|  |  |  |
| --- | --- | --- |
| Python Logical Operators | | |
| Symbol | Operator Name | Description |
| or | Logical OR | The condition is true if any of the two operands are non-zero. |
| and | Logical AND | If both the operands are true, then the condition is true. |
| not | Logical NOT | It is used to reverse the logical state of its operand. |

**Identity Operators**

For comparing the memory locations of two objects, identity operators are used.

**DJANGO COMMAND**

All the posts that contain the word 'title' in the title field?

>>> Post.objects.filter(title\_\_contains='title')

There are two underscore characters (\_) between title and contains. Django's ORM uses this rule to separate field names ("title") and operations or filters ("contains"). If you use only one underscore, you'll get an error like "FieldError: Cannot resolve keyword title\_contains".

Retrieving Objects:

Retrieve all objects:

all\_books = Book.objects.all()

Retrieve a single object (returns MultipleObjectsReturned if multiple objects are found)/ returns None if no objects are found:

first\_book = Book.objects.first()

Filtering Objects:

Retrieve objects based on a condition:

fiction\_books = Book.objects.filter(category='Fiction')

Retrieve objects using multiple conditions:

expensive\_books = Book.objects.filter(price\_\_gt=50, category='Fiction')

Update:

MyModel.objects.filter(field1=value1).update(field2=new\_value)

Delete

MyModel.objects.filter(field1=value1).delete()

Exclude objects based on a condition:

non\_fiction\_books = Book.objects.exclude(category='Fiction')

Querying by Field Lookups:

Case-insensitive match:

books = Book.objects.filter(title\_\_iexact='harry potter')

Partial match:

books = Book.objects.filter(title\_\_contains='python')

Date-based filtering:

recent\_books = Book.objects.filter(pub\_date\_\_year=2022)

Ordering Objects:

Order by a field in ascending order:

sorted\_books = Book.objects.order\_by('title')

Order by a field in descending order:

sorted\_books = Book.objects.order\_by('-price')

Aggregation:

Count the number of objects:

book\_count = Book.objects.count()

Calculate the average price:

avg\_price = Book.objects.aggregate(avg\_price=Avg('price'))

Chaining Querysets:

Chain multiple querysets:

filtered\_books = Book.objects.filter(category='Fiction').order\_by('-price')

Additional Operations:

Retrieve distinct values:

distinct\_categories = Book.objects.values\_list('category', flat=True).distinct()

`\*args` and `\*\*kwargs` are special syntax in Python used to pass a variable number of arguments to a function.

### \*args (Positional Arguments):

- `\*args` allows a function to accept an arbitrary number of positional arguments.

- It collects these arguments into a tuple within the function.

Example:

```python

def my\_function(\*args):

    for arg in args:

        print(arg)

my\_function('apple', 'banana', 'orange')

```

Output:

```

apple

banana

orange

```

### \*\*kwargs (Keyword Arguments):

- `\*\*kwargs` allows a function to accept an arbitrary number of keyword arguments.

- It collects these arguments into a dictionary within the function, where the keys are the argument names and the values are the corresponding values.

Example:

```python

def my\_function(\*\*kwargs):

    for key, value in kwargs.items():

        print(f"{key}: {value}")

my\_function(name='John', age=30, city='New York')

```

Output:

```

name: John

age: 30

city: New York

```

### Combining \*args and \*\*kwargs:

You can use both `\*args` and `\*\*kwargs` in the same function definition. In this case, `\*args` collects positional arguments into a tuple, and `\*\*kwargs` collects keyword arguments into a dictionary.

Example:

```python

def my\_function(\*args, \*\*kwargs):

    print("Positional arguments:")

    for arg in args:

        print(arg)

    print("\nKeyword arguments:")

    for key, value in kwargs.items():

        print(f"{key}: {value}")

my\_function('apple', 'banana', 'orange', name='John', age=30, city='New York')

```

Output:

```

Positional arguments:

apple

banana

orange

Keyword arguments:

name: John

age: 30

city: New York

```

Using `\*args` and `\*\*kwargs` allows for flexibility in function definitions, enabling functions to accept a wide range of input parameters.

**Q obj.**

from django.db.models import Q

from myapp.models import MyModel

# Example: Retrieve objects where either field1 is True or field2 is True

query\_condition = Q(field1=True) | Q(field2=True)

queryset = MyModel.objects.filter(query\_condition)

Certainly! Here are some business logic scenarios along with Django queries that you might encounter in a Django interview:

1. \*\*Scenario\*\*: Retrieve all active users who have made a purchase in the last month.

     from django.utils import timezone

     from myapp.models import User, Purchase

     active\_users = User.objects.filter(is\_active=True, purchase\_\_timestamp\_\_gte=timezone.now() - timezone.timedelta(days=30)).distinct()

     ```

2. \*\*Scenario\*\*: Calculate the total revenue generated from purchases made by users in a specific city.

