Advanced Queries in Django

SoftUni Team

Technical Trainers

Software University

https://softuni.bg

Table of Contents

Have a Question?

Software University

sli.do

#python-db

Custom Managers

Custom Manager

- In Django, a **manager** is an interface through which database **query operations** are **performed**
- By **default**, Django provides a **manager** called `objects` for every model
- You can create **custom managers** to
  - **encapsulate** specific query **logic**
  - make it **reusable** throughout your application

Custom Managers (2)

- **Custom managers** are **useful** when you want to **add**
  - **custom methods** and **filters**
  - to **retrieve data** from the database
- They allow you to define specialized **query sets** **tailored** to your application's **needs**
- To **create** a **custom manager**, you need to
  - **subclass** `models.Manager`
  - define your **custom methods** there

More at: https://docs.djangoproject.com/en/4.2/topics/db/managers/#custom-managers

Custom Manager Example

```
# models.py
from django.db import models

class EmployeeManager(models.Manager):
    def by_job_title(self, job_title):
        return self.filter(job_title=job_title)

class Employee(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    job_title = models.CharField(max_length=100)
    ...
    objects = EmployeeManager()
```

Custom method to retrieve employees with a specific job title

Employee model with the custom manager

Attach the custom manager to the Employee model

Custom Manager Example (2)

```
# caller.py
# Using the Custom Manager

# Retrieve all employees with the job title "Software Engineer"


def get_software_eng():
    software_engineers =
Employee.objects.by_job_title("Software Engineer")
    print(software_engineers)
```

Calling the custom manager's method

Problem: Available Products

- You are given an **ORM project skeleton** (you can download it from here) with predefined **Shop Management System**
- Create a custom manager called **`"ProductManager"`** for the model **`"Product"`** that extends the built-in model manager:
  - **`available_products`** - returns all **products** that are **currently available**
  - **`available_products_in_category(category_name: str)`** - returns all **products in a category** that are **currently available**

Solution: Available Products

```
class ProductManager(models.Manager):
    def available_products(self):
        return self.filter(is_available=True)

    def available_products_in_category(self, category_name):
        return self.filter(is_available=True,
                           category__name=category_name)


class Product(models.Model):
    ...
    objects = ProductManager()
```

Annotation

annotate()

Annotation

- **Annotation** in Django is a **powerful feature** that
  - allows you to **add calculated fields** to your **query results**
- The `annotate()` **method** is used
  - to **add** the **calculated fields** to the **queryset**
- **Annotation** can be **useful** when you need to **perform**
  - **aggregation** or **add derived values** to your model instances

Annotation (2)

- **Annotation** is a powerful **tool** that
  - **extends** your **query** **capabilities**
  - **allows** you to retrieve **aggregated** or **calculated** data **efficiently**
  - keeps your **model structure** **clean**
  - **separates** **model structure** from the **query logic**

Annotation Example (1)

```
# models.py

from django.db import models


class Employee(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    job_title = models.CharField(max_length=100)
    ...
```

Defining the Employee Model

Annotation Example (2)

```python
# caller.py

from django.db.models import Count
from .models import Employee

def count_per_job_title():
    employee_counts = 
Employee.objects.values('job_title').annotate(num_employees=
Count('id'))


    for entry in employee_counts:
        print(f"Job Title: {entry['job_title']}, Number of 
Employees: {entry['num_employees']}")
```

