

**Answers for Django Trainee at Accuknox**

**Name :Meet Solanki**

**Gmail : meetsolanki265@gmail.com**

**Mobile No : 9638200346**

[Topic:](https://docs.djangoproject.com/en/3.2/topics/signals/) [**Django**](https://docs.djangoproject.com/en/3.2/topics/signals/)[**Signals**](https://docs.djangoproject.com/en/3.2/topics/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 :**

**django signals execute by default in a synchronously. This means that the signal handlers need to be executed in the same flow as the code calling it, and they are expected to return before the code continues.**

**Code Snippet :**

**from django.db.models.signals import post\_save**

**from django.dispatch import receiver**

**from django.contrib.auth.models import User**

**import time**

**@receiver(post\_save, sender=User)**

**def signal\_handler(sender, instance, \*\*kwargs):**

**print("Signal handler started.")**

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

**print("Signal handler finished.")**

**Explaination :**

**The following code snippet demonstrates that Django signals execute by default in a synchronously. In the scenario of a save operation on a `User` model instance, the `post\_save` signal was received, and the function that was connected to it, `signal\_handler`, is called. The function accepts the signal for the `User` model with decorator `@receiver` assigned. Inside the handler, the line `time.sleep(5)` simulates a long time consumer while waiting 5 seconds to finish. The fact that Django processes signals in a synchronous nature means the signal handler completes whatever it needs to do before the calling process, such as an HTTP request, can continue.**

**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 ,because by default django signals will run within the same thread as the caller, which ensures signal handlers to be raised within the same thread that triggered the signal.**

**Code Snippet :**

**import threading**

**from django.db.models.signals import post\_save**

**from django.dispatch import receiver**

**from django.contrib.auth.models import User**

**@receiver(post\_save, sender=User)**

**def signal\_handler(sender, instance, \*\*kwargs):**

**print(f"Signal handler running in thread: {threading.current\_thread().name}")**

**def create\_user():**

**print(f"Caller running in thread: {threading.current\_thread().name}")**

**user = User.objects.create(username="test\_user")**

**Explaination :**

**This code is an example of how to use Django signals to call an action every time a `User` model instance is saved. It connects the `post\_save` signal with `signal\_handler`, decorated with `@receiver` of saves for the `User` model. It includes the following steps for the 'User' saving signal handler: prints "Signal handler started," while next continues to call 'time.sleep(5)' to mimic a long-running task, printing "Signal handler finished" after this. This suggests that the signal handler runs synchronously because it blocks and waits before any other processing can take place.**

**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, Django signals are triggered within the same database transaction by default. This means that if the transaction fails, the signal handler's changes are also rolled back.**

**Code Snippet :**

**from django.db.models.signals import post\_save**

**from django.dispatch import receiver**

**from django.db import transaction**

**from django.contrib.auth.models import User**

**@receiver(post\_save, sender=User)**

**def signal\_handler(sender, instance, \*\*kwargs):**

**if kwargs.get('created', False):**

**print("Signal: User created, transaction is active:", transaction.get\_connection().in\_atomic\_block)**

**# In your views.py**

**from django.db import transaction**

**def create\_user():**

**with transaction.atomic():**

**user = User.objects.create(username="test\_user")**

**print("User saved, transaction is active:", transaction.get\_connection().in\_atomic\_block)**

**Explaination :**

**This code shows that Django signals are triggered within the same database transaction. The `post\_save` signal for the `User` model is connected to a handler that checks if it's inside an active transaction using `transaction.get\_connection().in\_atomic\_block`. In the `create\_user` function, a user is created within a transaction block (`transaction.atomic()`). Both the signal and the function share the same transaction, ensuring that if the transaction fails, any changes in the signal handler are also rolled back.**

# 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>}**

**Answer :**

**class Rectangle:**

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

**self.length = length**

**self.width = width**

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

**self.current = 0**

**return self**

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

**if self.current == 0:**

**self.current += 1**

**return {'length': self.length}**

**elif self.current == 1:**

**self.current += 1**

**return {'width': self.width}**

**else:**

**raise StopIteration**

**obj = Rectangle(10, 5)**

**for dimension in obj:**

**print(dimension)**

**Output :**

****