```
from django.shortcuts import render
from django.contrib.contenttypes.models import ContentType
from django.db import connection
from django.db import transaction
from django.db.models import Q, F, Value, Func, ExpressionWrapper, DecimalField
from django.db.models.functions import Concat
from django.db.models.aggregates import Count, Max, Min, Avg, Sum
from store.models import OrderItem, Product, Order, Customer, Collection
from tags.models import TaggedItem


def say_hello(request):
    # ########################################################################
    # 01- Managers and QuerySets

    Product.objects  # Returns a manager object: Interface to database
    query_set = Product.objects.all()  # all() Returns a query_set
    # and query_sets are lazy so at some point in time evaluation of this query_set will be done
    Product.objects.count()  # Not return a query_set , just return a number
    # Evaluate:
    # 1) Iterate
    for product in query_set:
        print(product.title)
    # 2) List
    print(list(query_set))
    # 3) Access individual element
    print(query_set[0])
    # 4) Slice
    print(query_set[0:5])

    # ########################################################################
    # 02- Retrieving Objects

    # 1) all(): all items, returns query_set
    Product.objects.all()
    # 2) get(): single item, returns object, throws an exception
    Product.objects.get(id=1)
    # 3) filter(): filter items, returns query_set, don't throws and exception just returns None
    Product.objects.filter(id=1)
    # 4) first(): Returns an object
    Product.objects.filter(id=1).first()
    # 5) exists(): Returns a boolean value
    Product.objects.filter(id=1).exists()
    # ########################################################################
    # 03- Filtering Objects
```

```python
# 1) == 20
Product.objects.filter(unit_price=20)
# 2) >= 20
Product.objects.filter(unit_price__gte=20)
# 3) range
Product.objects.filter(unit_price__range=(10, 20))
# 4) attribute of a column(id of collection)
Product.objects.filter(collection__id__range=(1, 2, 3))
# 5) filter for string
Product.objects.filter(title__icontains='coffee')
Product.objects.filter(title__istartswith='coffee')
Product.objects.filter(title__iendswith='coffee')
# 6) filter for date
Product.objects.filter(last_update__day=10)
Product.objects.filter(last_update__year=2010)
Product.objects.filter(last_update__month=1)
Product.objects.filter(last_update__hour=12)
Product.objects.filter(last_update__minute=30)
Product.objects.filter(last_update__second__gt=15)
# 7) checking for null
Product.objects.filter(description__isnull=True)
# ########################################################################
# 04- Complex Lookups Using Q Objects

# 1) AND: inventory < 10 AND price < 20
Product.objects.filter(inventory__lt=10, unit_price_lt=20)
# or
Product.objects.filter(inventory__lt=10).filter(unit_price_lt=20)
# or
Product.objects.filter(Q(inventory__lt=10) & Q(unit_price__lt=20))
# 2) OR: inventory < 10 OR price < 20
Product.objects.filter(Q(inventory__lt=10) | Q(unit_price__lt=20))
# 3) OR: inventory < 10 OR NOT(price < 20)
Product.objects.filter(Q(inventory__lt=10) | ~Q(unit_price__lt=20))
# ########################################################################
# 05- Referencing Fields using F Objects

Product.objects.filter(unit_price=F('inventory'))
# ########################################################################
# 06- Sorting

# 1) Ascending
Product.objects.order_by('title')
# 2) Descending
Product.objects.order_by('-title')
# 3) order by multiple columns
```

```python
    Product.objects.order_by('-title', 'unit_price')
    # 4) reverse()
    Product.objects.order_by('-title').reverse()  # ASC
    # 5) earliest
    Product.objects.filter(collection__id=1).order_by('title')[0]  # eager
    Product.objects.filter(collection__id=1).order_by('title').earliest()  # lazy
    # 6) latest
    Product.objects.filter(collection__id=1).order_by('title').latest()
    # ######################################################################
    # 07- Limiting Results

    # Limit
    Product.objects.all()[0:5]
    # Limit and OFFSET
    Product.objects.all()[5:10]
    # ######################################################################
    # 08-  Selecting Fields To Query

    # 1) values(): just the fields we need, returns a dictionary
    Product.objects.values('id', 'title')
    # 2) values_list(): just the fields we need, returns a tuple
    Product.objects.values_list('id', 'title')

    # Exercise:
    # distinct(): removes the duplicates
    Product.objects.filter(id__in=OrderItem.objects.values('product_id').distinct()).order_by('title')

    # ######################################################################
    # 09-  Deferring Fields

    # only(): will get instance of the product class
    # values(): will get dictionary objects
    # 1) only()
    Product.objects.only('id', 'title')

    # in hello.html => product.title - $ product.price

    # WARNING: If we use other fields of product, our application will freeze
    # because we should use a query for all rows(for each product we should have a query to get price)
    # So a lot of overhead
    # WARNING: We don't have the same issue with the values()
    # because values() returns dictionary and these dictionary objects don't have this behavior
    # So if we access a field that doesn't exist in the dictionary that dictionary is not issue a query to the
database

    # 2) defer()
```

```
Product.objects.defer('description')
# ##########################################################################
# 10-  Selecting Related Objects

# preload a bunch of objects together
# select_related: 1
# prefetch_related: n
# 1) select_related()
Product.objects.select_related('collection').all()