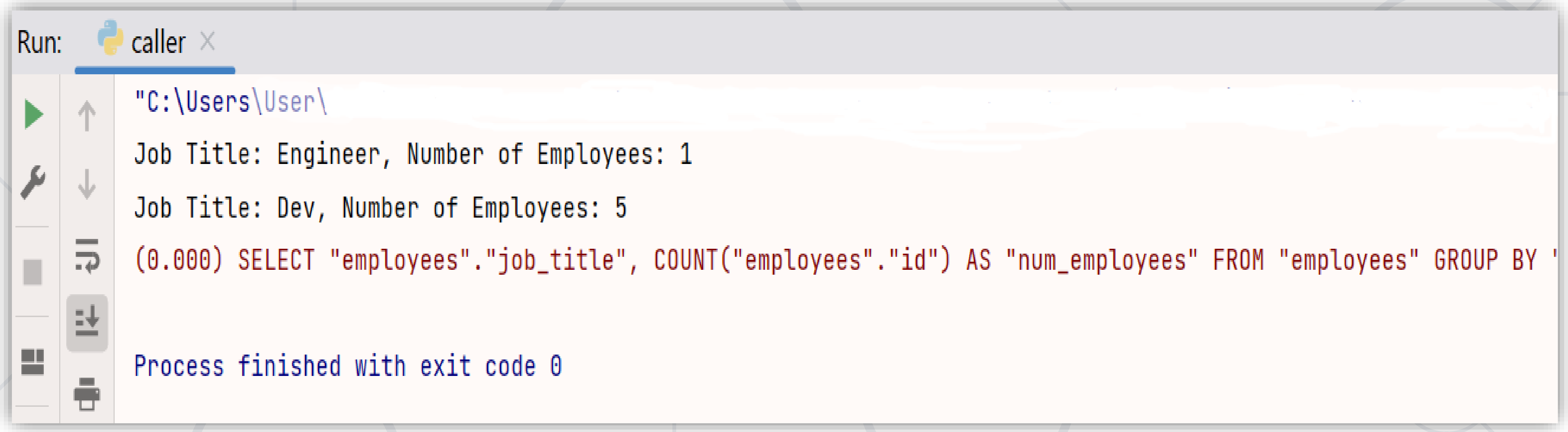
Count the number of employees per job title using annotation

The result is a queryset of dictionaries

Using the count aggregation function

Each entry is a dictionary

Annotation Example - Result



Run: caller

```
"C:\Users\User\
Job Title: Engineer, Number of Employees: 1
Job Title: Dev, Number of Employees: 5
(0.000) SELECT "employees"."job_title", COUNT("employees"."id") AS "num_employees" FROM "employees" GROUP BY '

Process finished with exit code 0
```

Problem: Product Quantity Ordered

- Create a function called **`"product_quantity_ordered"`** that returns a summary of the **total quantity ordered** for **each product** available in the store in the given format below:
  `"Quantity ordered of {product_name}: {total_ordered_quantity}"`
- Return only the information for **products that have at least one unit ordered**
- Arrange the information in **descending order based on the total quantity ordered**

Solution: Product Quantity Ordered

```
def product_quantity_ordered():
    total_products_ordered = (Product.objects.
annotate(total_ordered_quantity=Sum('orderproduct__quantity'))
                              .exclude(total_ordered_quantity=None)
                              .order_by('-total_ordered_quantity'))

    result = []
    for product in total_products_ordered:
        result.append(f"Quantity ordered of {product.name}:
{product.total_ordered_quantity}")
    return "\n".join(result)
```

Queries for Model Relationships

select_related(), prefetch_related()

Queries for Model Relationships

- **Specific methods** are used to **optimize** database **queries**
  - when dealing with **related objects** in your models
  - helping to **reduce** the **number** of **queries** executed
  - improving **performance**

Queries for Model Relationships (2)

- `select_related()`
  - Used to **optimize** queries involving `ForeignKey` and `OneToOneField` relationships
  - It fetches **related objects** in the same **same query**
    - rather than executing a separate query for each related object
  - Significantly **reduces** the **number** of database queries and **improves performance**

Queries for Model Relationships (3)

- `prefetch_related()`
  - Used for **optimizing** queries involving `ManyToManyField`, **reverse** `ForeignKey`, and **reverse** `OneToOneField` relationships
  - It fetches **related objects** in a separate query and **caches** them for **efficient** lookup
  - Helps to **avoid** the **N+1 query problem**, where N is the number of objects being queried

Queries for Relationships – Example (1)

```python
# models.py

from django.db import models

class Department(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()

class Employee(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    department = models.ForeignKey(Department, on_delete=models.SET_NULL,
null=True, related_name='employees')

class Project(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    employees = models.ManyToManyField(Employee)
```

