Advanced Django ORM concepts.

Assume we have the following model:
Python

```python
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    cost = models.DecimalField(max_digits=10, decimal_places=2)
    stock = models.IntegerField()
    category = models.CharField(max_length=50)
```

---

1. Q Objects (Complex Lookups)
Standard Django filtering (`.filter()`) "ANDs" conditions together. `Q` objects allow you to use **OR** and **NOT**logic.
Key Syntax:

- `|` = OR
- `&` = AND
- `~` = NOT

Examples:
Python

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

# OR: Get products that are either "Electronics" OR have a price over 1000
results = Product.objects.filter(Q(category="Electronics") | Q(price__gt=1000))
```

```
# NOT: Get products that are NOT "Toys"
results = Product.objects.filter(~Q(category="Toys"))


# Complex: "Electronics" that are either cheap (<50) OR out of stock (0)
results = Product.objects.filter(
    Q(category="Electronics") & (Q(price__lt=50) | Q(stock=0))
)
```

---

2. F Objects (Field References)
`F` objects allow you to reference the value of a model field *within the database query itself*, without loading it into Python memory. This is crucial for comparing two fields on the same model or performing atomic updates.
Use Case A: Comparisons
Filter based on the relationship between two fields in the same row.

Python
```
from django.db.models import F


# Find products where the Sale Price is less than the Cost (Selling at a loss)
losses = Product.objects.filter(price__lt=F('cost'))
```

Use Case B: Atomic Updates
Update a field based on its current value. This prevents race conditions.

Python
```
# Increase the stock of ALL products by 10
# The database does the math, not Python
Product.objects.update(stock=F('stock') + 10)
```

---

3. Aggregate (Summary Statistics)
`aggregate()` performs a calculation over the entire queryset and returns a single **dictionary** of values. It does *not* return a QuerySet.
Common Functions: `Sum`, `Avg`, `Max`, `Min`, `Count`.

Example:
Python
```python
from django.db.models import Avg, Sum

# Get the average price and total stock of all products
stats = Product.objects.aggregate(
    average_price=Avg('price'),
    total_inventory=Sum('stock')
)

# Output is a dictionary:
# {'average_price': 150.50, 'total_inventory': 5000}
print(stats['average_price'])
```

---

4. Annotation (Per-Object Calculation)
`annotate()` is similar to `aggregate`, but instead of collapsing the data into a dictionary, it calculates a value for **each item** in the QuerySet and attaches it as a temporary attribute. It returns a **QuerySet**.

Example:
Python
```python
from django.db.models import F

# Add a 'profit' attribute to every product object in the resu
lts
products = Product.objects.annotate(
    profit_margin=F('price') - F('cost')
)


for p in products:
    # We can now access 'profit_margin' as if it were a field
on the model
    print(f"{p.name} makes ${p.profit_margin}")
```

Summary Table

Feature	**Returns**	**Primary Purpose**
Q	QuerySet	Complex logic (OR, NOT) in filters.
F	Query Expression	Referencing field values in the DB (Comparisons/Updates).
Aggregate	Dictionary	Calculating statistics across the *whole* table.
Annotate	QuerySet	Calculating a new field for *each* row.