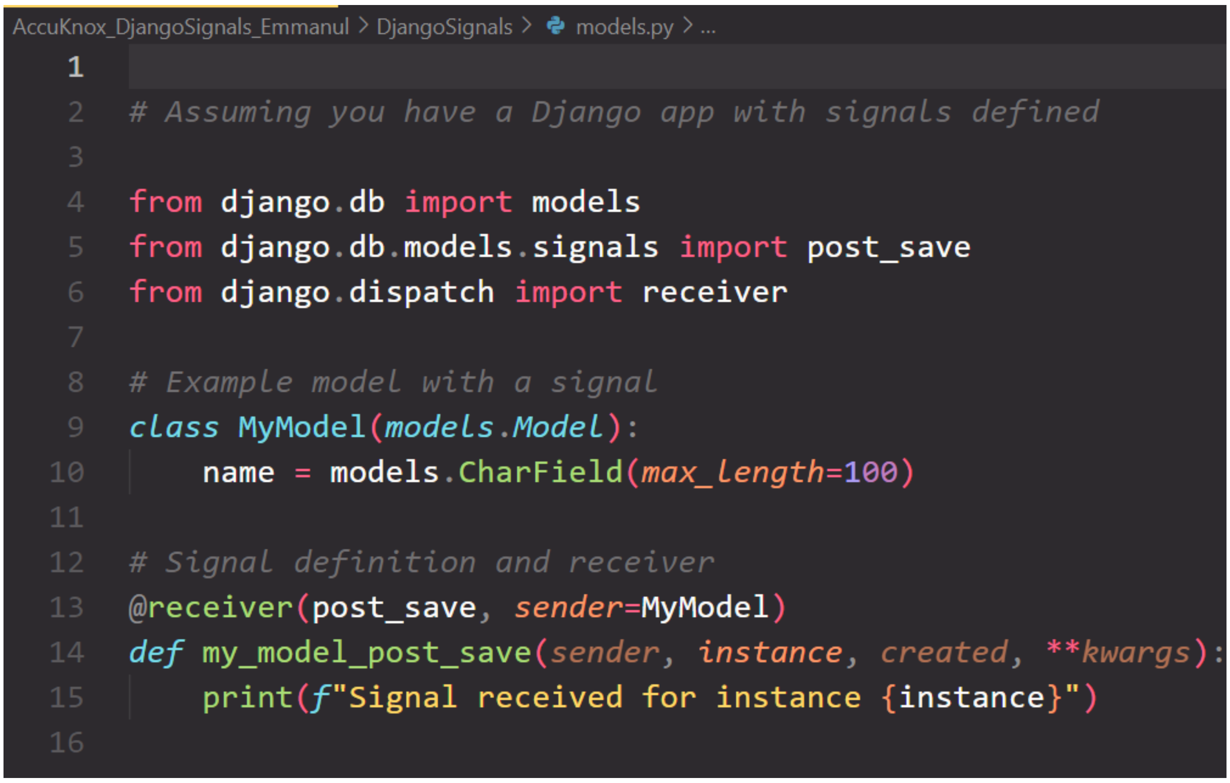
QUESTIONS & ANSWER FOR DJANGO TRAINEE AT ACCUKNOX

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, any connected receivers will be called immediately. This behaviour ensures that signals are processed in a predictable order and within the same request-response cycle in web applications.

Snap of my Code:

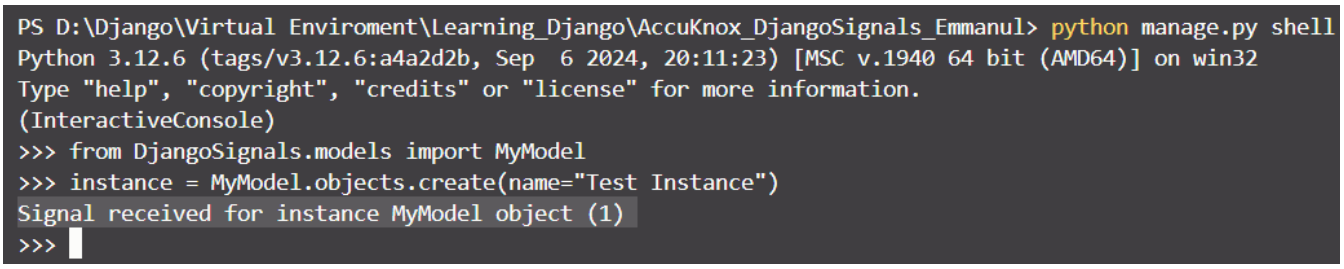

AccuKnox_DjangoSignals_Emmanul > DjangoSignals > models.py > ...

```python

# Assuming you have a Django app with signals defined

from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver

# Example model with a signal
class MyModel(models.Model):
    name = models.CharField(max_length=100)

# Signal definition and receiver
@receiver(post_save, sender=MyModel)
def my_model_post_save(sender, instance, created, **kwargs):
    print(f"Signal received for instance {instance}")

```

In this example:

- The **post_save** signal is connected to the **MyModel** model using the **@receiver decorator**.
- The **post_save** signal handler **my_model_post_save** is executed synchronously, printing a message to the console.

Output:



```
PS D:\Django\Virtual Enviroment\Learning_Django\AccuKnox_DjangoSignals_Emmanul> python manage.py shell
Python 3.12.6 (tags/v3.12.6:a4a2d2b, Sep  6 2024, 20:11:23) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from DjangoSignals.models import MyModel
>>> instance = MyModel.objects.create(name="Test Instance")
Signal received for instance MyModel object (1)
>>>
```

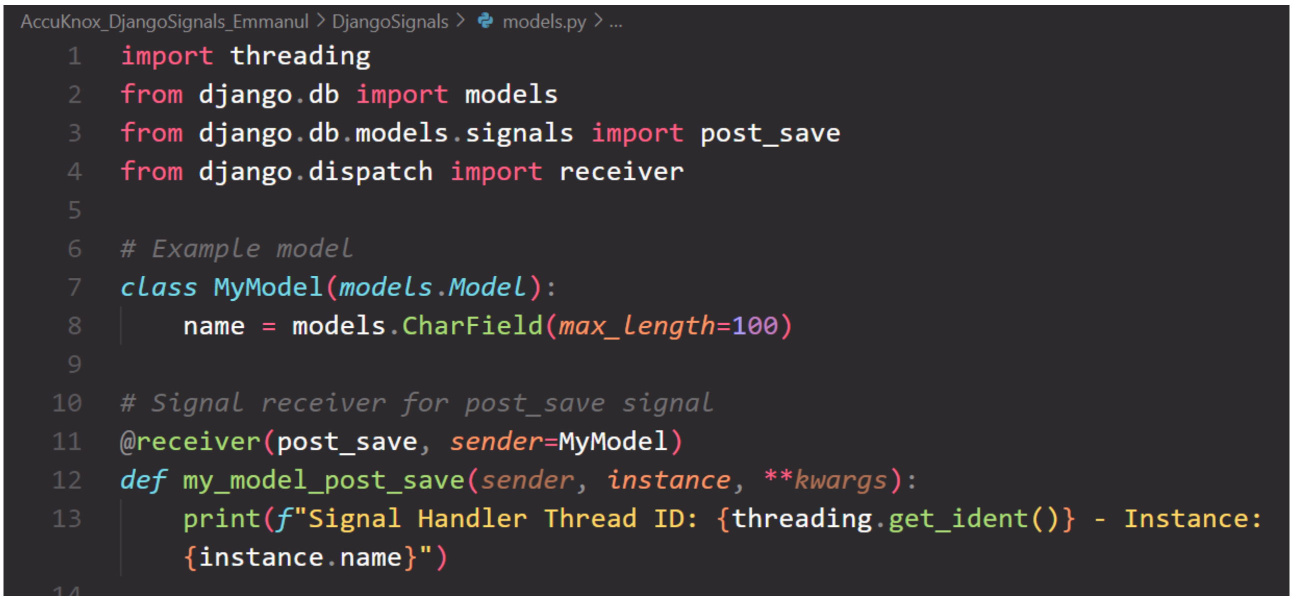
When **MyModel.objects.create()** is called, a new instance of MyModel is created and saved.

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 the signal handlers are executed synchronously, and in the same thread that triggered the signal.

We can confirm this by inspecting the thread IDs of the caller and the signal handler. If the thread IDs are the same, it indicates that the signal is running in the same thread.