Queries for Relationships – Example (2)

```
# caller.py
# Using select_related() to fetch related department data efficiently

def select_employee(emp_id):

    selected_employee = 
Employee.objects.select_related('department').get(pk=emp_id)

    print(selected_employee.last_name)
    print(selected_employee.department.name)

   # No additional query is executed for the department
```

Queries for Relationships – Example (3)

```python
# caller.py
# Accessing related objects using the related_name attribute

def get_employees_per_department(dep_id):
    selected_department = Department.objects.get(pk=dep_id)

    employees_in_department = selected_department.employees.all()

    print(selected_department.name)

    for employee in employees_in_department:
        print(f"- {employee.first_name} {employee.last_name}")
```

Queries for Relationships – Example (4)

```
# caller.py
# Using prefetch_related() with related_name

def get_departments_with_employees():

    departments_with_employees =
Department.objects.prefetch_related('employees').all()

    for department in departments_with_employees:
        print(department.name)
        for employee in department.employees.all():
            print(f"- {employee.first_name} {employee.last_name}")
```

Using the related name 'employees'

Using the related name 'employees'

Queries for Relationships – Example (5)

```
# caller.py
# Using prefetch_related() to fetch related projects efficiently for all employees

def employees_with_projects():
    employees_with_related_projects = Employee.objects.prefetch_related(
    'project_set', 'project_set__employees').all()

    for employee in employees_with_related_projects:
        print(f"Employee: {employee.first_name} {employee.last_name}")

        print("Projects:")
        for project in employee.project_set.all():
            print(f"- {project.name}")
```

Using the default related name 'project_set'

Printing projects for each employee

Problem: Ordered Products Per Customer

- Create a function called **`"ordered_products_per_customer"`** that returns a summary of **each ordered product by each customer** in the given format below:

```
"Order ID: {order_id1}, Customer: {customer_username1}
 - Product: {product_name1}, Category: {category_name1}
…"
```

- Arrange the information in **ascending order by the order ID**

Solution: Ordered Products Per Customer

```
def ordered_products_per_customer():
    prefetched_orders = 
Order.objects.prefetch_related('orderproduct_set__product__category')
.order_by('id')

    result = []
    for order in prefetched_orders:
        result.append(f"Order ID: {order.id}, Customer: 
{order.customer.username}")
        for order_product in order.orderproduct_set.all():
            result.append(f"- Product: {order_product.product.name}, 
Category: {order_product.product.category.name}")

    return "\n".join(result)
```

Query-related Tools

Q and F Objects

Q object

- **Q object** is a powerful **tool** that
  - allows you to build **complex** queries
  - by **combining multiple conditions**
  - using **logical operators**
- It's especially useful when you need
  - to create **dynamic** queries
  - with **various conditions**
  - combined in a **flexible** way

Q object (2)

- The **Q object** is part of Django's query **expression** system
  - Provides a more **programmatic** approach to constructing queries
  - Uses **logical** operators like
    - **AND** (**&**), **OR** (**|**), **NOT** (**~**), and **XOR** (**^**)
- You can create **instances** of the **Q object** with **conditions**
  - use them to construct **more complex** queries

Q object Example

```python
from django.db.models import Q
from .models import Employee

# Using Q object to construct complex queries

def filter_employees_q_obj():
    query = Q(department=1) | Q(job_title='Dev')
    filtered_employees = Employee.objects.filter(query)

    for employee in filtered_employees:
        print(f"{employee.first_name} {employee.last_name}")
```

OR operator

Q object Example (2)

```
from django.db.models import Q
from .models import Employee

# Using Q object for a more complex query

def filter_employees_q_obj_complex():
    query = Q(first_name__startswith='J') & (Q(department=2) |
Q(job_title='Manager'))
    filtered_employees = Employee.objects.filter(query)

    for employee in filtered_employees:
        print(f"{employee.first_name} {employee.last_name}")
```

AND operator

Problem: Available Products Prices

- Create a function called "**`filter_products`**" that returns information for **all available products** in the store **with prices greater than 3.00** in the format:

```
"{product_name1}: {product_price1}lv.
 …
 {product_nameN}: {product_priceN}lv.}"
```

- Arrange the information in **descending order by the price**
  - If there are **two or more products with the same price**, order them by **name in ascending order** (alphabetically)

