**Question 1**: By default are django signals executed synchronously or asynchronously? Please support your answer with a code snippet that conclusively proves your stance. The code does not need to be elegant and production ready, we just need to understand your logic.

**Answer:** By default, Django signals are executed synchronously. This means that when a signal is sent, the corresponding signal handler(s) are executed immediately before the flow of the code continues.

code -

from django.db.models.signals import post\_save

from django.dispatch import receiver

from django.contrib.auth.models import User

# Define a signal receiver

@receiver(post\_save, sender=User)

def user\_post\_save\_handler(sender, instance, \*\*kwargs):

print("Signal handler executed")

# Simulate saving a user

def create\_user():

print("Creating user...")

user = User.objects.create(username="test\_user")

print("User created")

create\_user()

o/p-

Creating user...

Signal handler executed

User created

Explain-

-When User.objects.create() is called, the post\_save signal is sent.

-The signal handler is executed immediately and synchronously after the user is saved.

-The "Signal handler executed" message appears **before** "User created," demonstrating that the signal handler runs synchronously before the main code continues.

**Question 2**: Do django signals run in the same thread as the caller? Please support your answer with a code snippet that conclusively proves your stance. The code does not need to be elegant and production ready, we just need to understand your logic.

**Answer:** Yes, Django signals run in the same thread as the caller by default. This means that both the code that sends the signal and the code in the signal handler are executed in the same thread.

Code-

import threading

from django.db.models.signals import post\_save

from django.dispatch import receiver

from django.contrib.auth.models import User

# Signal handler to print the current thread name

@receiver(post\_save, sender=User)

def user\_post\_save\_handler(sender, instance, \*\*kwargs):

print(f"Signal received in thread: {threading.current\_thread().name}")

# Function to create a user and check the thread

def create\_user():

print(f"Caller in thread: {threading.current\_thread().name}")

user = User.objects.create(username="test\_user")

create\_user()

Output:

Caller in thread: MainThread

Signal received in thread: MainThread

Explain:-

-The output shows that both the caller and the signal handler run in the same thread (MainThread).

-If Django signals were running in a different thread, the thread name in the signal handler would differ.

**Question 3**: By default do django signals run in the same database transaction as the caller? Please support your answer with a code snippet that conclusively proves your stance. The code does not need to be elegant and production ready, we just need to understand your logic.

**Answer:** Yes, by default, Django signals run in the same database transaction as the caller. If a transaction is used and subsequently rolled back, any changes made by the signal handler are also rolled back.

Code -

from django.db import transaction

from django.db.models.signals import post\_save

from django.dispatch import receiver

from django.contrib.auth.models import User

# Signal handler

@receiver(post\_save, sender=User)

def user\_post\_save\_handler(sender, instance, \*\*kwargs):

print("Signal handler executed.")

# Function to create a user within a transaction and raise an exception to trigger a rollback

def create\_user():

try:

with transaction.atomic():

print("Transaction started")

user = User.objects.create(username="test\_user")

print("Raising exception to rollback...")

raise Exception("Transaction rollback")

except Exception as e:

print(f"Transaction rolled back: {e}")

create\_user()

Output-

Transaction started

Signal handler executed.

Raising exception to rollback...

Transaction rolled back: Transaction rollback

Explanation:

-The signal handler is executed after the User instance is saved.

-When an exception is raised, the transaction is rolled back, meaning any changes (including those triggered by signals) are also rolled back.

-This demonstrates that Django signals are part of the transaction context.

**Topic: Custom Classes in Python**

**Description:** You are tasked with creating a Rectangle class with the following requirements:

1. An instance of the Rectangle class requires length:int and width:int to be initialized.
2. We can iterate over an instance of the Rectangle class
3. When an instance of the Rectangle class is iterated over, we first get its length in the format: **{'length': <VALUE\_OF\_LENGTH>}** followed by the width **{width: <VALUE\_OF\_WIDTH>}**

**Answer:** We can achieve this by defining a custom \_\_iter\_\_ method within the Rectangle class. This allows the class instance to be iterable and return the specified dictionary format for both the length and the width.

Code-

class Rectangle:

def \_\_init\_\_(self, length: int, width: int):

self.length = length

self.width = width

# Define an iterator that yields length and width in the required format

def \_\_iter\_\_(self):

yield {'length': self.length}

yield {'width': self.width}

# Example usage:

rect = Rectangle(5, 3)

# Iterating over the instance of Rectangle

for dimension in rect:

print(dimension)

Output-

{'length': 5}

{'width': 3}

Explanation:

* The \_\_iter\_\_ method allows us to iterate over the Rectangle instance and retrieve the length and width in the required dictionary format.
* First, the length is returned, followed by the width, meeting the specification.