**user\_\_profile\_\_city=city**: Uses double underscores (**\_\_**) to follow relationships across models. This condition means that the query will only include purchases where the associated user's profile's city is **"New York"**

from myapp.models import Purchase

 city = "New York"  # Example city

     total\_revenue = Purchase.objects.filter(user\_\_profile\_\_city=city).aggregate(total\_revenue=models.Sum('amount'))['total\_revenue']

# if only Purchase model is avl.

city = "New York"

total\_revenue = Purchase.objects.filter(city=city).aggregate(total\_revenue=models.Sum('amount'))['total\_revenue']

     ```

3. \*\*Scenario\*\*: Retrieve the top 5 users with the highest total purchase amount.

   - \*\*Django Query\*\*:

     ```python

     from myapp.models import User, Purchase

     top\_users = User.objects.annotate(total\_purchase=models.Sum('purchase\_\_amount')).order\_by('-total\_purchase')[:5]

     ```

4. \*\*Scenario\*\*: Identify users who have not made any purchases in the last 6 months.

   - \*\*Django Query\*\*:

     ```python

     from django.utils import timezone

     from myapp.models import User

     inactive\_users = User.objects.exclude(purchase\_\_timestamp\_\_gte=timezone.now() - timezone.timedelta(days=180))

     ```

5. \*\*Scenario\*\*: Calculate the average order value for purchases made in a specific product category.

   - \*\*Django Query\*\*:

     ```python

     from myapp.models import Purchase, Product

     category = "Electronics"  # Example category

     avg\_order\_value = Purchase.objects.filter(product\_\_category=category).aggregate(avg\_order\_value=models.Avg('amount'))['avg\_order\_value']

     ```

6. \*\*Scenario\*\*: Retrieve all products with low inventory (less than 10 items in stock).

   - \*\*Django Query\*\*:

     ```python

     from myapp.models import Product

     low\_inventory\_products = Product.objects.filter(stock\_\_lt=10)

     ```

7. \*\*Scenario\*\*: Identify users who have made purchases worth more than $1000 in the last quarter.

   - \*\*Django Query\*\*:

     ```python

     from django.utils import timezone

     from myapp.models import User, Purchase

     high\_value\_users = User.objects.filter(purchase\_\_timestamp\_\_gte=timezone.now() - timezone.timedelta(days=90), purchase\_\_amount\_\_gt=1000).distinct()

     ```

These scenarios cover various business logic requirements commonly encountered in Django applications, along with the corresponding Django queries to fulfill those requirements.

**Filtering Objects:**

## **Field Lookups Reference**

A list of all field look up keywords:

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| [contains](https://www.w3schools.com/django/ref_lookups_contains.php) | Contains the phrase |
| [icontains](https://www.w3schools.com/django/ref_lookups_icontains.php) | Same as contains, but case-insensitive |
| date | Matches a date |
| day | Matches a date (day of month, 1-31) (for dates) |
| [endswith](https://www.w3schools.com/django/ref_lookups_endswith.php) | Ends with |
| [iendswith](https://www.w3schools.com/django/ref_lookups_iendswith.php) | Same as endswidth, but case-insensitive |
| [exact](https://www.w3schools.com/django/ref_lookups_exact.php) | An exact match |
| [iexact](https://www.w3schools.com/django/ref_lookups_iexact.php) | Same as exact, but case-insensitive |
| [in](https://www.w3schools.com/django/ref_lookups_in.php) | Matches one of the values |
| isnull | Matches NULL values |
| [gt](https://www.w3schools.com/django/ref_lookups_gt.php) | Greater than |
| [gte](https://www.w3schools.com/django/ref_lookups_gte.php) | Greater than, or equal to |
| hour | Matches an hour (for datetimes) |
| [lt](https://www.w3schools.com/django/ref_lookups_lt.php) | Less than |
| [lte](https://www.w3schools.com/django/ref_lookups_lte.php) | Less than, or equal to |
| minute | Matches a minute (for datetimes) |
| month | Matches a month (for dates) |
| quarter | Matches a quarter of the year (1-4) (for dates) |
| [range](https://www.w3schools.com/django/ref_lookups_range.php) | Match between |
| regex | Matches a regular expression |
| iregex | Same as regex, but case-insensitive |
| second | Matches a second (for datetimes) |
| [startswith](https://www.w3schools.com/django/ref_lookups_startswith.php) | Starts with |
| [istartswith](https://www.w3schools.com/django/ref_lookups_istartswith.php) | Same as startswith, but case-insensitive |
| time | Matches a time (for datetimes) |
| week | Matches a week number (1-53) (for dates) |
| week\_day | Matches a day of week (1-7) 1 is sunday |
| iso\_week\_day | Matches a ISO 8601 day of week (1-7) 1 is monday |
| year | Matches a year (for dates) |
| iso\_year | Matches an ISO 8601 year (for dates) |