# 2) prefetch_related()
Product.objects.prefetch_related('promotions').all()

# Exercise:
Order.objects.select_related('customer').prefetch_related('orderitem_set__product').order_by('-
placed_at')[0:5]
# ##########################################################################
# 11- Aggregating Objects

# 1) Count()
Product.objects.aggregate(count=Count('id'))
# 2) Min()
Product.objects.aggregate(min=Min('unit_price'))
# 3) Max()
Product.objects.aggregate(max=Max('unit_price'))
# 4) Avg()
Product.objects.aggregate(avg=Avg('unit_price'))
# 5) Sum()
Product.objects.aggregate(sum=Sum('unit_price'))
# ##########################################################################
# 12-  Annotating Objects

Customer.objects.annotate(is_new=Value(True))
Customer.objects.annotate(new_id=F('id') + 1)

# ##########################################################################
# 13- Calling Database Functions

Customer.objects.annotate(
    full_name=Func(F('first_name'), Value(' '), F('last_name'), function='CONCAT')
)

Customer.objects.annotate(
    full_name=Concat('first_name', Value(' '), 'last_name')
)

# ##########################################################################
```

```
    # 14- Grouping Data

    Customer.objects.annotate(
        orders_count=Count('order')
    )

    # ##########################################################################
    # 15- Working With Expression Wrappers

    # for building complex expressions
    Product.objects.annotate(
        discounted_price=ExpressionWrapper(F('unit_price') * 0.8, output_field=DecimalField())
    )

    # ##########################################################################
    # 16- Querying Generic Relationships

    TaggedItem.objects.select_related('tag') \
        .filter(
        content_type=ContentType.objects.get_for_model(Product),
        object_id=1
    )

    # ##########################################################################
    # 17- Custom Managers

    TaggedItem.objects.get_tags_for(Product, 1)
    # Create custom manager in models.py(TaggedItemManager, TaggedItem)
class TaggedItemManager(models.Manager):
    def get_tags_for(self, obj_type, obj_id):
        return \
            TaggedItem.objects.select_related('tag') \
                .filter(content_type=ContentType.objects.get_for_model(obj_type),
                        object_id=obj_id
                        )

class TaggedItem(models.Model):
    objects = TaggedItemManager()
    tag = models.ForeignKey(Tag, on_delete=models.CASCADE)
    content_type = models.ForeignKey(ContentType, on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey()

    # ##########################################################################
    # 18- Understanding QuerySet Cache
    query_set = Product.objects.all()
    products = list(query_set)  # read from database
    # Reading from disk is so much slower than memory So django will store the products in querySet
cache
    products = list(query_set)  # read from querySet cache

    # WARNING: caching only happens if we evaluate all the items in querySet Example:
    query_set = Customer.objects.all()
    customer = query_set[0]
    customers = list(query_set)

    # ######################################################################
    # 19- Creating Objects

    # 1) Better Approach because of feature changes
    collection = Collection()
    collection.title = 'Video Games'
    collection.featured_product = Product(id=1)  # first way
    collection.featured_product_id = 1  # second way
    collection.save()

    # 2)
    collection = Collection(title='Video Games',
                            featured_product_id=1)
    collection.save()

    # 3) save automatically
    Collection.objects.create(title='Video Games',
                              featured_product_id=1)

    # ######################################################################
    # 20- Updating Objects

    # 1)
    collection = Collection.objects.get(pk=11)
    collection.featured_product = None
    collection.save()

    # 2) for better performance
    Collection.objects.filter(pk=11).update(featured_product=None)

    # ######################################################################
    # 21- Deleting Objects

    # 1)
    collection = Collection(pk=11)
    collection.delete()

    # 2)
    Collection.objects.filter(id__gt=5).delete()

    # ############################################################################
    # 22- Transaction

    # 1) decorator
    @transaction.atomic()
    def create_order_item():
        order = Order()
        order.customer_id = 1
        order.save()

        item = OrderItem()
        item.order = order
        item.quantity = 10
        item.unit_price = 12.5
        item.save()

    # 2) context manager
    with transaction.atomic():
        order = Order()
        order.customer_id = 1
        order.save()

        item = OrderItem()
        item.order = order
        item.quantity = 10
        item.unit_price = 12.5
        item.save()

    # ############################################################################
    # 23- Executing Raw SQL Queries

    # 1) raw()
    Product.objects.raw('SELECT id, title FROM store_product')

    # 2) connection: directly connect to database and bypass model layer(Don't Map to our model
object)
    # first way:
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM store_product')
    cursor.close()
```

```
# second way(better approach):
with connection.cursor as cursor:
    cursor.execute('SELECT * FROM store_product')
    # for calling stored procedure
    cursor.callproc('get_customers', [1, 2, 3])
# #######################################################################
return render(request, 'hello.html', {'name': 'Mosh'})
```