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. The code does not need to be elegant and production ready, we just need to understand your logic.**

So, by default, Django signals are synchronous, meaning that when you trigger a signal, Django will wait for the signal handler to finish whatever it's doing before moving on to the next step in your code.

Imagine it like this: you're making dinner and someone asks you to pass the salt. You stop what you're doing, hand them the salt, then go back to cooking. That little pause in cooking is similar to how Django handles signals it waits for the signal to finish before continuing.

example to prove that Django signals aren't asynchronous by default:

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

class MyModel(models.Model):
    name = models.CharField(max_length=100)

# Signal handler
receiver(post_save, sender=MyModel)
def my_signal_handler(sender, instance, **kwargs):
    print("Signal received! Let's simulate a delay...")
    time.sleep(5)  # This sleep simulates something taking
time
    print("Done with signal after 5 seconds.")
```

Running this in a view or the Django shell:

```
from myapp.models import MyModel

print("About to create the object...")
MyModel.objects.create(name="Testing")
print("Object created!")
```

Output:

```
About to create the object...
Signal received! Let's simulate a delay...
(Wait for 5 seconds...)
Done with signal after 5 seconds.
Object created!
```

As you can see, it pauses for 5 seconds because of the `time.sleep(5)` in the signal handler. Only after the signal is done does the final print statement ("Object created!") execute.

If Django signals were asynchronous, you'd expect the "Object created!" message to appear immediately without waiting for the signal handler. But that's not the case here. This is the classic example showing that Django signals, by default, are synchronous.

So yes, until you explicitly make them asynchronous, Django will handle them one by one, waiting for each task to finish before moving on.

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.**

Yes, by default, Django signals run in the same thread as the caller. This means when you trigger a signal, the signal handler (receiver) is executed in the same thread as the code that sent the signal. This behavior is part of why Django signals are synchronous, as they block the main thread until the signal handler finishes.

To prove this, we can use Python's `threading` module to check the current thread's ID in both the main code and the signal handler. If the thread IDs match, it's conclusive proof that they are running in the same thread.

Example Code:

```
# models.py
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver
import threading

class MyModel(models.Model):
    name = models.CharField(max_length=100)

# Signal handler
@receiver(post_save, sender=MyModel)
def my_signal_handler(sender, instance, **kwargs):
    print(f"Signal handler running in thread:
{threading.get_ident()}")
```

Now, in your view or shell, you create a `MyModel` instance and check the thread ID:

```
from myapp.models import MyModel
import threading

print(f"Main thread: {threading.get_ident()}")
MyModel.objects.create(name="Test Thread")
```

Output:

```
Main thread: 140281852446464  # or some thread ID
Signal handler running in thread: 140281852446464
```

Explanation:

- The `threading.get_ident()` function returns the ID of the current thread.
- When you create an instance of MyModel, it triggers the post_save signal. We check the thread ID both in the main code and inside the signal handler.
- If both thread IDs match, this confirms that the signal handler is running in the same thread as the caller.

This is solid proof that Django signals are handled in the same thread unless you explicitly make them asynchronous by using something like Celery or threading yourself.

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.**

By default, yes, Django signals run in the same transaction as the code that triggered them. So if you're saving a model and a signal (like `post_save`) is triggered, both the model save and the signal handler's work are part of the same database transaction. If something goes wrong and the transaction is rolled back, everything including what the signal handler did gets rolled back too.

Let's prove this with an example. It's like you're working on a group project, and everyone's changes only get saved when the project gets submitted. If someone messes up and the project isn't submitted, no one's changes count. The same thing happens with transactions and signals.

Here's the code:

```
# models.py
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.db import transaction

class MyModel(models.Model):
    name = models.CharField(max_length=100)

# Signal handler
@receiver(post_save, sender=MyModel)
def my_signal_handler(sender, instance, **kwargs):
    print("Signal handler running...")
```

```
    if transaction.get_connection().in_atomic_block:
        print("Yes! We're in the same transaction!")
    else:
        print("Nope, we're not in the same transaction.")

    # Modify the instance in the signal handler
    instance.name = "Changed by signal"
    instance.save()
```

Now, let's create a model instance and trigger the signal inside a transaction block, but we'll simulate an error to cause a rollback:

```
from myapp.models import MyModel
from django.db import transaction

try:
    with transaction.atomic():  # Start a transaction
        print("Starting main transaction...")
        my_instance = MyModel.objects.create(name="Original
Name")
        print(f"Model created with name: {my_instance.name}")
        # Boom! Force an error
        raise Exception("Oops, something went wrong!")
except Exception as e:
    print(e)

# Let's check if the name was changed
print(f"Final name in the DB: {MyModel.objects.first()}")
```

What happens:

1. We create the model instance, which triggers the `post_save` signal.
2. The signal handler runs, changes the name of the instance to `"Changed by signal"`.
3. Then, we simulate an error (`raise Exception`), which causes the transaction to roll back.

Output:

```
Starting main transaction...
Signal handler running...
Yes! We're in the same transaction!
Oops, something went wrong!
Final name in the DB: None  # or nothing gets saved at all
```

Explanation:

- The signal handler modifies the `name` field to `"Changed by signal"`, but because we simulated an error, everything gets rolled back. So when we check the database at the end, the model doesn't exist, or the changes aren't saved.

- If the signal handler wasn't part of the same transaction, you'd still see the modified name ("Changed by signal") in the database, even though we caused an error.

So, the bottom line is: **Yes**, Django signals run in the same transaction as the caller. If the caller's transaction rolls back, everything inside the signal handler gets rolled back too, just like when a group project doesn't get submitted because one person messed up.

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>}

- **`__init__`**: Initializes `length` and `width`, and sets an index for iteration.
- **`__iter__`**: Resets the index and returns the instance, making it iterable.
- **`__next__`**: Returns `length` first, then `width`. After both are returned, raises `StopIteration` to end the loop.

Example:

```
rect = Rectangle(10, 5)

for value in rect:
    print(value)
```

- **Output**: First returns `{'length': 10}`, then `{'width': 5}`.