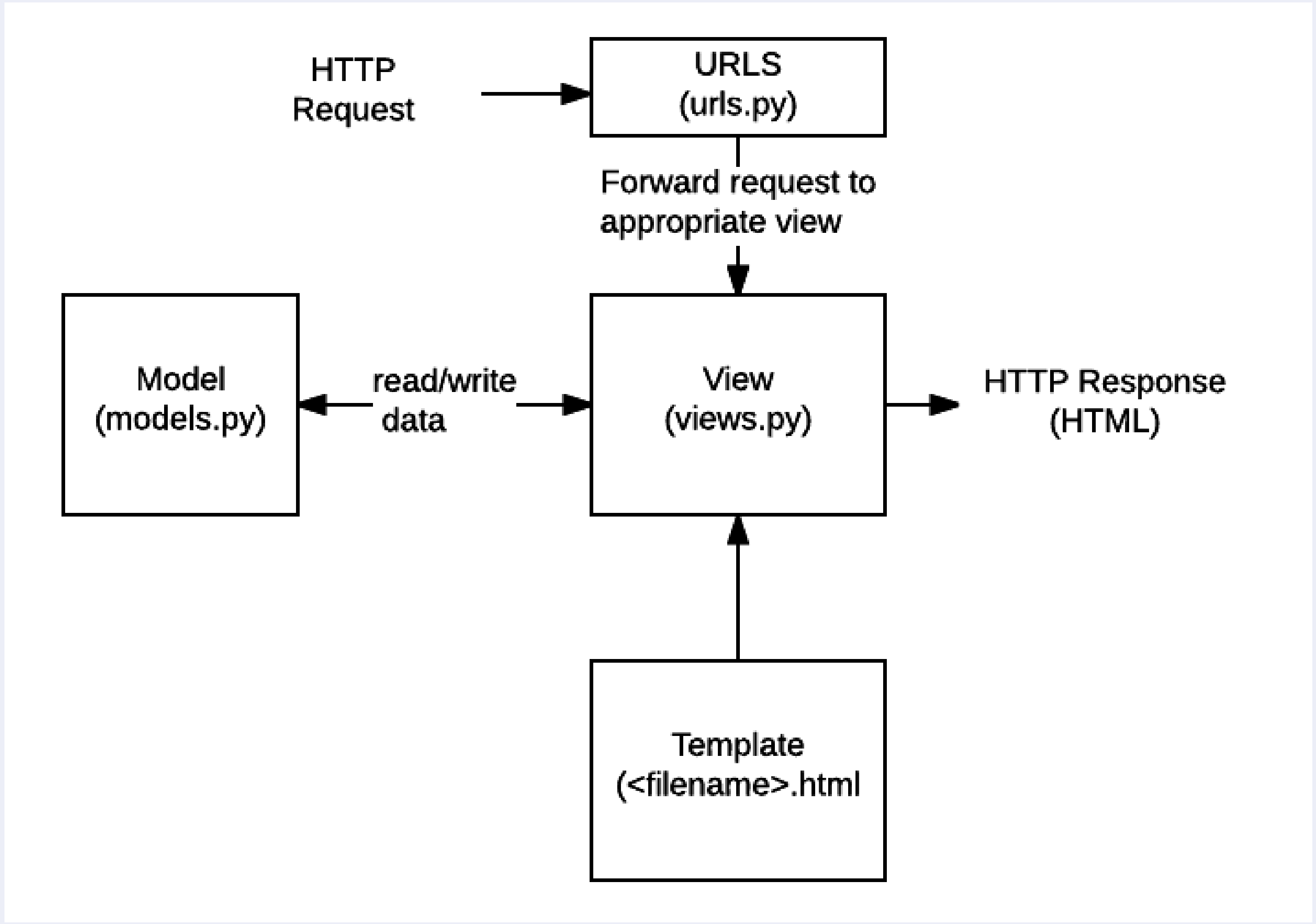
DJANGO ARCHITECTURE

URLs: A URL mapper is used to redirect HTTP requests to the appropriate view based on the request URL.

View: A view is a request handler function, which receives HTTP requests and returns HTTP responses.

Models: Models are Python objects that define the structure of an application's data, and provide mechanisms to manage (add, modify, delete) and query records in the database.

Templates: is a text file defining the structure or layout of a file (such as an HTML page)

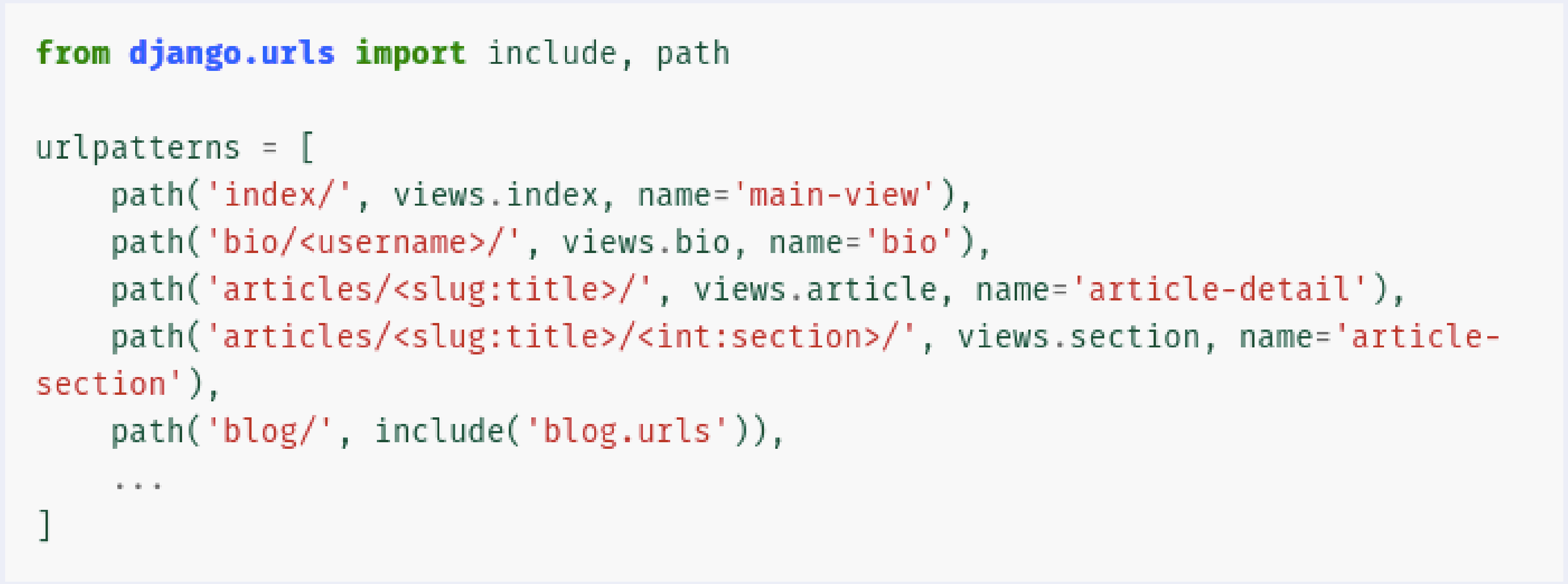


URLS

A URL mapper is typically stored in a file named **urls.py**

Urlpatterns defines a list of mappings between *routes* (specific URL *patterns)* and corresponding view functions.

path(*route*, *view*, *kwargs*= *None*, *name*=*None*)



```
from django.urls import include, path

urlpatterns = [
    path('index/', views.index, name='main-view'),
    path('bio/<username>/', views.bio, name='bio'),
    path('articles/<slug:title>/', views.article, name='article-detail'),
    path('articles/<slug:title>/<int:section>/', views.section, name='article-
section'),
    path('blog/', include('blog.urls')),
    ...
]
```

VIEWS

Views are the heart of the web application

Receive HTTP requests from web clients and returning HTTP responses

Types of Django Views:

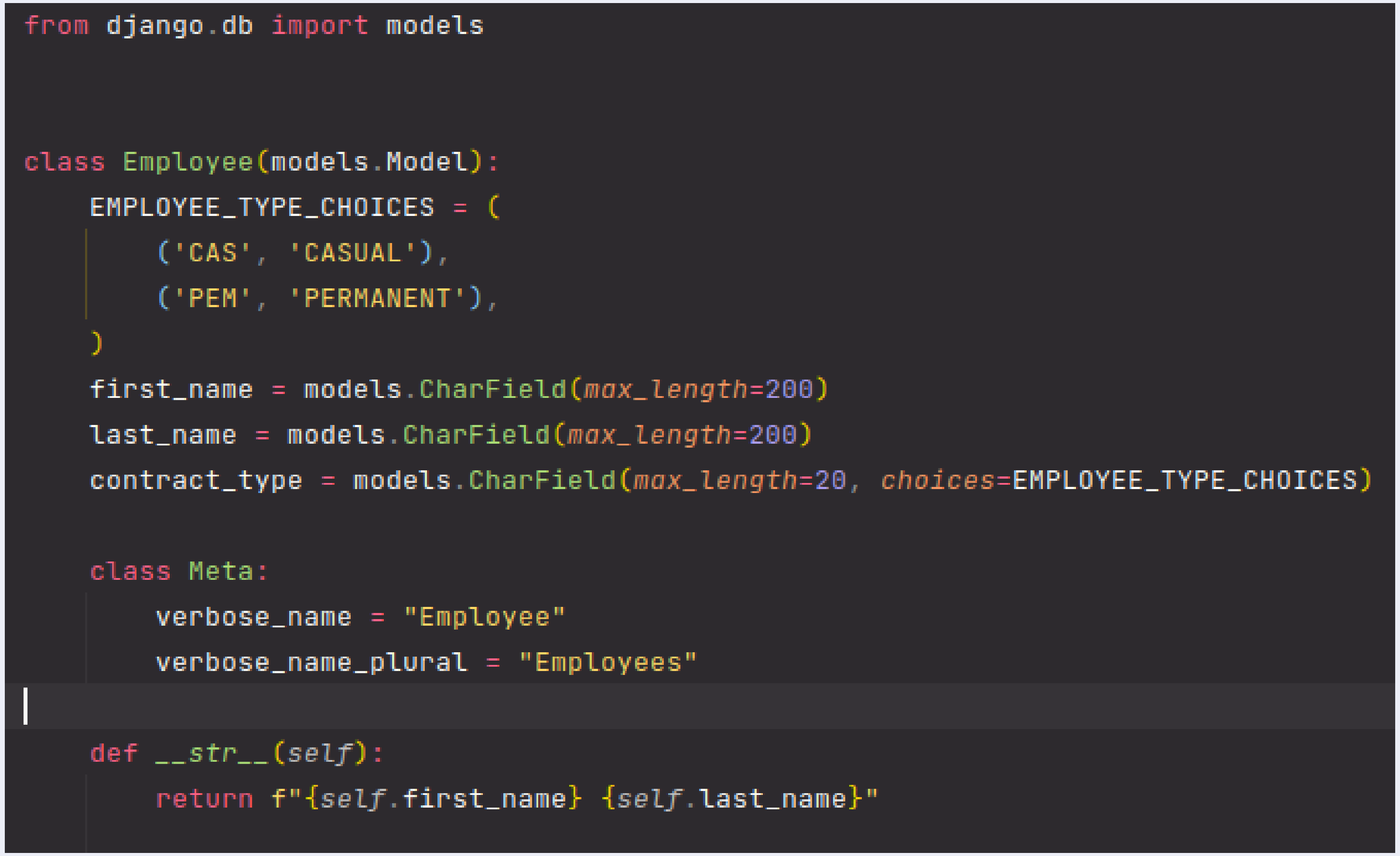
- **Function Based Views:** FBV take a standard Python function form of accepting a **request** object as an argument and returning a **response** object
- **Class Based Views:** based on python classes, CBV provide an alternative way of creating views out of a collection of methods that can be subclassed and overridden to provide common views of data without having to write too much code

MODELS

Models define the structure of stored data

including the field *types* and possibly also their maximum size, default values, selection list options, help text for documentation, label text for forms, etc

The definition of the model is independent of the underlying database

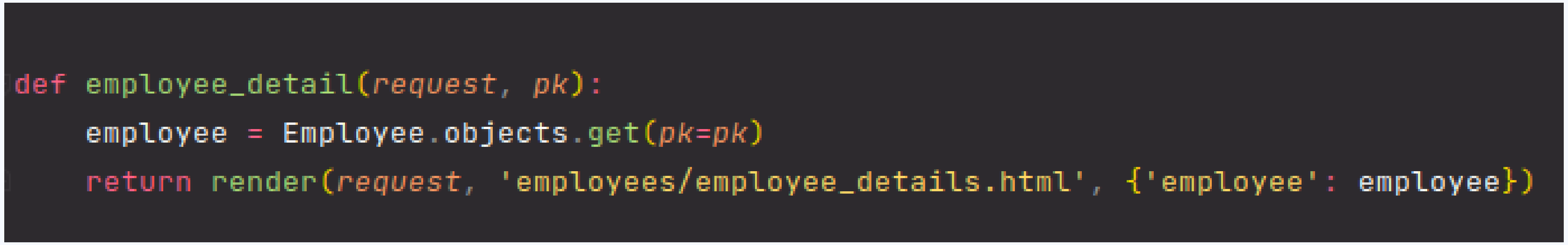


```python
from django.db import models


class Employee(models.Model):
    EMPLOYEE_TYPE_CHOICES = (
        ('CAS', 'CASUAL'),
        ('PEM', 'PERMANENT'),
    )
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)
    contract_type = models.CharField(max_length=20, choices=EMPLOYEE_TYPE_CHOICES)

    class Meta:
        verbose_name = "Employee"
        verbose_name_plural = "Employees"

    def __str__(self):
        return f"{self.first_name} {self.last_name}"
```

ACCESSING MODELS IN VIEW

Django automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects

More on django models in the Django Models section.



```
def employee_detail(request, pk):
    employee = Employee.objects.get(pk=pk)
    return render(request, 'employees/employee_details.html', {'employee': employee})
```

