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.

ANSWER: By default, Django signals are executed synchronously. This means that when a signal is sent, all connected receivers are executed one after another before the code continues execution.

Here's a code snippet to demonstrate this behavior:

```
# models.py

import time

from django.db import models

from django.db.models.signals import post_save

from django.dispatch import receiver


class MyModel(models.Model):

    name = models.CharField(max_length=100)


@receiver(post_save, sender=MyModel)

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

    print("Signal received, starting a blocking operation...")

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

    print("Blocking operation complete.")
```

test in Django

```
from myapp.models import MyModel

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

print("Model instance saved.")
```

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.

ANSWER: Yes, by default, Django signals run in the same thread as the caller. This means that when a signal is sent, it executes within the same thread as the code that triggered it.

Here's a code snippet to demonstrate this behavior:

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

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

@receiver(post_save, sender=MyModel)
def my_signal_receiver(sender, instance, **kwargs):
    current_thread = threading.current_thread()
    print(f"Signal receiver is running in thread: {current_thread.name}")
```

test in Django

```
from myapp.models import MyModel
import threading
# Print the current thread name before triggering the signal
print(f"Main thread is: {threading.current_thread().name}")

# Create an instance which will trigger the post_save signal
instance = MyModel.objects.create(name="Test")
```

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.

ANSWER: Yes, by default, Django signals run in the same database transaction as the caller. This means that if the transaction fails or is rolled back, any database operations performed by the signal will also be rolled back.

Here's a code snippet to demonstrate this behavior:

Here's a code snippet to demonstrate this behavior:

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


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


@receiver(post_save, sender=MyModel)
def my_signal_receiver(sender, instance, **kwargs):
    print("Signal receiver started.")
    # Perform a database operation within the signal
    instance.name = "Updated in signal"
    instance.save()
    print("Signal receiver finished, instance updated.")
    # Raise an exception to cause a rollback
    raise Exception("Intentional exception to rollback transaction")
```

test in Django

```
from myapp.models import MyModel
from django.db import transaction
try:
    with transaction.atomic():
        instance = MyModel.objects.create(name="Initial")
        print("Model instance saved.")
except Exception as e:
    print(f"Transaction failed with error: {e}")
# Check if the instance was updated or not
instance = MyModel.objects.filter(name="Updated in signal").exists()
print(f"Was the instance updated by the signal? {'Yes' if instance else 'No'}")
```

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

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>}**

ANSWER: Here's the implementation of the Rectangle class that meets with question's requirements:

```
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:
rectangle = Rectangle(10, 5)

for attribute in rectangle:
    print(attribute)
```

CSS

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