1. **<model\_name>.objects.all()**: View all items in a model.
2. **<model\_name>.objects.get()**: Retrieve a single item from a model based on specified criteria.
3. **<model\_name>.objects.filter()**: Filter items in a model based on specified criteria.
4. **<model\_name>.objects.create()**: Create a new item in a model.
5. **<model\_instance>.save()**: Save changes to a model instance.
6. **<model\_instance>.delete()**: Delete a model instance from the database.

New Django ORM Scenarios and Queries:

Scenario: Retrieve all products with a price greater than $50 and sort them by price in descending order.

products = Product.objects.filter(price\_\_gt=50).order\_by('-price')

Scenario: Get the total number of active users.

python

Copy code

active\_users\_count = User.objects.filter(is\_active=True).count()

Scenario: Retrieve all purchases made by users from a specific city (e.g., "San Francisco").

python

Copy code

city = "San Francisco"

purchases = Purchase.objects.filter(user\_\_profile\_\_city=city)

Scenario: Get the average rating of a product.

python

Copy code

product\_id = 1  # Example product ID

average\_rating = Review.objects.filter(product\_id=product\_id).aggregate(avg\_rating=models.Avg('rating'))['avg\_rating']

Scenario: Retrieve the latest 10 orders placed by a specific user.

python

Copy code

user\_id = 1  # Example user ID

latest\_orders = Order.objects.filter(user\_id=user\_id).order\_by('-timestamp')[:10]

Scenario: Find all books published in the current year.

python

Copy code

from django.utils import timezone

current\_year = timezone.now().year

books = Book.objects.filter(publication\_date\_\_year=current\_year)

Scenario: Calculate the total sales for a given product.

python

Copy code

product\_id = 1  # Example product ID

total\_sales = Sale.objects.filter(product\_id=product\_id).aggregate(total\_sales=models.Sum('amount'))['total\_sales']

Scenario: Retrieve all employees who joined in the last month.

python

Copy code

from django.utils import timezone

import datetime

last\_month = timezone.now() - datetime.timedelta(days=30)

recent\_employees = Employee.objects.filter(joining\_date\_\_gte=last\_month)

Scenario: Get the list of products that have not been sold yet.

python

Copy code

unsold\_products = Product.objects.filter(sales\_\_isnull=True)

Scenario: Fetch all orders along with their related products and customers using select\_related.

python

Copy code

orders = Order.objects.select\_related('product', 'customer').all()

**New Django ORM Scenarios and Queries:**

1) Scenario: Retrieve all products with a price greater than $50 and sort them by price in descending order.

products = Product.objects.filter(price\_\_gt=50).order\_by('-price')

2) Scenario: Get the total number of active users.

active\_users\_count = User.objects.filter(is\_active=True).count()

3) Scenario: Retrieve all purchases made by users from a specific city (e.g., "San Francisco").

city = "San Francisco"

purchases = Purchase.objects.filter(user\_\_profile\_\_city=city)

4) Scenario: Get the average rating of a product.

product\_id = 1 # Example product ID

average\_rating = Review.objects.filter(product\_id=product\_id).aggregate(avg\_rating=models.Avg('rating'))['avg\_rating']

5) Scenario: Retrieve the latest 10 orders placed by a specific user.

python

Copy code

user\_id = 1 # Example user ID

latest\_orders = Order.objects.filter(user\_id=user\_id).order\_by('-timestamp')[:10]

Scenario: Find all books published in the current year.

python

Copy code

from django.utils import timezone

current\_year = timezone.now().year

books = Book.objects.filter(publication\_date\_\_year=current\_year)

Scenario: Calculate the total sales for a given product.

python

Copy code

product\_id = 1 # Example product ID

total\_sales = Sale.objects.filter(product\_id=product\_id).aggregate(total\_sales=models.Sum('amount'))['total\_sales']

Scenario: Retrieve all employees who joined in the last month.