resource: More on making queries in Django

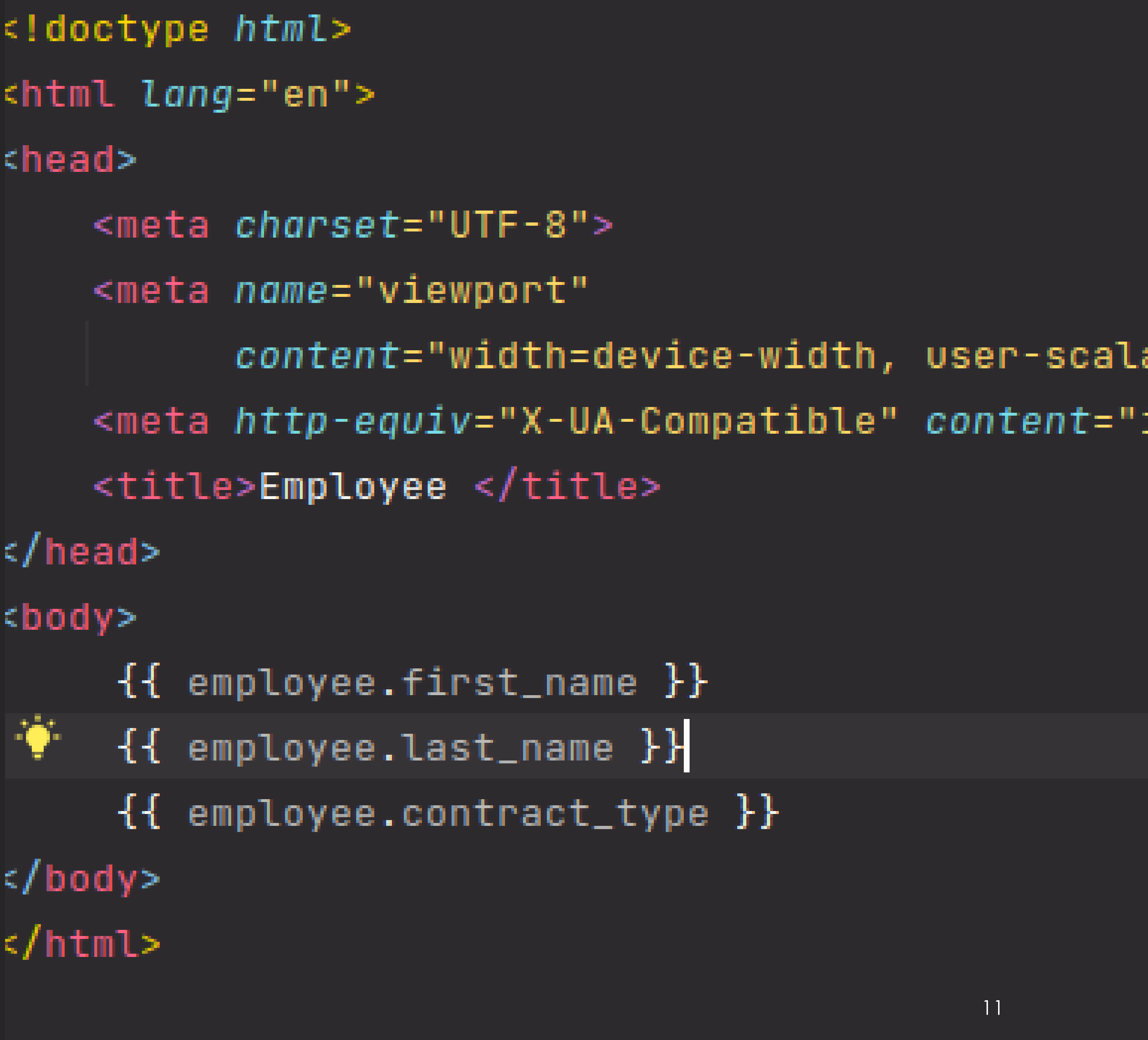
TEMPLATES

Django's template engine provides a powerful mini-language for:

- defining the user-facing layer of your application,
- encouraging a clean separation of application and presentation logic.

Templates not only show static data but also the data from different databases connected to the application through a context dictionary

Templates can be maintained by anyone with an understanding of HTML



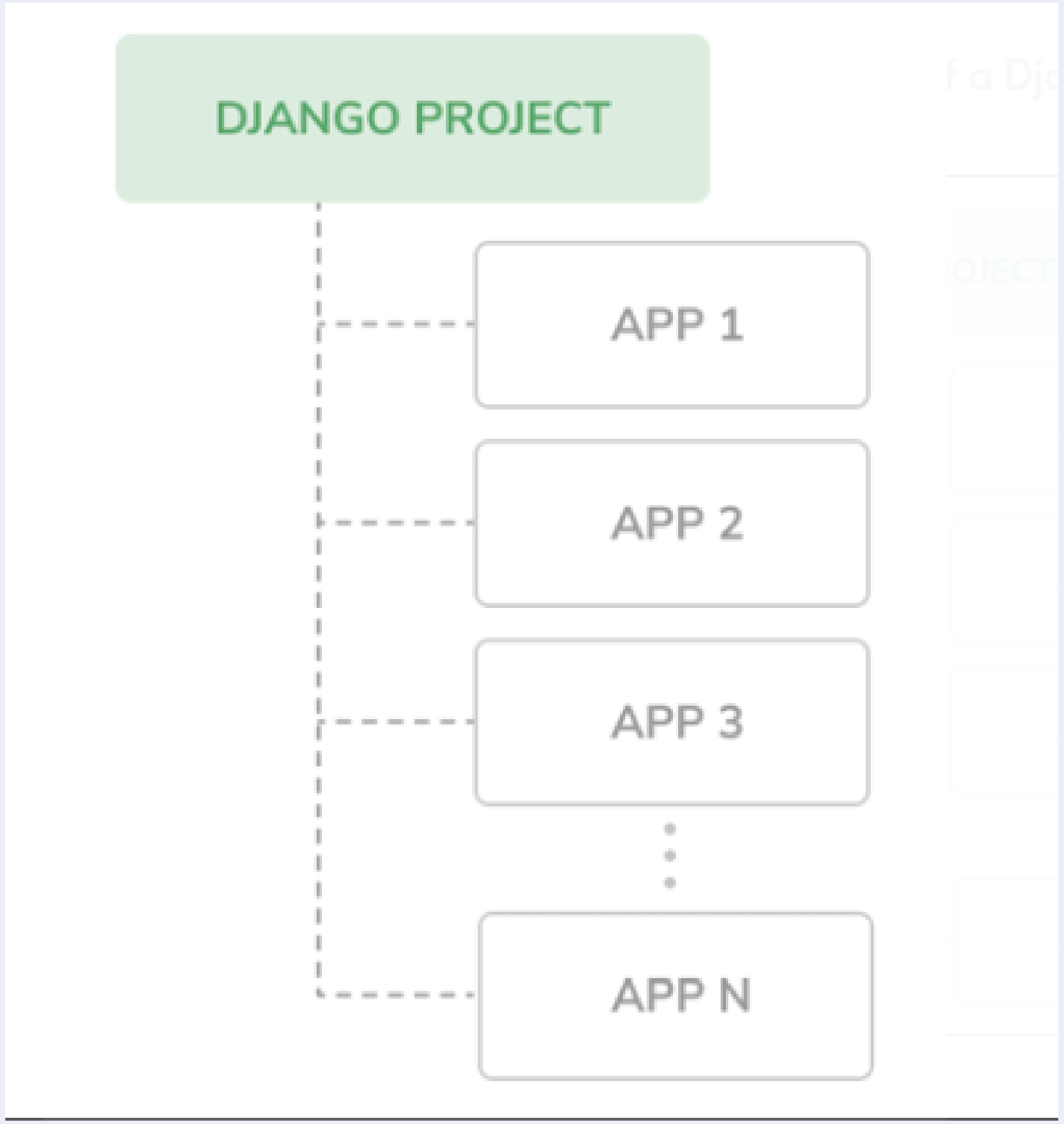
```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
          content="width=device-width, user-scala
    <meta http-equiv="X-UA-Compatible" content="
    <title>Employee </title>
</head>
<body>
    {{ employee.first_name }}
    {{ employee.last_name }}
    {{ employee.contract_type }}
</body>
</html>
```

DJANGO PROJECT VS DJANGO APP (APPLICATION)

A project is considered a Django installation with some settings.

An application is a group of models, views, templates, and URLs.

Applications interact with the framework to provide some specific functionalities and may be reused in various projects



DJANGO PROJECT

Django provides a command that allows you to create an initial project file structure

django-admin startproject <*project_name*>

manage.py: This is a command-line utility used to interact with your project. It is a thin wrapper around the django-admin.py tool. You don't need to edit this file.

Project_name/: This is your project directory, which consists of the following files:

- **__init__.py:** An empty file that tells Python to treat the project_name directory as a Python module.
- **asgi.py:** This is the configuration to run your project as ASGI, the emerging Python standard for asynchronous web servers and applications.
- **settings.py:** This indicates settings and configuration for your project and contains initial default settings.
- **urls.py:** This is the place where your URL patterns live. Each URL defined here is mapped to a view.
- **wsgi.py:** This is the configuration to run your project as a **Web Server Gateway Interface** (**WSGI**) application.

