Topic: Django Signals

Question 1: By default, are Django signals executed synchronously or asynchronously?

Answer: By default, Django signals are executed synchronously. This means that the signal handler is called immediately when the signal is sent, and the sender will wait for the handler to finish before continuing its execution. To demonstrate this, we can use a code snippet where the signal handler introduces a delay, and observe that the rest of the code execution is blocked until the signal handler completes.

Code Snippet:

```
# signals.py
import time
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.contrib.auth.models import User

@receiver(post_save, sender=User)
def my_signal_handler(sender, instance, **kwargs):
    print("Signal handler started...")
    time.sleep(5)  # Introduce a delay
    print("Signal handler finished...")

# Triggering the signal
def create_user():
    print("Creating a new user...")
    User.objects.create(username='testuser')
    print("User created.")

# If we call create_user() in a view or shell
create_user()

# Output:
# Creating a new user...
# Signal handler started...
# (5-second delay)
# Signal handler finished...
# User created.
```

Explanation: In this example, the output shows that the message "User created." is printed only after the signal handler completes its execution, proving that Django signals are executed synchronously by default.

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. This means that if the signal is sent from the main thread, the signal handler will also execute in the main thread. We can prove this by printing the current thread's name inside the signal handler and the function from which the signal is sent.

Code Snippet:

```
# signals.py
```

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

@receiver(post_save, sender=User)
def my_signal_handler(sender, instance, **kwargs):
    print(f"Signal handler thread: {threading.current_thread().name}")

# Triggering the signal
def create_user():
    print(f"Caller thread: {threading.current_thread().name}")
    User.objects.create(username='testuser')

# If we call create_user() in a view or shell
create_user()

# Output:
# Caller thread: MainThread
# Signal handler thread: MainThread
```

Explanation: The output shows that both the caller function (create_user()) and the signal handler (my_signal_handler) run in the MainThread, confirming that Django signals run in the same thread as the caller by default.

Question 3: By default, do Django signals run in the same database transaction as the caller?

Answer: By default, Django signals can run in the same database transaction as the caller, depending on the signal type. For example, the post_save signal runs after a model's save() method completes, which means it is within the same transaction scope if the save operation is part of a database transaction.

Code Snippet:

```
# signals.py
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

@receiver(post_save, sender=User)
def my_signal_handler(sender, instance, **kwargs):
    print("Signal handler started...")
    print(f"Is transaction active: {transaction.get_connection().in_atomic_block}")

# Triggering the signal
def create_user():
    with transaction.atomic():
        print("Creating a new user inside a transaction...")
        User.objects.create(username='testuser')
```

```
# If we call create_user() in a view or shell
create_user()

# Output:
# Creating a new user inside a transaction...
# Signal handler started...
# Is transaction active: True
```

Explanation: The output shows Is transaction active: True, indicating that the signal handler is running within the same database transaction block as the caller, proving that by default, Django signals can run in the same transaction context.

Topic: Custom Classes in Python

Description: Creating a Rectangle class with specific requirements.
Answer:

Here's how you can implement a Rectangle class in Python that meets the provided requirements:

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

    def __iter__(self):
        # Yield length first, followed by width
        yield {'length': self.length}
        yield {'width': self.width}

# Usage example
rect = Rectangle(5, 10)

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

Explanation:

- The Rectangle class is initialised with length and width as integers.
- The __iter__ method is defined to make the instance iterable. It uses the yield statement to return the length in the format {'length': <VALUE_OF_LENGTH>} first, and then the width in the format {'width': <VALUE_OF_WIDTH>}.
- When iterating over an instance of Rectangle, the __iter__ method is called, allowing us to get the desired output.