**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.

**Ans.1** By default, Django signals are executed synchronously. This means that when a signal is sent, the connected receiver functions are executed immediately and in the same thread as the sender. This can be demonstrated with a simple code snippet.

Following is the code Django signal and a receiver function:

1. **Define a Signal and Receiver:**

from django.dispatch import Signal, receiver

# Define a custom signal

my\_signal = Signal()

# Define a receiver function

@receiver(my\_signal)

def my\_receiver(sender, \*\*kwargs):

print("Signal received. Executing receiver function.")

1. **Send the Signal:**

import time

def send\_signal():

print("Sending signal...")

my\_signal.send(sender=None)

print("Signal sent.")

# Measure the time taken to send the signal and execute the receiver

start\_time = time.time()

send\_signal()

end\_time = time.time()

print("Time taken:", end\_time - start\_time)

1. **Output:**

Sending signal...

Signal received. Executing receiver function.

Signal sent.

Time taken: <some small value>

**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.

**Ans 2.** Yes, by default, Django signals run in the same thread as the caller.

**Code:**

import threading

from django.dispatch import Signal, receiver

import time

# Define a custom signal

my\_signal = Signal()

# Define a receiver function

@receiver(my\_signal)

def my\_receiver(sender, \*\*kwargs):

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

time.sleep(2) # Simulate some work

print("Receiver function completed")

def send\_signal():

print(f"Sending signal from thread: {threading.current\_thread().name}")

my\_signal.send(sender=None)

print("Signal sent")

# Run the signal in the main thread

print("Running in main thread")

send\_signal()

# Run the signal in a separate thread

def run\_in\_thread():

print(f"Starting new thread: {threading.current\_thread().name}")

send\_signal()

thread = threading.Thread(target=run\_in\_thread)

thread.start()

thread.join()

**Output:**

Running in main thread

Sending signal from thread: MainThread

Receiver function running in thread: MainThread

Receiver function completed

Signal sent

Starting new thread: Thread-1

Sending signal from thread: Thread-1

Receiver function running in thread: Thread-1

Receiver function completed

Signal sent

This output shows that the receiver function runs in the same thread as the sender, whether it's the main thread or a separate thread.

**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.

**Ans 3.** Yes, by default, Django signals run in the same database transaction as the caller.

**Code:**

from django.db import transaction

from django.dispatch import Signal, receiver

from django.db import models

# Define a simple model

class MyModel(models.Model):

name = models.CharField(max\_length=100)

# Define a custom signal

my\_signal = Signal()

# Define a receiver function

@receiver(my\_signal)

def my\_receiver(sender, \*\*kwargs):

print("Receiver function started")

MyModel.objects.create(name="Created in receiver")

print("Receiver function completed")

def send\_signal\_with\_transaction():

print("Starting transaction")

with transaction.atomic():

print("Creating object in sender")

MyModel.objects.create(name="Created in sender")

print("Sending signal")

my\_signal.send(sender=None)

print("Raising exception to rollback transaction")

raise Exception("Rollback transaction")

# Run the function

try:

send\_signal\_with\_transaction()

except Exception as e:

print(f"Caught exception: {e}")

# Check the database state

print("Objects in database after rollback:")

print(MyModel.objects.all().values())

**Output:**

Starting transaction

Creating object in sender

Sending signal

Receiver function started

Receiver function completed

Raising exception to rollback transaction

Caught exception: Rollback transaction

Objects in database after rollback:

<QuerySet []>

This output shows that:

1. The receiver function is called within the same transaction as the sender.
2. When an exception is raised after sending the signal, causing a transaction rollback, both the object created in the sender and the object created in the receiver are rolled back.
3. After the rollback, no objects remain in the database, proving that the receiver's actions were part of the same transaction.

If the receiver were running in a separate transaction, we would still see the object created by the receiver in the database after the rollback. The fact that we don't see any objects confirms that the receiver runs in the same transaction as the caller.

**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>}**

**Code:**

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}

# Create an instance of Rectangle

rectangle = Rectangle(10, 5)

# Iterate over the instance and print the results

for dimension in rectangle:

print(dimension)