DJANGO APPLICATION

To create Django app we use the tool (**manage.py**)

python manage.py startapp <app_name>

Created app comes with these files:

admin.py: This is where you register models to include them in the Django administration site—using this site is optional.

apps.py: This includes the main configuration of the blog application.

migrations: This directory will contain database migrations of your application. Migrations allow Django to track your model changes and synchronize the database accordingly.

models.py: This includes the data models of your application; all Django applications need to have a models.py file, but this file can be left empty.

tests.py: This is where you can add tests for your application.

views.py: The logic of your application goes here; each view receives an HTTP request, processes it, and returns a response.

CODING DEMO

DJANGO MODELS

Django Models Documentation

OUTLINES

Designing Django Models

Django Fields

Django Migrations

Relationships

WHAT ARE DJANGO MODELS

A model is the single, definitive source of information about your data.

It contains the essential fields and behaviors of the data you're storing

- Each model is a Python class that subclasses **django.db.models.Model**.
- Each attribute of the model represents a database field
- With all of this, Django gives you an automatically-generated database-access API

To use Django models, the containing app has to be registered int the **INSTALLED_APPS** from the settings.py

By default, Django gives each model an auto-incrementing primary key, If you'd like to specify a custom primary key, specify **primary_key=True** on one of your fields.

DJANGO MODELS

Fields

- Django models are required to have fields
- Fields are specified by class attributes

More on field types

Field Types: Each field in your model should be an instance of the appropriate **Field** class

Tells the database what type of Data to store

Field Options: Each field takes a certain set of field-specific arguments (required, CharField, null, choices more on field options

DJANGO RELATIONSHIPS

The power of relational databases lies in relating tables to each other

Django offers ways to define the three most common types of database relationships:

One-to-one: implies that one record is associated with another record.

Many-to-one: implies that one model record can have many other model records associated with itself.

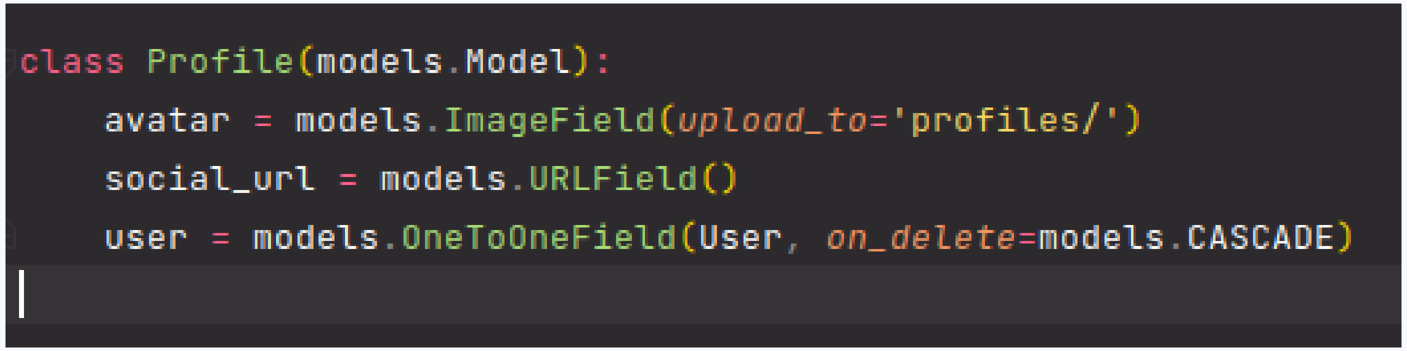
Many-to-many: implies that many records can have many other records associated amongst one another

ONE TO ONE RELATIONSHIP

To define a one-to-one relationship, use **OneToOneField**

OneToOneField requires a positional argument: the class to which the model is related

OneToOneField classes used to automatically become the primary key on a model



```
class Profile(models.Model):
    avatar = models.ImageField(upload_to='profiles/')
    social_url = models.URLField()
    user = models.OneToOneField(User, on_delete=models.CASCADE)
```

MANY TO ONE RELATIONSHIP

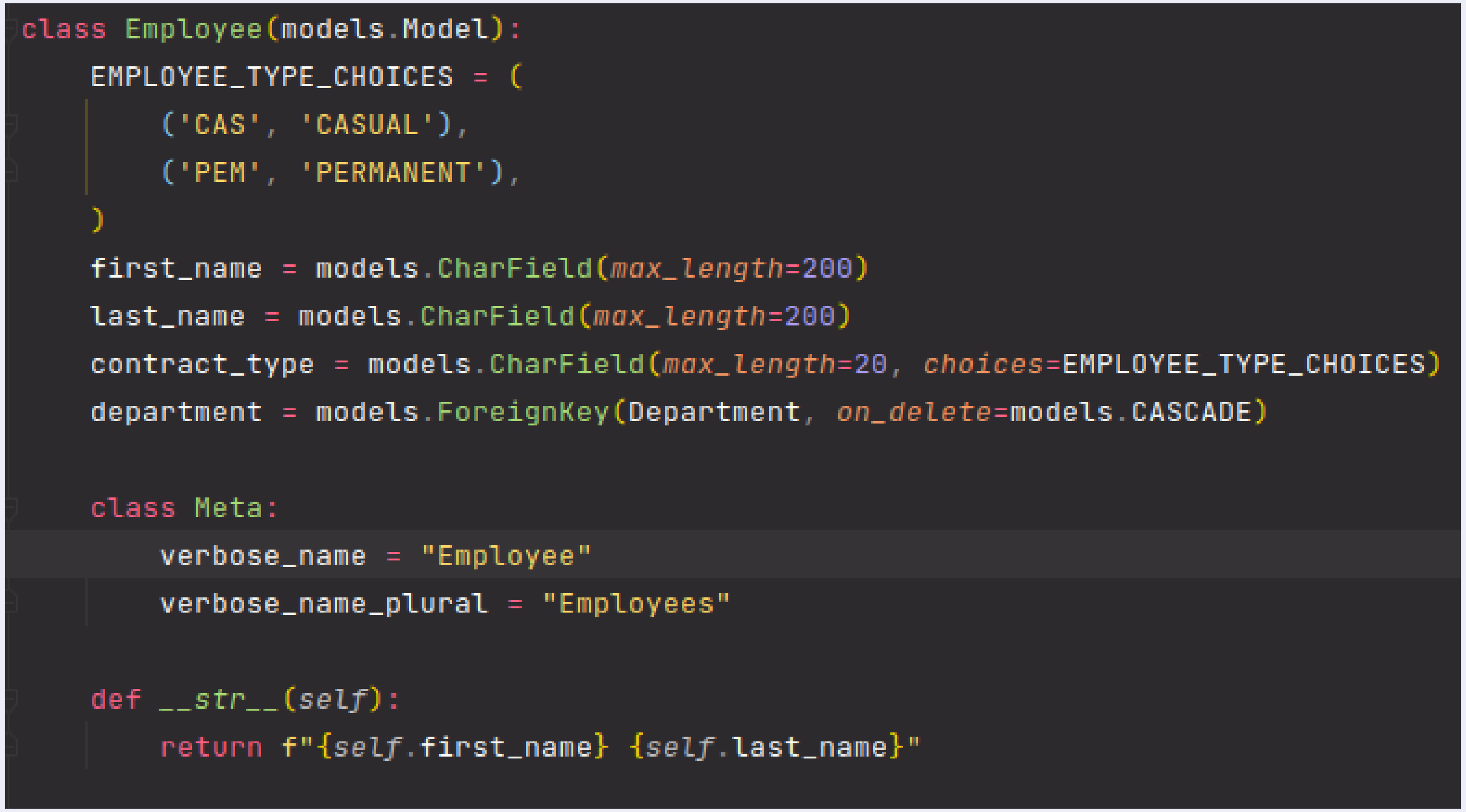
To define a many-to-one relationship, use **django.db.models.ForeignKey**

ForeignKey requires two positional arguments: the class to which the model is related and **"on_delete"** for the referential integrity

Many to one relationship can also be defined recursively

To create a recursive relationship – an object that has a many-to-one relationship with itself – use **models.ForeignKey('self', on_delete= models.CASCADE)**

