Topic: **Django Signals**

1.

In Django signals are executed synchronously by default , means that when a signal is sent, the associated signal handlers are executed immediately, blocking the flow of code execution until the handlers are done.

Flow of Execution :

- The test function will print "Creating MyModel instance".
- The signal handler will execute as soon as the MyModel instance is created, delaying the next print statement for 5 seconds due to the time.sleep(5) call.
- After the delay, the test function will print "MyModel instance created.".

CODE :

```
from django.db import models

from django.db.models.signals import post_save

from django.dispatch import receiver

import time


class MyModel(models.Model):

    name = models.CharField(max_length=100)


@receiver(post_save, sender=MyModel)

def my_signal_handler(sender, instance, **kwargs):

    print("Signal received, processing started...")

    time.sleep(5)

    print("Signal processing finished.")


def test_signal():

    print("Creating My_Model instance...")

    instance = MyModel.objects.create(name="Test Instance")

    print(" instance created.")
```

O/P ---

Creating MyModel instance...

Signal received, processing started...

[5-second delay]

Signal processing finished.

MyModel instance created.

2.

Yes ,Django signals run in the same thread as the caller , by default means when a signal is triggered, its connected handlers run in the same thread as the code that triggered the signal.

Flow of Execution:

- **test_signal() is called**:
- Prints the thread name for the caller function (likely MainThread).
- Creates and saves a new instance of MyModel, which triggers the post_save signal.
- **post_save Signal is Triggered**:
- The signal is automatically triggered after the instance is saved to the database.
- **my_signal_handler() Executes**:
- The signal handler prints the thread name (likely MainThread).
- It completes the signal processing and prints "Signal processing complete.".

CODE:

```
import threading
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver

class MyModel(models.Model):
    name = models.CharField(max_length=100)
```

```
@receiver(post_save, sender=MyModel)
def my_signal_handler(sender, instance, **kwargs):
    print(f"Signal handler running in thread: {threading.current_thread().name}")
    print("Signal processing complete.")

def test_signal():
    print(f"Caller function running in thread: {threading.current_thread().name}")
    instance = MyModel.objects.create(name="Test Instance")

test_signal()
```

O/P ---

Caller function running in thread: MainThread

Signal handler running in thread: MainThread

Signal processing complete.

3.

Yes, by default, Django signals run in the same database transaction as the caller if the signal is triggered within a database transaction block. This means that if the caller's transaction is rolled back, the signal's operations will also be rolled back, and if the transaction is committed, the signal's operations will also be committed.

Flow of Execution

- The test_signal_with_transaction function will start a database transaction.
- The MyModel instance is created inside the transaction, which triggers the post_save signal.
- The signal handler will attempt to create a SignalLog entry.
- The transaction is intentionally rolled back due to the raised exception.

```
from django.db import models, transaction
from django.db.models.signals import post_save
from django.dispatch import receiver
```

```
class MyModel(models.Model):
    name = models.CharField(max_length=100)

@receiver(post_save, sender=MyModel)
def my_signal_handler(sender, instance, **kwargs):
    print("Signal handler: updating signal_log entry...")
    SignalLog.objects.create(log=f"Signal processed for {instance.name}")
    print("Signal handler: signal_log entry created.")

class SignalLog(models.Model):
    log = models.CharField(max_length=255)

def test_signal_with_transaction():
    try:
        with transaction.atomic():
            print("Caller: creating MyModel instance inside transaction...")
            instance = MyModel.objects.create(name="Test Instance")
            print("Caller: forcing transaction rollback...")
            raise Exception("Forcing transaction rollback to test signal behavior.")
    except Exception as e1:
        print(f"Transaction failed with error: {e1}")

test_signal_with_transaction()

print("SignalLog entries:", SignalLog.objects.all())
```

O/P -----

```
Caller: creating MyModel instance inside transaction...
Signal handler: updating signal_log entry...
Signal handler: signal_log entry created.
Caller: forcing transaction rollback...
```

Transaction failed with error: Forcing transaction rollback to test signal behavior.

SignalLog entries: <QuerySet []> # No entries should exist due to rollback

Topic: Custom Classes in Python

- **__init__ method**: Initializes the Rectangle object with length and width attributes.
- **__iter__ method**: This method is called when you iterate over the object. It yields the length first in the format {'length': <VALUE_OF_LENGTH>} and then yields the width in the format {'width': <VALUE_OF_WIDTH>}.

```
class Rectangle:
    def __init__(self, length: int, width: int):
        self.length = length
        self.width = width

    def __iter__(self):

        yield {'length': self.length}
        yield {'width': self.width}

rect = Rectangle(length=10, width=5)

for item in rect:
    print(item)
```

O/P –

{'length': 10}

{'width': 5}