Solution: Available Products Prices

```
def filter_products():
    query = Q(is_available=True) & Q(price__gt=3.00)
    products = Product.objects.filter(query).order_by('-price', 'name')
    result = []
    for product in products:
        result.append(f"{product.name}: {product.price}lv.")

    return "\n".join(result)
```

F object

- **F object** is a **tool** that allows you to
  - reference a **field's value** in a query **expression**
- It's **useful** for performing **operations**
  - involving the **values** of **fields**
  - **within** the **database query** itself
  - does **not fetch** the values
  - does **not perform** the operations in Python code

F object (2)

- Using the **F object**
  - You can **compare** and **manipulate field values directly** in the database **query**
    - **comparing** the **values** of **two fields**
    - **updating fields** with **other fields' values**
  - Leads to **more efficient** and **optimized** queries

F object Example

```
# models.py
class Employee:
    salary = models.FloatField(default=1.00)
    ...
```

```
# caller.py
from django.db.models import F
from .models import Employee

# Using F object to update field values
def update_salary_f_obj():
    Employee.objects.update(salary=F('salary') * 1.1)
```

The value of salary field

F object Example (2)

```python
from django.db.models import F, Avg
from .models import Employee

# Using F object for a more complex query

def above_avg_f_obj():
    employees_above_avg_salary = Employee.objects.annotate(
    avg_department_salary=Avg('department__employees__salary')
).filter(salary__gt=F('avg_department_salary'))

    for employee in employees_above_avg_salary:
        print(f"{employee.first_name} {employee.last_name} - 
Salary Above Average!")
```

Calculates the average salary within each department

Follows the relationship chain

Problem: Give Discounts

- Create a function called **`"give_discount"`** that reduces the product's price by 30% of **all available products with prices greater than 3.00**
- Then, it returns information about **all available products** and **their prices** in the following format:
  `"{product_name}: {product_price}lv."`
- Arrange the information in **descending order by the price**
  - If there are **two or more products with the same price**, order them by **name in ascending order** (alphabetically)

Solution: Give Discounts

```python
def give_discount():
    reduction = F('price') * 0.7
    query = Q(is_available=True) & Q(price__gt=3.00)
    Product.objects.filter(query).update(price=reduction)
    all_available_products = (Product.objects.
                              filter(is_available=True).
                              order_by('-price', 'name'))
    result = []
    for product in all_available_products:
        result.append(f"{product.name}: {product.price}lv.")

    return "\n".join(result)
```

Debugging Queries

Debugging Queries

- There are several popular **tools** and **libraries** for **debugging queries** in Django
- These **tools** help you
  - **analyze** and **optimize** the **SQL queries** generated by Django ORM
  - provide **different levels** of **insights** into your application's query **performance**
- The choice of tool depends on your **preferences** and the **depth** of **analysis** you require

***Note**: Debugging tools are invaluable during development, you should **avoid** using them in **production** environments due to **security concerns** and **performance overhead**

Debugging Tools

- **Django Debug Toolbar**
  - Provides an interactive panel on your website that displays various information, including SQL queries, query execution time, cache usage, and more
- **Silk**
  - Offers a graphical interface to inspect executed queries, view query execution time, and analyze other aspects of your application's performance
- **django-querycount**
  - Lightweight tool that simply prints the number of database queries executed for a specific view

django-extensions Tool

- django-extensions
  - A third-party Django **package**
  - Provides **various** useful **utilities** and **extensions**
    - **enhanced** query debugging capabilities
    - **easier** to understand and analyze the SQL queries generated by Django ORM

Using django-extensions

- **Installation**

```
pip install django-extensions
```

  - Add **'django_extensions'** to your **INSTALLED_APPS** list in your project's **settings**

- **Shell Plus**

```
python manage.py shell_plus
```

  - An **enhanced** version of the **Django shell** called **Shell Plus**
  - **Automatically imports** your **models** and commonly used **packages**
  - **Saving** you **time** when **experimenting** and **debugging**

Using django-extensions (2)