More on Many to One Relationship



```
class Employee(models.Model):
    EMPLOYEE_TYPE_CHOICES = (
        ('CAS', 'CASUAL'),
        ('PEM', 'PERMANENT'),
    )
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)
    contract_type = models.CharField(max_length=20, choices=EMPLOYEE_TYPE_CHOICES)
    department = models.ForeignKey(Department, on_delete=models.CASCADE)

    class Meta:
        verbose_name = "Employee"
        verbose_name_plural = "Employees"

    def __str__(self):
        return f"{self.first_name} {self.last_name}"
```

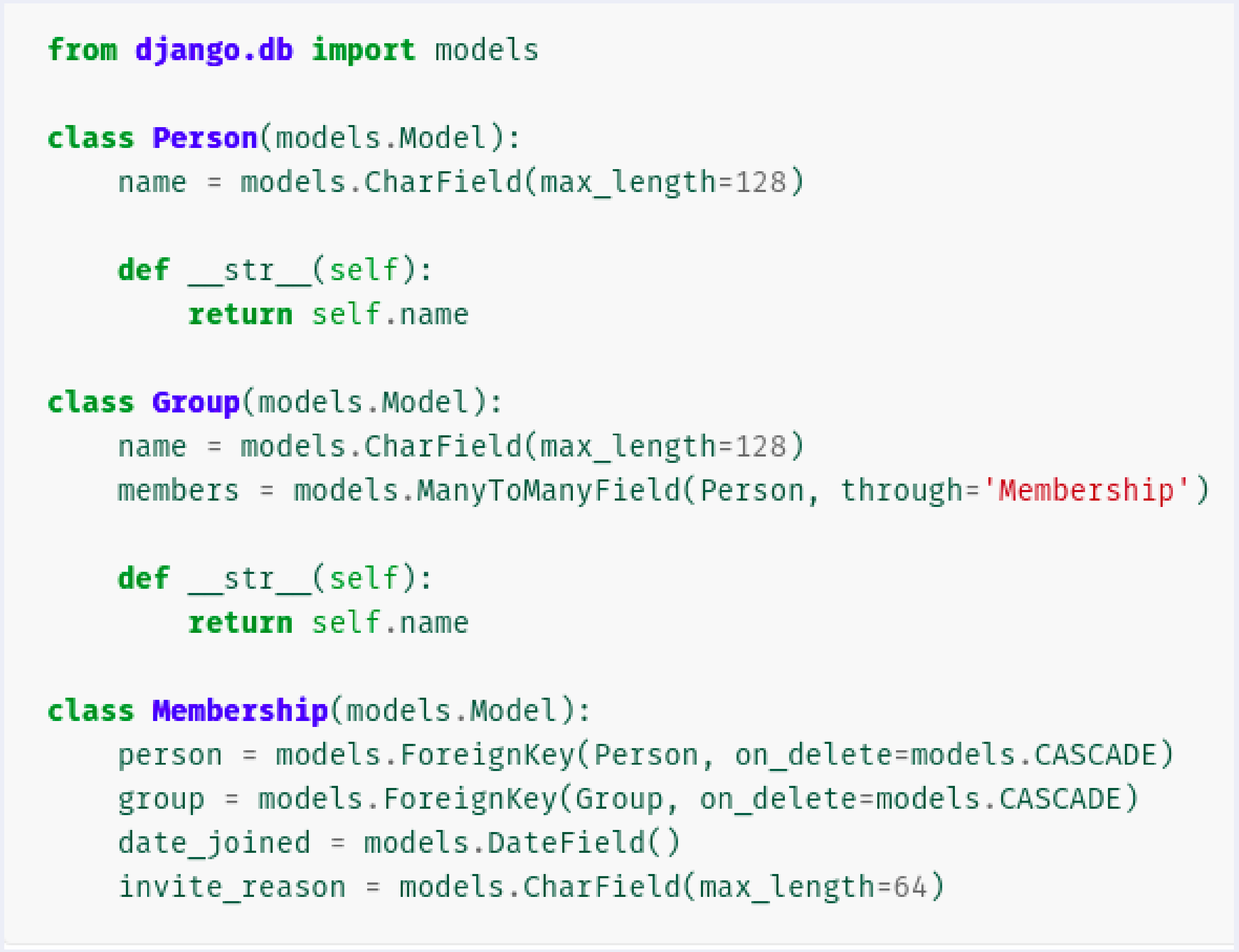
MANY TO MANY RELATIONSHIP

To define a many-to-many relationship, use **ManyToManyField**

ManyToManyField requires a positional argument: the class to which the model is related.

Behind the scenes, Django creates an intermediary join table to represent the many-to-many relationship.

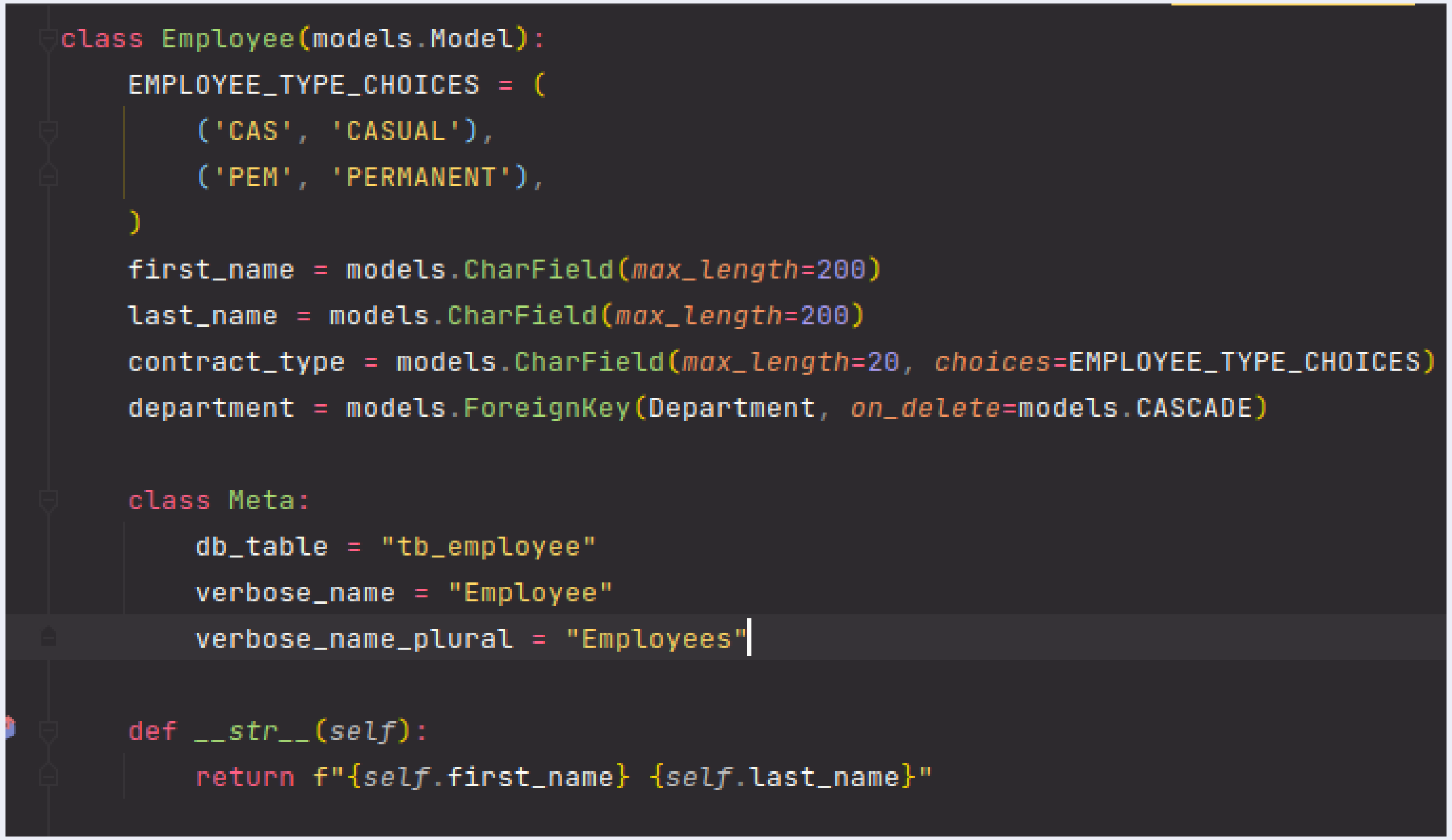
Click for more on Many to Many relationship



```python
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

    def __str__(self):
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

META OPTIONS

Django provides a way to add metadata to a model by using an inner **class Meta**, like so:



```
class Employee(models.Model):
    EMPLOYEE_TYPE_CHOICES = (
        ('CAS', 'CASUAL'),
        ('PEM', 'PERMANENT'),
    )
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)
    contract_type = models.CharField(max_length=20, choices=EMPLOYEE_TYPE_CHOICES)
    department = models.ForeignKey(Department, on_delete=models.CASCADE)

    class Meta:
        db_table = "tb_employee"
        verbose_name = "Employee"
        verbose_name_plural = "Employees"

    def __str__(self):
        return f"{self.first_name} {self.last_name}"
```

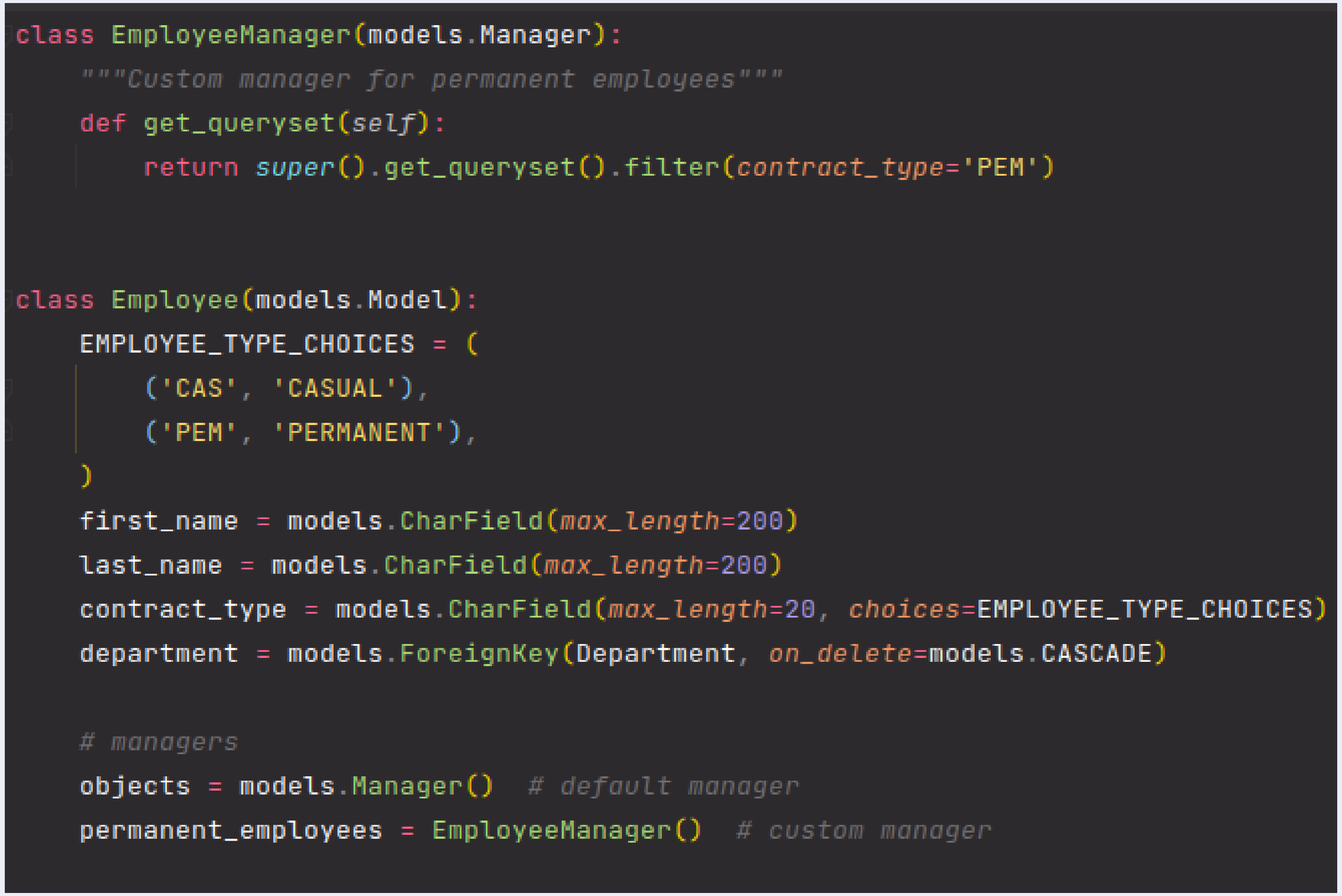
MODEL ATTRIBUTES

The most important attribute of a model is the **Manager**.

It's the interface through which database query operations are provided to Django models

At least one **Manager** exists for every model in a Django application

If no custom Manager is defined, the default name is **objects**.



```
class EmployeeManager(models.Manager):
    """Custom manager for permanent employees"""
    def get_queryset(self):
        return super().get_queryset().filter(contract_type='PEM')


class Employee(models.Model):
    EMPLOYEE_TYPE_CHOICES = (
        ('CAS', 'CASUAL'),
        ('PEM', 'PERMANENT'),
    )
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)
    contract_type = models.CharField(max_length=20, choices=EMPLOYEE_TYPE_CHOICES)
    department = models.ForeignKey(Department, on_delete=models.CASCADE)

