Questions for Django Trainee at Accuknox

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 triggered, all connected receiver functions are executed one after another, in the order they were registered, and the process waits for each receiver to complete before moving on.

A simple code snippet to demonstrate this:

```
import time
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.contrib.auth.models import User

# Define a receiver function
@receiver(post_save, sender=User)
def my_receiver(sender, instance, **kwargs):
    print("Receiver started")
    time.sleep(5)  # Simulate a long-running task
    print("Receiver finished")

# Simulate saving a user instance
```

```
user = User(username='testuser')
user.save()
```

In this example:

1. The my_receiver function is connected to the post_save signal of the User model.
2. When a user instance is saved, the my_receiver function is called.
3. The time.sleep(5) line simulates a long-running task, causing the function to pause for 5 seconds.
4. The print statements will show that the receiver starts and finishes, demonstrating that the signal handling is synchronous.

When you run this code, you'll see the following output:

```
Receiver started
(5 seconds pause)
Receiver finished
```

This confirms that the signal is handled synchronously, as the code execution waits for the receiver function to complete 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.

Answer = Yes, by default, Django signals run in the same thread as the caller. This means that when a signal is triggered, the connected receiver functions are executed in the same thread that triggered the signal.

a code snippet to demonstrate this:

```
import threading
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.contrib.auth.models import User
```

```
# Define a receiver function
@receiver(post_save, sender=User)
def my_receiver(sender, instance, **kwargs):
    print(f"Receiver thread: {threading.current_thread().name}")

# Simulate saving a user instance
def save_user():
    print(f"Caller thread: {threading.current_thread().name}")
    user = User(username='testuser')
    user.save()

# Run the save_user function
save_user()
```

In this example:

1. The my_receiver function is connected to the post_save signal of the User model.
2. When a user instance is saved, the my_receiver function is called.
3. Both the caller function (save_user) and the receiver function (my_receiver) print the name of the current thread.

When you run this code, you'll see output similar to this:

```
Caller thread: MainThread
Receiver thread: MainThread
```

This output confirms that both the caller and the receiver are running in the same thread, demonstrating that Django signals are executed in the same thread as the caller.

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 = By default, Django signals run in the same database transaction as the caller. This means that if a signal is triggered within a transaction, the signal handlers will also be executed within that same transaction.

A code snippet to demonstrate this:

```
from django.db import transaction
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.contrib.auth.models import User
from django.db import connection

# Define a receiver function
@receiver(post_save, sender=User)
def my_receiver(sender, instance, **kwargs):
    cursor = connection.cursor()
    cursor.execute("SELECT txid_current()")
    txid = cursor.fetchone()[0]
    print(f"Receiver transaction ID: {txid}")

# Simulate saving a user instance within a transaction
def save_user():
    with transaction.atomic():
        cursor = connection.cursor()
        cursor.execute("SELECT txid_current()")
        txid = cursor.fetchone()[0]
        print(f"Caller transaction ID: {txid}")
```

```
    user = User(username='testuser')
    user.save()

# Run the save_user function
save_user()
```

In this example:

1. The my_receiver function is connected to the post_save signal of the User model.
2. Both the caller function (save_user) and the receiver function (my_receiver) print the current transaction ID using txid_current() from PostgreSQL.
3. The save_user function wraps the user save operation in a transaction using transaction.atomic().

When you run this code, you'll see output similar to this:

```
Caller transaction ID: 12345
Receiver transaction ID: 12345
```

This output confirms that both the caller and the receiver are running within the same database transaction, as the transaction IDs are identical.

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):
        yield {'length': self.length}
        yield {'width': self.width}

# Example usage:
rect = Rectangle(10, 5)
for dimension in rect:
    print(dimension)
```

Explanation:

1. **Initialization**: The __init__ method initializes the Rectangle instance with length and width.
2. **Iteration**: The __iter__ method is defined to make the Rectangle instance iterable. It uses the yield statement to first return the length in the required format and then the width.

When you create an instance of Rectangle and iterate over it, you will get the output in the specified format:

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

This implementation ensures that the Rectangle class meets the specified requirements.