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

By default, Django signals are executed **synchronously**. This means that when a signal is triggered, the connected receivers (handlers) are executed one after another, and the sender waits for all handlers to complete before continuing.

Here's a code snippet to demonstrate this:

```
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=100)


@receiver(post_save, sender=MyModel)

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

    print("Signal received for:", instance.name)

    time.sleep(2)  # Simulate a delay

    print("Signal handler finished")


# Creating an instance of MyModel

obj = MyModel.objects.create(name="Test")

print("Object created")
```

When you run this code, you will see that the message "Object created" is printed only after the signal handler finishes its execution, indicating synchronous behavior.

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

Yes, by default, Django signals run in the same thread as the caller. This means that the signal handlers are executed in the same thread that triggered the signal.

Here’s a code snippet to demonstrate this:

```
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=100)


@receiver(post_save, sender=MyModel)

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

    print("Signal received for:", instance.name)

    print("Signal handler thread ID:", threading.get_ident())


# Creating an instance of MyModel

obj = MyModel.objects.create(name="Test")

print("Object created")

print("Main thread ID:", threading.get_ident())
```

When you run this code, you will see that the thread ID printed inside the signal handler is the same as the main thread ID, confirming that they run in the same thread.

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

By default, Django signals do not run in the same database transaction as the caller. Django runs in autocommit mode, meaning each database query is executed and committed immediately.

Here’s a code snippet to demonstrate this:

```
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=100)


@receiver(post_save, sender=MyModel)
def my_signal_handler(sender, instance, **kwargs):
    print("Signal received for:", instance.name)
    print("Signal handler in transaction:", transaction.get_connection().in_atomic_block)


# Creating an instance of MyModel
with transaction.atomic():
    obj = MyModel.objects.create(name="Test")
    print("Object created in transaction:", transaction.get_connection().in_atomic_block)
```

When you run this code, you will see that the signal handler
prints False for transaction.get_connection().in_atomic_block, indicating that it is not running in the same 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>}**

Ans:

Here's a Rectangle class that meets your requirements:

1. The class requires length and width to be initialized.
2. The class is iterable.
3. When iterated over, it yields the length first and then the width in the specified format.

```
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)
print("Rectanlge Dimensions:")
for dimension in rect:
    print(dimension)
```

Explanation:

- The __init__ method initializes the Rectangle instance with length and width.
- The __iter__ method makes the class iterable. It uses the yield statement to return the length first and then the width in the specified format.

When you run the example usage, you should see the following output:

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