    # managers
    objects = models.Manager()  # default manager
    permanent_employees = EmployeeManager()  # custom manager
```

MODEL METHODS

custom methods on a model to add custom “row-level” functionality to your objects

Unlike **Manager** methods are intended to do “table-wide” things, model methods should act on a particular **model instance**

Model methods helps in keeping business logic in one place(model), though this is not considered as a best practice approach from the software architecture standpoint.

It's possible also to override the predefined model methods (like save, clean,..)

Click here for more model methods in Django

MODEL INHERITANCE

Model inheritance in Django works almost identically to the way normal class inheritance works in Python

There are three styles of inheritance that are possible in Django

- Abstract Base Classes (click for more)
- Multi-table Inheritance (click for more)
- Proxy Models (click for more)

DJANGO MIGRATIONS

WHAT ARE MIGRATIONS

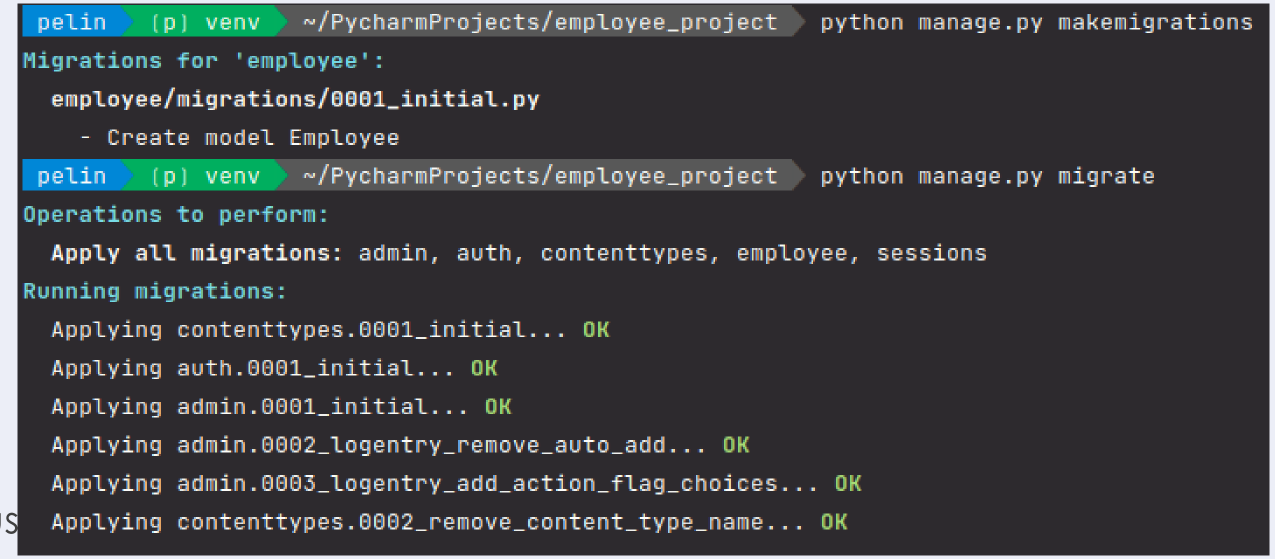
Migrations are Django's way of propagating changes you make to your models into your database schema.

Migrations commands:

- **Migrate:** which is responsible for applying and unapplying migrations.
- **Makemigrations:** responsible for creating new migrations based on the changes you have made to your models.
- **Sqlmigrate:** which displays the SQL statements for a migration
- **Showmigrations:** which lists a project's migrations and their status
- **Squashingmigrations:** reducing an existing set of many migrations down to one

The migration files for each app live in a "migrations" directory inside of that app

Django will make migrations for any change to your models or fields - even options that don't affect the database



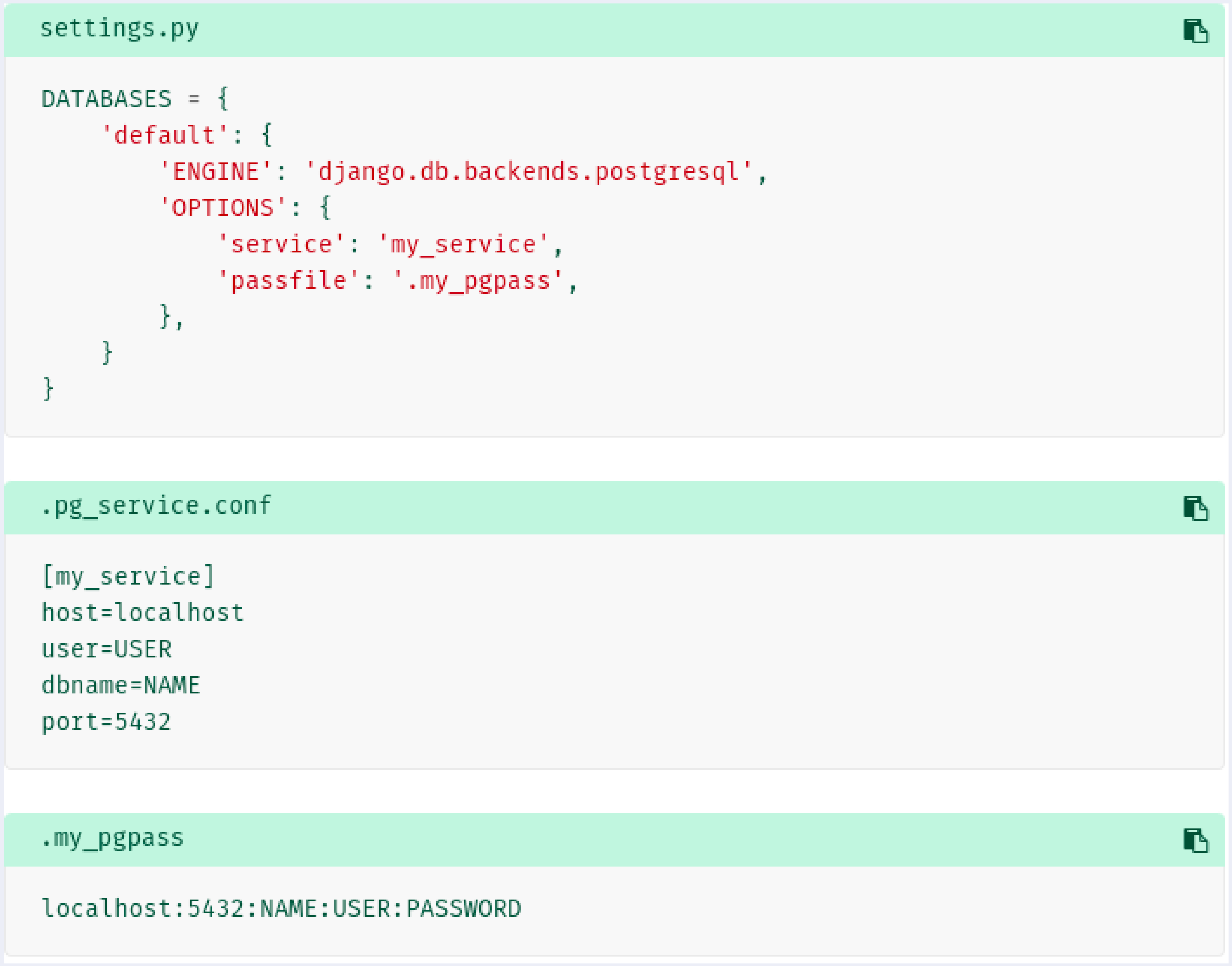
```
pelin  (p) venv  ~/PycharmProjects/employee_project  python manage.py makemigrations
Migrations for 'employee':
  employee/migrations/0001_initial.py
    - Create model Employee
pelin  (p) venv  ~/PycharmProjects/employee_project  python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, employee, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
```

DJANGO DATABASES SUPPORT

Django support a number of databases systems:

- **PostgreSQL**
- **MySQL**
- **SQLite**

Click for more configurations



settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'OPTIONS': {
            'service': 'my_service',
            'passfile': '.my_pgpass',
        },
    }
}
```

.pg_service.conf

```
[my_service]
host=localhost
user=USER
dbname=NAME
port=5432
```

