1.Custom Managers:
- In Django, a manager is an interface through which DB query operations are performed
- By default, Django provide 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 are useful when you want to add:
    - custom methods and filters
    - to retreive data from the DB
- 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

2.Annotation:
- Annotation in Django is a powerful feature that allows you to add calculated fileds to your query results

2.1. annotate() method:
- is used to add calculated fields to the queryset
- can be useful when you need to perform aggregation or add derived values to your model instances

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

```
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']}")
```

3.Queries for Model Relationships:
- Specific methods are used to optimize DB queries:
    - when dealing with related objects in your models
    - helping to reduce the number of queries executed
    - improving performance

3.1. select_related():
- used to optimize queries involving FK and OneToOneField relationships
- it fetches related objects in the same query rather than executing a separate query for each related object
- significantly reduces the number of DB querie and improves performance

3.2. prefetch_related():
- used for optimizing queries involving ManyToManyField, reverse FK, 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 queries

4.Query-related Tools

4.1. Q object: a powerful tool that:

- allows you to build complex queries by combining multiple conditions using logical operators

It is especially used when you need:

- to create dynamic queries
- with various conditions
- combined in a flexible way

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(~), OR(^)

You can create instances of the Q object with conditions - use them to construct more complex queries

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

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}')
```

4.2. F object: a tool that allows you to reference a field's value in a query expression

- It is useful for performing operations:
  - involving the values of fields
  - within the DB query itself
  - does not fetch the values
  - does not perform the operations in Python code

You can compare and manipulate field values directly in the DB query:

- comparing the values of two fields
- updating fields with other fields' values
- leads to more efficient and optimized queries

5.Debugging Queries:

- Django Debug Toolbar
- Silk
- Django-queryconstruct
- Django-extensions
  ```
  pip install django-extensions
  ```
- Shell Plus
  ```
  python manage.py shellplus
  python manage.py shell_plus --print-sql
  ```