Question 1: Are Django signals executed synchronously or asynchronously by default?

Answer:

By default, Django signals are executed **synchronously**. When a signal is emitted, the connected signal handler functions are called immediately, before the control returns to the code after the signal is triggered.

Code:

```
# models.py


import time

from django.db.models.signals import post_save

from django.dispatch import receiver

from django.db import models


class MyModel(models.Model):

    name = models.CharField(max_length=50)


@receiver(post_save, sender=MyModel)

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

    print("Signal handler started.")

    time.sleep(5)  # Simulate a long-running task

    print("Signal handler finished.")


# In Django shell or view

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

print("Instance created.")
```

Expected output:

```
Signal handler started.

Signal handler finished.

Instance created.
```

Explanation: The output shows that the signal handler (my_signal_handler) runs **before** the print("Instance created.") statement, proving that Django signals are executed **synchronously**. The delay caused by time.sleep(5) holds up the execution, showing it is blocking (i.e., synchronous).

Question 2: Do Django signals run in the same thread as the caller?

Answer:

Yes, by default, Django signals run in the **same thread** as the caller. The signal handler function is invoked directly in the same thread where the signal was emitted.

Code

```
# models.py


import threading

from django.db.models.signals import post_save

from django.dispatch import receiver

from django.db import models


class MyModel(models.Model):

    name = models.CharField(max_length=50)


@receiver(post_save, sender=MyModel)

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

    print(f"Signal handler thread: {threading.current_thread().name}")


# In Django shell or view

print(f"Main thread: {threading.current_thread().name}")

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

Expected output:

```
Main thread: MainThread

Signal handler thread: MainThread
```

Explanation: Both the main thread and the signal handler print the same thread name (MainThread), which confirms that Django signals run in the **same thread** as the caller by default.

Question 3: Do Django signals run in the same database transaction as the caller?

Answer:

Yes, by default, Django signals run in the **same database transaction** as the caller. If an exception is raised in the signal handler, it can cause the transaction to fail, and the changes made before the signal was called will not be committed to the database.

Code

```
# models.py


from django.db import transaction

from django.db.models.signals import post_save

from django.dispatch import receiver

from django.db import models


class MyModel(models.Model):

    name = models.CharField(max_length=50)


@receiver(post_save, sender=MyModel)

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

    raise Exception("Error in signal handler")


# In Django shell or view

try:

    with transaction.atomic():
```

```
        instance = MyModel.objects.create(name="Test Instance")
        print("Instance created.")
except Exception as e:
    print(f"Exception caught: {e}")

# Check if the instance was saved in the database
print(MyModel.objects.filter(name="Test Instance").exists())
```

Expected output:

Instance created.

Exception caught: Error in signal handler

False

Explanation: The signal handler raises an exception, which causes the entire transaction to fail. As a result, the new MyModel instance is not saved to the database (False is printed when checking if the instance exists). This proves that Django signals run in the **same database transaction** as the caller by default.

Topic: Custom Classes in Python

A Python implementation of a Rectangle class that meets the specified requirements:

1. The class initializes with length and width.
2. It allows iteration over its attributes, where the first value yielded is a dictionary of the format {'length': <VALUE_OF_LENGTH>}, followed by a dictionary {'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}

# Example usage
rect = Rectangle(10, 5)

# Iterating over the Rectangle instance
for dimension in rect:
    print(dimension)
```

Output

```
{'length': 10}
{'width': 5}
```