.my_pgpass

```
localhost:5432:NAME:USER:PASSWORD
```

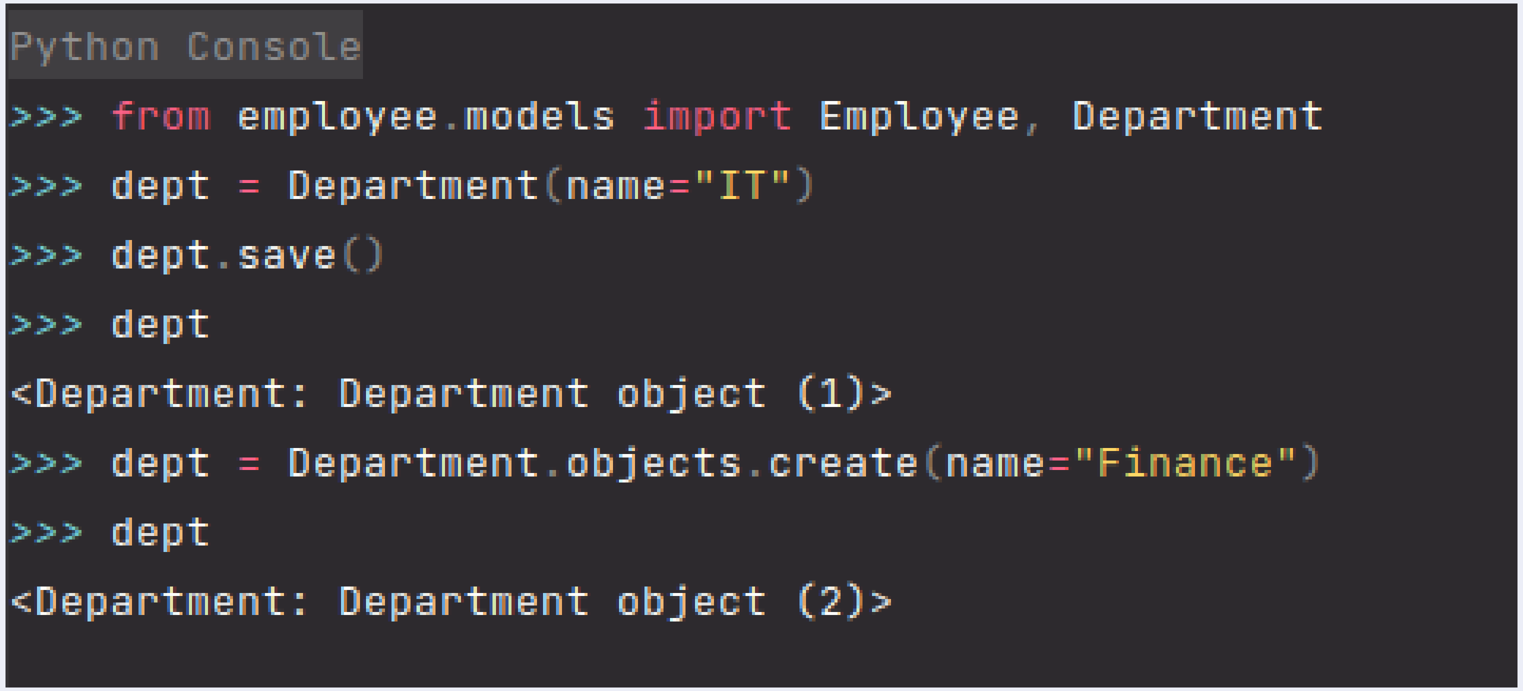
CODING DEMO

MAKING QUERIES IN DJANGO

CREATING OBJECTS

A model class represents a database table, and an instance of that class represents a particular record in the database table

To create an object, instantiate it using keyword arguments to the model class, then call **save()** to save it to the database



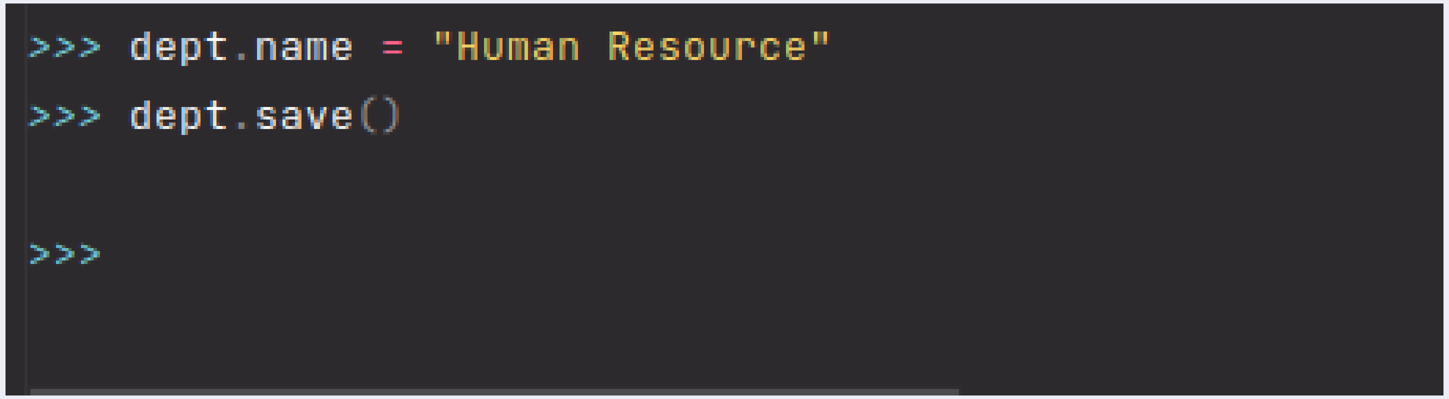
Python Console

```
>>> from employee.models import Employee, Department
>>> dept = Department(name="IT")
>>> dept.save()
>>> dept
<Department: Department object (1)>
>>> dept = Department.objects.create(name="Finance")
>>> dept
<Department: Department object (2)>
```

UPDATING OBJECTS

To save changes (update) to an object that's already in the database, use **save()**.

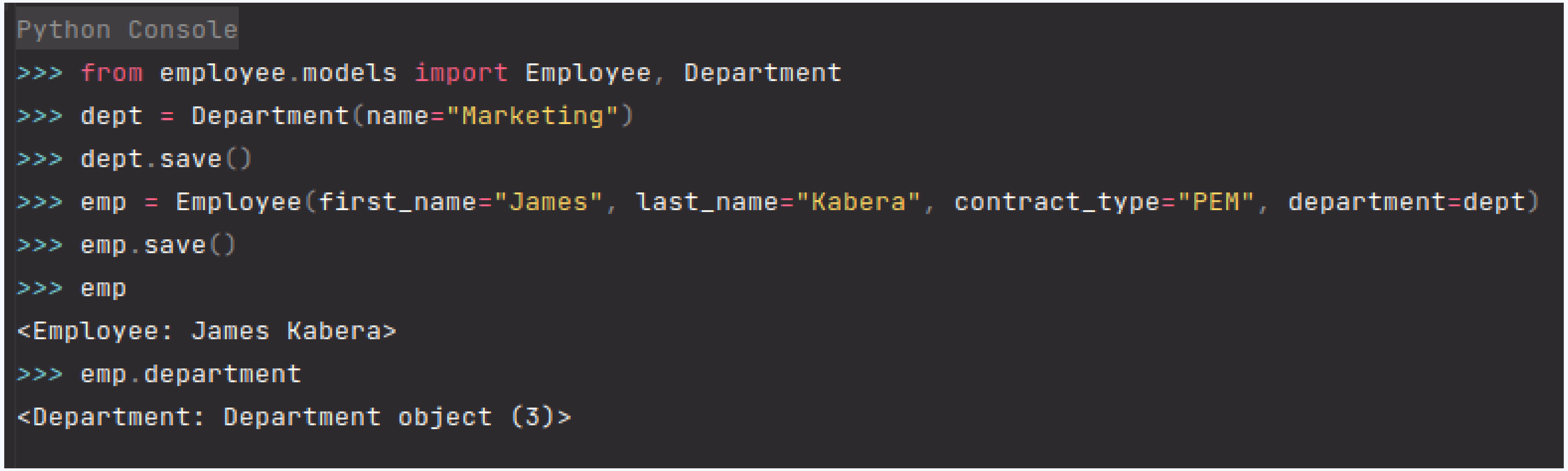
Django doesn't hit the database until you explicitly call **save()**



```
>>> dept.name = "Human Resource"
>>> dept.save()

>>>
```

CREATING RELATED OBJECTS

Creating/Updating related object works exactly the same way as saving a normal field – assign an object of the right type to the field in question



```
Python Console
>>> from employee.models import Employee, Department
>>> dept = Department(name="Marketing")
>>> dept.save()
>>> emp = Employee(first_name="James", last_name="Kabera", contract_type="PEM", department=dept)
>>> emp.save()
>>> emp
<Employee: James Kabera>
>>> emp.department
<Department: Department object (3)>
```

RETRIEVING OBJECTS

To retrieve objects from your database, construct a **QuerySet** via a **Manager** on your model class

A **QuerySet** represents a collection of objects from your database. It can have zero, one or many *filters*.

You get a **QuerySet** by using your model's **Manager**. (the default one called **objects** or a custom one)

QuerySets are lazy – the act of creating a **QuerySet** doesn't involve any database activity

Django will only run the query when the **QuerySet** is ***evaluated***

RETRIEVING OBJECTS

Retrieving specific objects with filters:

Retrieving specific objects with get:

Limiting QuerySets

Field lookups

Lookups that span relationships

Complex lookups with Q objects

More on Queries

CODING DEMO

THANK YOU