**Django Trainee Assignment - Accuknox**

**Topic 1: Django Signals**

**Question 1: Are Django Signals Executed Synchronously or Asynchronously by Default?**

**Explanation**

By default, Django signals are **executed synchronously**. This means that when a signal is triggered, it runs in the same execution flow as the sender. If the signal takes time to execute, it will block the sender from completing its operation.

Django does not provide built-in asynchronous execution for signals, but developers can manually make signals run asynchronously using **Celery** or threading.

**Code Proof**

To confirm Django signals are synchronous, let's create a signal that logs messages before and after execution.

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)

@receiver(post\_save, sender=MyModel)

def my\_signal(sender, instance, \*\*kwargs):

print("Signal execution started...")

time.sleep(5) # Simulate a delay

print("Signal execution finished!")

Now, when we create an instance:

from myapp.models import MyModel

import time

print("Creating MyModel instance...")

start\_time = time.time()

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

end\_time = time.time()

print(f"Total execution time: {end\_time - start\_time:.2f} seconds")

**Expected Output:**

Creating MyModel instance...

Signal execution started...

(Signal waits for 5 seconds)

Signal execution finished!

Total execution time: 5.XX seconds

This confirms that Django signals are synchronous because the total execution time includes the delay.

**Question 2: Do Django Signals Run in the Same Thread as the Caller?**

**Explanation**

By default, Django signals execute in the **same thread** as the event that triggered them. This means they do not create a new thread unless explicitly configured to do so.

**Code Proof**

To verify this, we will compare the thread IDs of the main execution and the signal.

import threading

@receiver(post\_save, sender=MyModel)

def my\_signal(sender, instance, \*\*kwargs):

print(f"Signal executed in thread: {threading.get\_ident()}")

Trigger the signal:

import threading

from myapp.models import MyModel

print(f"Main execution thread: {threading.get\_ident()}")

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

**Expected Output:**

Main execution thread: 140735712452480

Signal executed in thread: 140735712452480

Since both have the same thread ID, Django signals run in the same thread as the caller.

**Question 3: Do Django Signals Run in the Same Database Transaction as the Caller?**

**Explanation**

By default, Django signals **run in the same database transaction as the caller**. If the transaction fails, the signal's effects are also rolled back.

**Code Proof**

We will test whether an error in a signal causes a transaction rollback.

from django.db import transaction

@receiver(post\_save, sender=MyModel)

def my\_signal(sender, instance, \*\*kwargs):

print("Signal executing...")

raise Exception("Forcing an error in the signal!")

Trigger the signal inside a transaction:

from django.db import transaction

from myapp.models import MyModel

try:

with transaction.atomic():

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

print("Transaction committed successfully!")

except Exception as e:

print(f"Transaction failed: {e}")

**Expected Output:**

Signal executing...

Transaction failed: Forcing an error in the signal!

This confirms that Django signals run inside the same database transaction as the caller.

**Topic 2: Custom Classes in Python**

**Task: Implement a Rectangle Class**

**Requirements:**

1. The class should have length and width as required parameters.
2. The instance should be **iterable**.
3. The iteration should return:
   * {'length': value}
   * {'width': value}

**Basic Implementation**

To make an instance iterable, we need to define the \_\_iter\_\_ method.

class Rectangle:

def \_\_init\_\_(self, length: int, width: int):

self.length = length

self.width = width

self.\_attributes = [{"length": self.length}, {"width": self.width}]

def \_\_iter\_\_(self):

return iter(self.\_attributes)

**Advanced: Custom Iterator Implementation**

Instead of using iter(), we can define a custom iterator class.

class RectangleIterator:

def \_\_init\_\_(self, rectangle):

self.rectangle = rectangle

self.index = 0

self.attributes = [{"length": rectangle.length}, {"width": rectangle.width}]

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.index >= len(self.attributes):

raise StopIteration

result = self.attributes[self.index]

self.index += 1

return result

class Rectangle:

def \_\_init\_\_(self, length: int, width: int):

self.length = length

self.width = width

def \_\_iter\_\_(self):

return RectangleIterator(self)

**Testing the Code**

rect = Rectangle(10, 5)

for attr in rect:

print(attr)

**Expected Output:**

{'length': 10}

{'width': 5}