Snap of my Code:



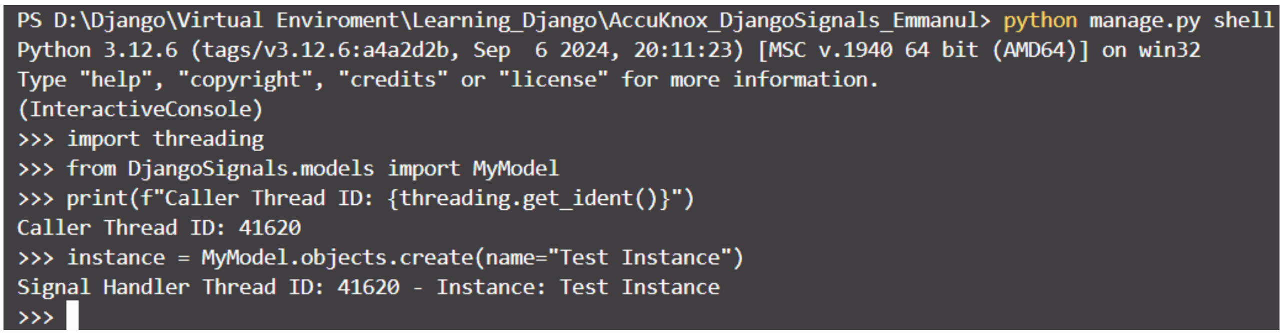
AccuKnox_DjangoSignals_Emmanul > DjangoSignals > models.py > ...

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

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

# Signal receiver for post_save signal
@receiver(post_save, sender=MyModel)
def my_model_post_save(sender, instance, **kwargs):
    print(f"Signal Handler Thread ID: {threading.get_ident()} - Instance: {instance.name}")
```

Explanation:

1. **threading.get_ident()**: This function returns the current thread's identifier. We use it to compare the thread IDs of the caller and the signal handler.
2. **Signal Handler (my_model_post_save)**: This function is connected to the **post_save** signal of **MyModel**. It prints the thread ID of the signal handler, which gets called after the model instance is saved.

Output:



```
PS D:\Django\Virtual Enviroment\Learning_Django\AccuKnox_DjangoSignals_Emmanul> python manage.py shell
Python 3.12.6 (tags/v3.12.6:a4a2d2b, Sep  6 2024, 20:11:23) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> import threading
>>> from DjangoSignals.models import MyModel
>>> print(f"Caller Thread ID: {threading.get_ident()}")
Caller Thread ID: 41620
>>> instance = MyModel.objects.create(name="Test Instance")
Signal Handler Thread ID: 41620 - Instance: Test Instance
>>>
```

Conclusion:

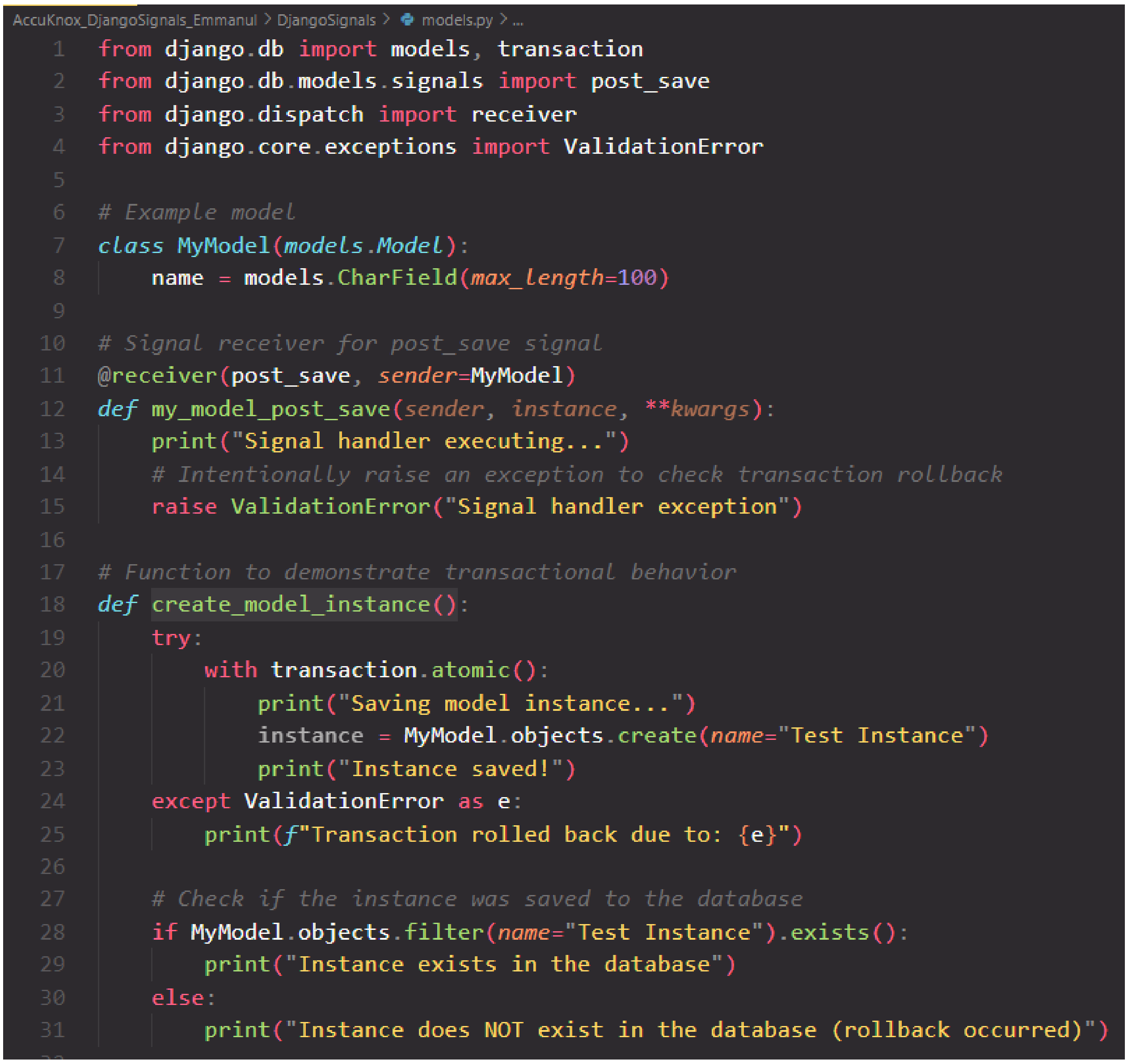
From the output, we will notice that both the caller and the signal handler run on the **same thread** (ID: 41620 in this case). **This conclusively proves that Django signals, by default, run in the same thread as the caller. If Django signals were asynchronous or ran in a separate thread, the thread IDs would be different.**

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 the signal handlers are executed within the same transaction as the operation that triggered the signal (e.g., save(), delete()).

To prove this, we can trigger a signal during a database transaction, intentionally raise an exception in the signal handler, and observe whether the transaction is rolled back when the exception occurs. If the signal runs in the same transaction, the exception raised in the signal handler should cause the entire transaction (including the save() operation) to roll back.

Snap of my Code:



AccuKnox_DjangoSignals_Emmanul > DjangoSignals > models.py > ...

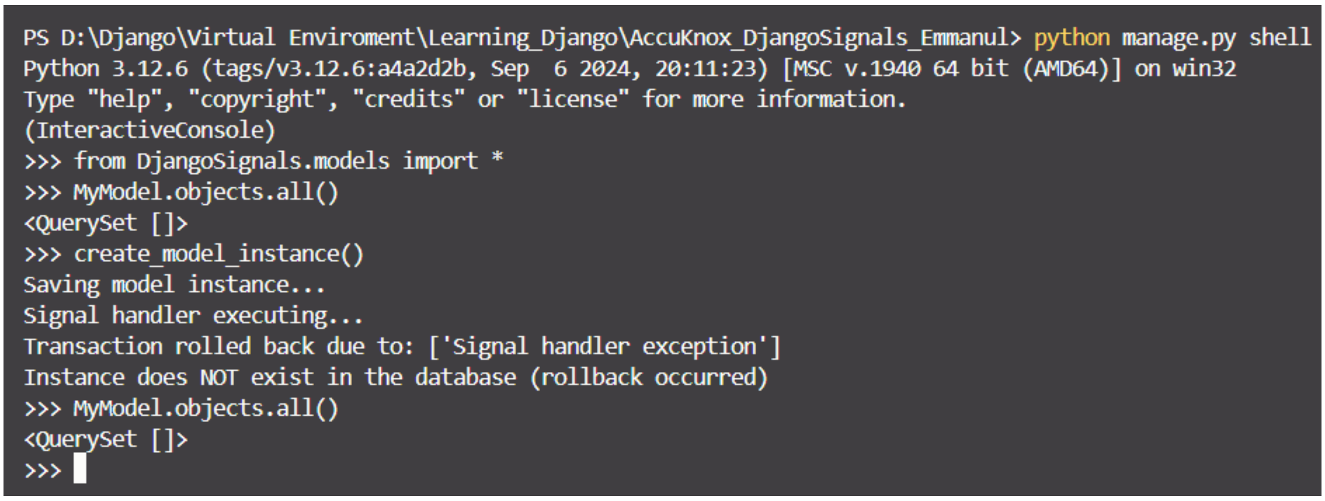
```python
from django.db import models, transaction
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.core.exceptions import ValidationError