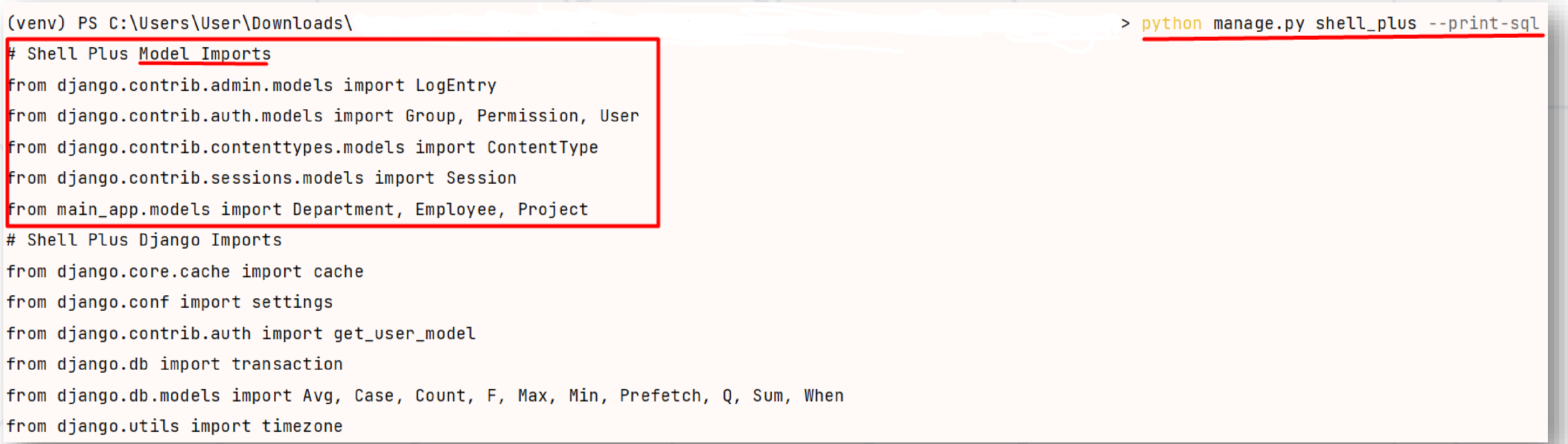
- **Printing SQL Queries**

```
python manage.py shell_plus --print-sql
```

  - This command will print SQL queries as they are executed in `shell_plus`
    - along with the **execution time** and **database** used
  - It's a great way to **identify** any potential **performance issues**
- **Other Utilities**
  - **django-extensions** offers **various** other **utilities** such as graph generation, template rendering, and more