python

Copy code

from django.utils import timezone

import datetime

last\_month = timezone.now() - datetime.timedelta(days=30)

recent\_employees = Employee.objects.filter(joining\_date\_\_gte=last\_month)

Scenario: Get the list of products that have not been sold yet.

python

Copy code

unsold\_products = Product.objects.filter(sales\_\_isnull=True)

Scenario: Fetch all orders along with their related products and customers using select\_related.

python

Copy code

orders = Order.objects.select\_related('product', 'customer').all()

DJ CRUD

class StatusViewSet(viewsets.ModelViewSet):

queryset = Status.objects.all()

    serializer\_class = StatusSerializer

    def get\_queryset(self):

        return Status.objects.all()

    def list(self, request, \*args, \*\*kwargs):

        try:

            #search\_query = request.query\_params.get('search', '')

            queryset = self.get\_queryset()

            serializer = self.get\_serializer(queryset, many=True)

            return Response({"success":True, "message":"List retrieved successfully", "data":serializer.data}, status=status.HTTP\_200\_OK)

        except Exception as e:

            return Response({"success":False, "message":"Something went wrong. Please try again later.", "data":None, "errors":str(e)}, status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

class StatusViewSet(viewsets.ModelViewSet):

* This line defines a Django viewset class named **StatusViewSet**.
* **viewsets.ModelViewSet** is a Django REST Framework class that provides CRUD (Create, Retrieve, Update, Delete) operations for a model. It is a convenience class that combines several mixins and simplifies the creation of CRUD views.
* queryset = Status.objects.all()
* This line sets the initial queryset for the viewset to retrieve all objects of the **Status** model.
* It retrieves all instances of the **Status** model from the database using **Status.objects.all()**.
* serializer\_class = StatusSerializer
* This line specifies the serializer class to be used for serializing and deserializing instances of the **Status** model.
* It associates the **StatusSerializer** class with the **StatusViewSet**, indicating that instances of the **Status** model should be serialized/deserialized using the **StatusSerializer**.
* def get\_queryset(self):
  + This line defines a method named **get\_queryset()** within the **StatusViewSet** class.
  + It overrides the default **get\_queryset()** method provided by **ModelViewSet** to customize the queryset used by the viewset.

1. return Status.objects.all()
   * This line returns the queryset containing all instances of the **Status** model.
   * It ensures that whenever the **get\_queryset()** method is called, it returns all instances of the **Status** model, regardless of any additional filtering.
2. def list(self, request, \*args, \*\*kwargs):
   * This line defines a method named **list()** within the **StatusViewSet** class.
   * It overrides the default **list()** method provided by **ModelViewSet** to customize the behavior of listing objects.
3. queryset = self.get\_queryset()
   * This line retrieves the queryset containing all instances of the **Status** model using the **get\_queryset()** method defined earlier.
   * It ensures that the queryset used for listing objects is consistent with the queryset used for other operations.
4. serializer = self.get\_serializer(queryset, many=True)
   * This line initializes a serializer instance to serialize the queryset obtained in the previous step.
   * It uses the **get\_serializer()** method provided by **ModelViewSet** to create a serializer instance for the queryset, indicating that it contains multiple instances (**many=True**).
5. return Response({"success":True, "message":"List retrieved successfully", "data":serializer.data}, status=status.HTTP\_200\_OK)
   * This line returns a success response containing the serialized data obtained from the queryset.
   * It includes a success message, the serialized data, and a status code indicating success (HTTP 200 OK).
6. except Exception as e:
   * This line starts an exception handling block to catch any exceptions that occur during the execution of the **list()** method.
7. return Response({"success":False, "message":"Something went wrong. Please try again later.", "data":None, "errors":str(e)}, status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)
   * This line returns an error response if an exception occurs during the execution of the **list()** method.
   * It includes an error message, details of the exception (**str(e)**), and a status code indicating an internal server error (HTTP 500 Internal Server Error).

Pandas ( Wipro Question)

# Manipulation on string by .translate() / .maketrans()

str.maketrans('@#', ' ') creates a translation table that maps @, #, and', ' ') creates a translation table that maps @, #, and to spaces.

str.translate(my\_table) applies this translation table to replace these special characters with spaces in the string.

str = 'Poulomi'

mytable = str.maketrans('Po','tf','mi') # x replaced by y, z will remove

print(str.translate(mytable))

o/p : tfulf

str = 'soumik das 12345 @#$'

my\_table = str.maketrans(' ' , ' ','@#$')

str.translate(my\_table)

o/p: 'soumik das 12345 '