Topic: Django Signals

Question 1: By default are django signals executed synchronously or asynchronously? Please support your answer with a code snippet that conclusively proves your stance.

Answer: Yes, by default Django signals are executed synchronously. This means that the signal is handled immediately after it is sent, and the sender will wait for the signal's receiver functions to complete before continuing.

To prove this, here's a code snippet that demonstrates this synchronous behavior.

We'll create a signal and a receiver function, then check how the signal is executed in the flow of the code.

CODE:

```
import time

from django.dispatch import Signal, receiver


# Define a signal

my_signal = Signal()


# Define a receiver function

@receiver(my_signal)

def signal_receiver(sender, **kwargs):

    print("Signal received. Processing...")

    time.sleep(3)  # Simulate a time-consuming process

    print("Signal processing finished.")


# Simulate sending the signal and measuring the time

def send_signal():

    print("Sending signal...")

    my_signal.send(sender=None)  # Send the signal

    print("Signal sent. Continuing execution.")
```

```
if __name__ == "__main__":
    send_signal()
```

OUTPUT:

```
Sending signal...
Signal received. Processing...
(Waits for 3 seconds)
Signal processing finished.
Signal sent. Continuing execution.
```

This clearly demonstrates that Django signals are executed synchronously by default because the program waits for the signal handling to complete before moving on to the next line.

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.

Answer: Yes, Django signals run in the same thread as the caller by default. To prove this, we can check the thread identifiers of both the caller and the signal receiver. If the thread Ids are the same, it confirms that the signal runs in the same thread as the caller.

CODE:

```
import threading
from django.dispatch import Signal, receiver

# Define a signal
my_signal = Signal()

# Define a receiver function
@receiver(my_signal)
def signal_receiver(sender, **kwargs):
```

```
    print("Signal received in thread:", threading.current_thread().name)

# Simulate sending the signal and print the caller's thread
def send_signal():
    print("Signal sent from thread:", threading.current_thread().name)
    my_signal.send(sender=None)  # Send the signal

if __name__ == "__main__":
    send_signal()
```

OUTPUT:

Signal sent from thread: MainThread

Signal received in thread: MainThread

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.

Answer: Yes, by default, Django signals run in the same database transaction as the caller. This means that if the caller is within an atomic transaction, and a signal is triggered, both the caller and the signal handlers will share the same transaction. If the transaction is rolled back, the signal's database operations will also be rolled back.

Code to Prove This:

We can create a signal and a receiver function that performs a database operation. We'll then trigger the signal within a transaction block, cause an error after the signal, and check whether the signal's database operation gets rolled back.

```
from django.db import models, transaction
from django.dispatch import Signal, receiver
```

```
# Define a simple model to test with
class TestModel(models.Model):
    name = models.CharField(max_length=100)

# Define a signal
test_signal = Signal()

# Define a receiver function that creates a record in the database
@receiver(test_signal)
def signal_receiver(sender, **kwargs):
    print("Signal received. Inserting record into the database.")
    TestModel.objects.create(name="Signal Record")

# Function to trigger the signal inside a transaction
def test_transaction():
    try:
        with transaction.atomic():
            print("Starting transaction. Sending signal...")
            test_signal.send(sender=None)  # Send the signal
            print("Signal sent. Raising exception to rollback transaction.")
            raise Exception("Simulating an error")
    except Exception as e:
        print(f"Exception caught: {e}")

    # Check if the signal's record was committed
    if TestModel.objects.filter(name="Signal Record").exists():
        print("Signal's record exists. Transaction was NOT rolled back.")
    else:
```

```
    print("Signal's record does NOT exist. Transaction was rolled back.")

if __name__ == "__main__":
    test_transaction()
```

OUTPUT:

Starting transaction. Sending signal...

Signal received. Inserting record into the database.

Signal sent. Raising exception to rollback transaction.

Exception caught: Simulating an error

Signal's record does NOT exist. Transaction was rolled back.

Explaination:

Transaction Block (transaction.atomic()): The code runs within a transaction.

Signal Triggered: Inside the transaction block, the signal is triggered, and the signal handler inserts a record into the database.

Exception Raised: After the signal is sent, we deliberately raise an exception to cause the transaction to roll back.

Check Database State: After the exception, we check if the record inserted by the signal handler exists in the database.

The output shows that the record inserted by the signal handler does not exist after the exception.

This proves that the signal's database operation was rolled back along with the caller's transaction.

Topic: Custom Classes in Python

Description: You are tasked with creating a Rectangle class with the following requirements:

An instance of the Rectangle class requires length:int and width:int to be initialized.

We can iterate over an instance of the Rectangle class

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}

# Example usage:
rect = Rectangle(10, 5)
for dim in rect:
    print(dim)
```

OUTPUT:

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