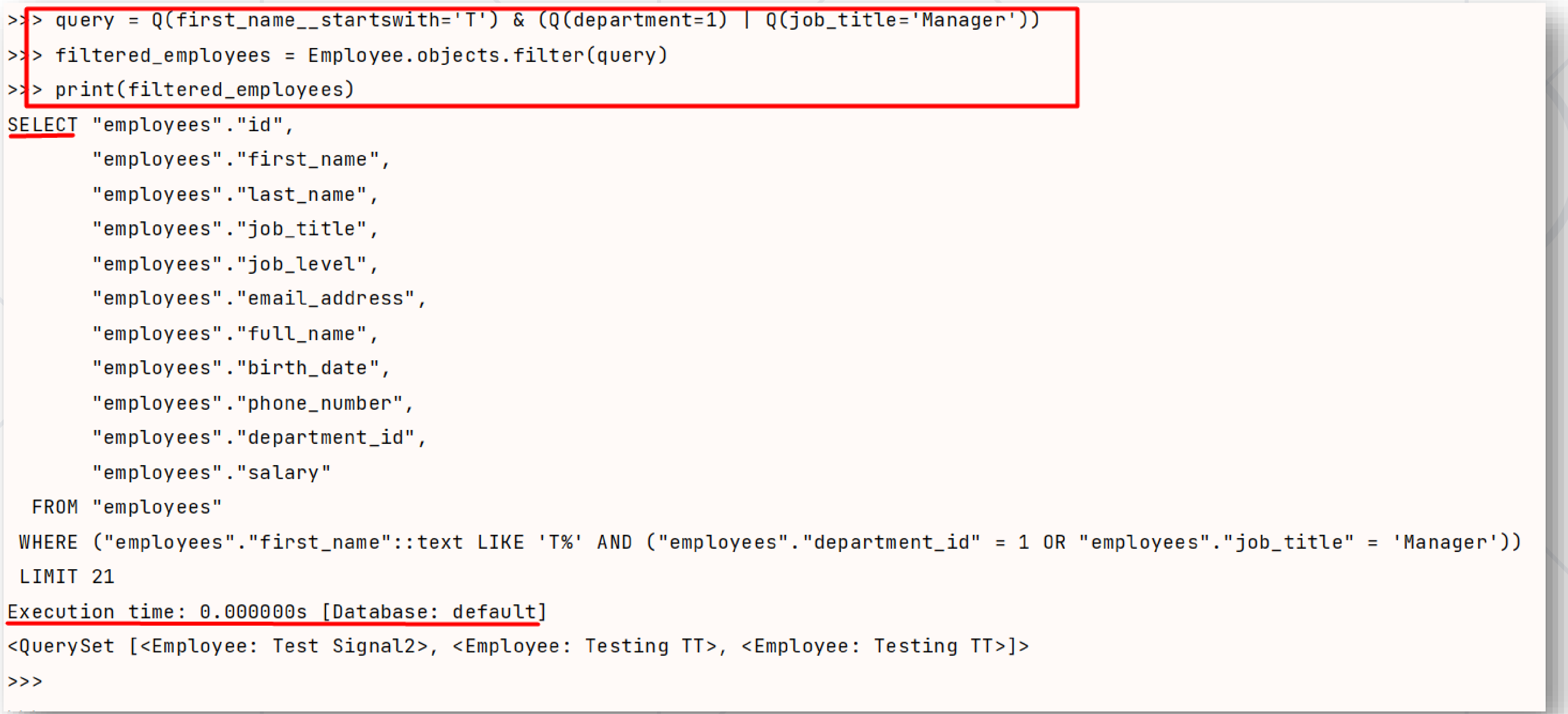
More at: https://django-extensions.readthedocs.io/en/latest/command_extensions.html

Using Shell Plus



```
(venv) PS C:\Users\User\Downloads\                                                             > python manage.py shell_plus --print-sql
# Shell Plus Model Imports
from django.contrib.admin.models import LogEntry
from django.contrib.auth.models import Group, Permission, User
from django.contrib.contenttypes.models import ContentType
from django.contrib.sessions.models import Session
from main_app.models import Department, Employee, Project
# Shell Plus Django Imports
from django.core.cache import cache
from django.conf import settings
from django.contrib.auth import get_user_model
from django.db import transaction
from django.db.models import Avg, Case, Count, F, Max, Min, Prefetch, Q, Sum, When
from django.utils import timezone
```

Using Shell Plus (2)



```
>>> query = Q(first_name__startswith='T') & (Q(department=1) | Q(job_title='Manager'))
>>> filtered_employees = Employee.objects.filter(query)
>>> print(filtered_employees)
SELECT "employees"."id",
       "employees"."first_name",
       "employees"."last_name",
       "employees"."job_title",
       "employees"."job_level",
       "employees"."email_address",
       "employees"."full_name",
       "employees"."birth_date",
       "employees"."phone_number",
       "employees"."department_id",
       "employees"."salary"
  FROM "employees"
 WHERE ("employees"."first_name"::text LIKE 'T%' AND ("employees"."department_id" = 1 OR "employees"."job_title" = 'Manager'))
 LIMIT 21
Execution time: 0.000000s [Database: default]
<QuerySet [<Employee: Test Signal2>, <Employee: Testing TT>, <Employee: Testing TT>]>
>>>
```

Live Demo

Live Exercises in Class

Summary

- Custom Managers
- Annotation
- Queries for Model Relationships
  - `select_related`, `prefetch_related`
- Query-related Tools
  - **Q** and **F** objects
- Debugging Queries

Questions?

Software University

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

Flutter™ International

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

BOSCH

Postbank
Решения за твоето утре

PHAR VISION

SmartIT

DXC TECHNOLOGY

createX

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License