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

# Signal receiver for post_save signal
@receiver(post_save, sender=MyModel)
def my_model_post_save(sender, instance, **kwargs):
    print("Signal handler executing...")
    # Intentionally raise an exception to check transaction rollback
    raise ValidationError("Signal handler exception")

# Function to demonstrate transactional behavior
def create_model_instance():
    try:
        with transaction.atomic():
            print("Saving model instance...")
            instance = MyModel.objects.create(name="Test Instance")
            print("Instance saved!")
    except ValidationError as e:
        print(f"Transaction rolled back due to: {e}")

    # Check if the instance was saved to the database
    if MyModel.objects.filter(name="Test Instance").exists():
        print("Instance exists in the database")
    else:
        print("Instance does NOT exist in the database (rollback occurred)")
```

Explanation:

1. **transaction.atomic()**: This ensures that all database operations inside this block are part of the same transaction. If any exception occurs, the entire transaction will be rolled back.
2. **Signal Handler (my_model_post_save)**: The post_save signal handler raises a **ValidationError** after the instance is saved. This simulates a failure within the signal, which should trigger a rollback if the signal handler runs in the same transaction.

3. **create_model_instance()**: This function creates a new instance of **MyModel** inside a transaction. If the signal handler raises an exception, the entire transaction should be rolled back, and the instance should not be saved in the database.

Output:



```
PS D:\Django\Virtual Enviroment\Learning_Django\AccuKnox_DjangoSignals_Emmanul> python manage.py shell
Python 3.12.6 (tags/v3.12.6:a4a2d2b, Sep  6 2024, 20:11:23) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from DjangoSignals.models import *
>>> MyModel.objects.all()
<QuerySet []>
>>> create_model_instance()
Saving model instance...
Signal handler executing...
Transaction rolled back due to: ['Signal handler exception']
Instance does NOT exist in the database (rollback occurred)
>>> MyModel.objects.all()
<QuerySet []>
>>> 
```

Conclusion:

In the output, we can see that even though the instance was initially saved, the exception raised in the signal handler caused the entire transaction to be rolled back. As a result, the model instance does **not** exist in the database.

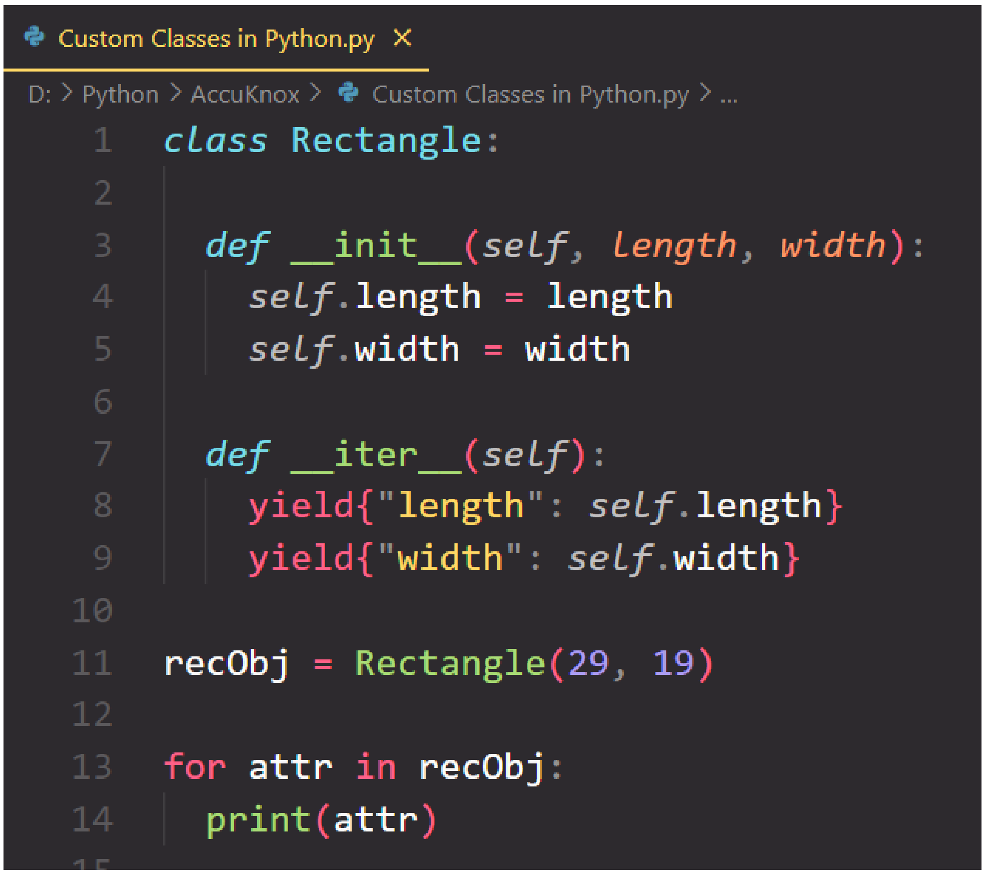
This conclusively proves that Django signals, by default, run in the same database transaction as the caller. If any exception is raised within the signal handler, it will affect the entire transaction.

Q. 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>}**

ANSWER:

Snap of my Code:


Custom Classes in Python.py
D: > Python > AccuKnox > Custom Classes in Python.py > ...

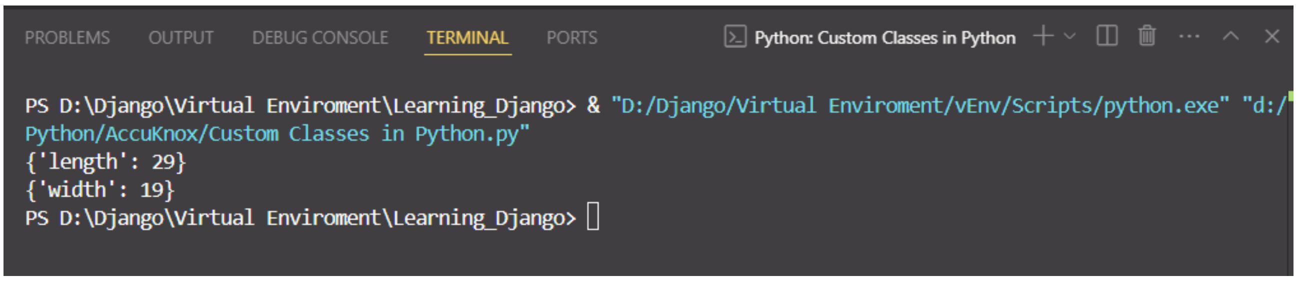
```python
class Rectangle:

  def __init__(self, length, width):
    self.length = length
    self.width = width

  def __iter__(self):
    yield{"length": self.length}
    yield{"width": self.width}

recObj = Rectangle(29, 19)

for attr in recObj:
  print(attr)
```

Explanation:

1. **__init__ method**: Initializes the Rectangle instance with length and width.

2. **__iter__ method**: This method makes the class iterable. It uses yield to return the length and width as dictionaries in the specified format.
3. **Usage**: When iterating over the rectangle instance, it first yields the length in the form {'length': <VALUE>}, followed by the width in the form {'width': <VALUE>}.

Output:



```
PS D:\Django\Virtual Enviroment\Learning_Django> & "D:/Django/Virtual Enviroment/vEnv/Scripts/python.exe" "d:/
Python/AccuKnox/Custom Classes in Python.py"
{'length': 29}
{'width': 19}
PS D:\Django\Virtual Enviroment\Learning_Django>
```