**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 the signal handlers are executed within the same request/response cycle as the signal emitter.

To demonstrate this, let's create a simple Django signal and handler and track the timing:

**Code:**

import time

from django.dispatch import Signal, receiver

# Custom signal

test\_signal = Signal()

# Signal handler

@receiver(test\_signal)

def signal\_handler(sender, \*\*kwargs):

print("Signal handler started")

time.sleep(5) # Simulate a time-consuming task

print("Signal handler finished")

# Simulate signal sending

start\_time = time.time()

test\_signal.send(sender=None)

end\_time = time.time()

print(f"Total time taken: {end\_time - start\_time} seconds")

If the signal is synchronous, the signal handler will block the execution until it completes (in this case, 5 seconds). The output will be:

Signal handler started

Signal handler finished

Total time taken: 5.x seconds

This proves that Django signals are executed synchronously by default.

**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 by default run in the same thread as the caller. To demonstrate this, we can print the thread identifiers from both the main thread and the signal handler:

**Code:**

import threading

from django.dispatch import Signal, receiver

# Custom signal

test\_signal = Signal()

# Signal handler

@receiver(test\_signal)

def signal\_handler(sender, \*\*kwargs):

print(f"Signal handler thread ID: {threading.get\_ident()}")

# Simulate signal sending

print(f"Main thread ID: {threading.get\_ident()}")

test\_signal.send(sender=None)

The output should show that the thread IDs are the same, confirming that the signal handler runs in the same thread as the signal sender:

Main thread ID: 12345678

Signal handler thread ID: 12345678

**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, Django signals that are emitted within a transaction-aware context (e.g., inside a transaction.atomic() block) will run within the same database transaction.

Consider the following example:

from django.db import transaction

from django.dispatch import Signal, receiver

from myapp.models import MyModel

# Custom signal

test\_signal = Signal()

# Signal handler

@receiver(test\_signal)

def signal\_handler(sender, \*\*kwargs):

# Try to create a new instance of MyModel

MyModel.objects.create(name="SignalHandler")

# Simulate signal sending within a transaction

try:

with transaction.atomic():

print("Transaction started")

test\_signal.send(sender=None)

raise Exception("Error to rollback transaction")

except Exception as e:

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

# Check if MyModel object was created

print(MyModel.objects.all())

If signals run in the same transaction, the creation of MyModel in the signal handler will be rolled back along with the rest of the transaction, and no new object will be created. This proves that Django signals run within the same database transaction as the caller.

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>}

**Custom Python Class: Rectangle**

Here’s a Rectangle class that meets the specified requirements:

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}

# Usage

rect = Rectangle(5, 10)

for attr in rect:

print(attr)

**Output:**

{'length': 5}

{'width': 10}

This implementation allows you to iterate over the Rectangle instance and retrieve its length and width in the required format.