Accuknox Assignment

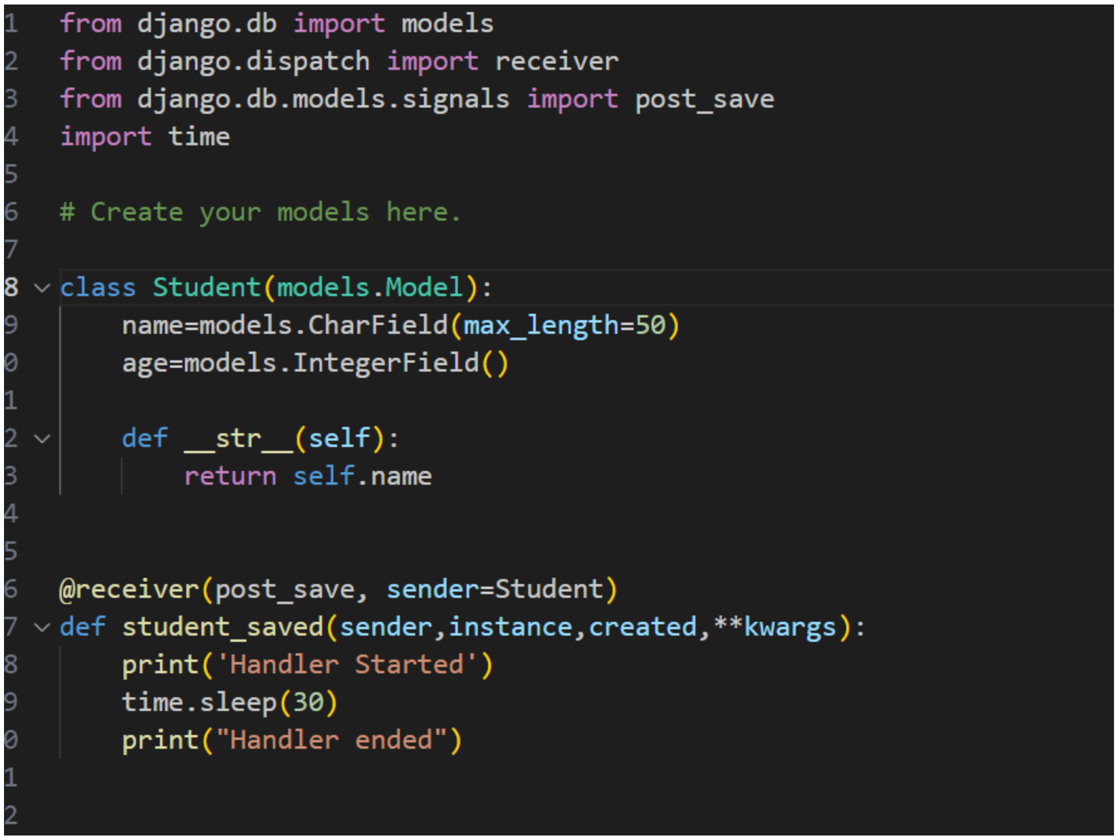
Question 1: By default are django signals executed synchronously or asynchronously? Please support your answer with a code snippet that conclusively proves your stance. The code does not need to be elegant and production ready, we just need to understand your logic.

Answer:
By default, Django signals are executed synchronously. This means that when a signal is emitted, all the connected signal handlers (receiver functions) are called immediately, and the code execution does not proceed until all the signal handlers finish executing.

We can prove that Django signals are executed synchronously by showing that the code waits for the signal handler to complete before continuing execution.

Here is a simple code snippet to demonstrate this:

Models.py:

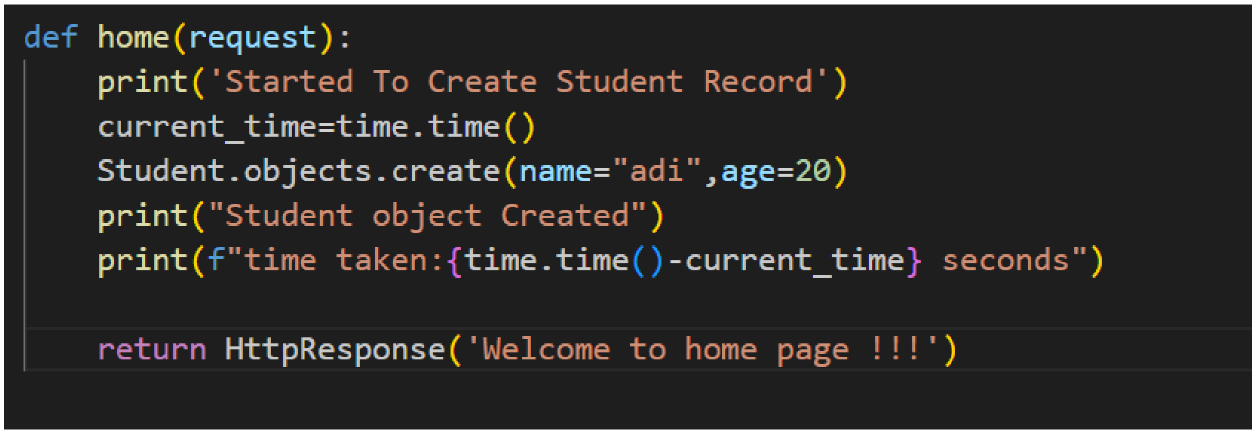


```python
from django.db import models
from django.dispatch import receiver
from django.db.models.signals import post_save
import time

# Create your models here.

class Student(models.Model):
    name=models.CharField(max_length=50)
    age=models.IntegerField()

    def __str__(self):
        return self.name


@receiver(post_save, sender=Student)
def student_saved(sender,instance,created,**kwargs):
    print('Handler Started')
    time.sleep(30)
    print("Handler ended")
```

Accuknox Assignment

In the models.py file, a simple Student model is defined with two fields: name and age. A signal handler (student_saved) is connected to the post_save signal, which triggers every time a Student instance is saved. This handler simulates a long-running task by introducing a 30-second delay using time.sleep(30) before printing a completion message.

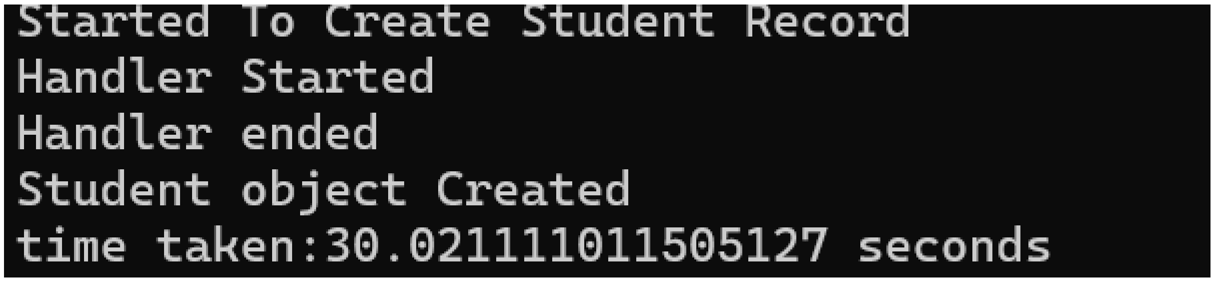
Views.py:



```python
def home(request):
    print('Started To Create Student Record')
    current_time=time.time()
    Student.objects.create(name="adi",age=20)
    print("Student object Created")
    print(f"time taken:{time.time()-current_time} seconds")

    return HttpResponse('Welcome to home page !!!')
```

In the views.py file, the home view is responsible for creating a new Student object. It starts by printing a message and recording the current time. Once the Student object is created, the post_save signal triggers the student_saved handler, which introduces a 30-second delay. After the handler completes, the view prints a confirmation that the object was created and calculates the total time taken for the operation.

Output:



```
Started To Create Student Record
Handler Started
Handler ended
Student object Created
time taken:30.021111011505127 seconds
```

Accuknox Assignment

When this code runs, you can observe that after the message "Started To Create Student Record" is printed, the program does not immediately proceed to the next steps. Instead, it waits for the signal handler to finish executing, which takes 30 seconds due to the time.sleep(30) call. Only after the handler prints "Handler ended" does the view print "Student object Created" and display the total time taken, which includes the 30-second delay.

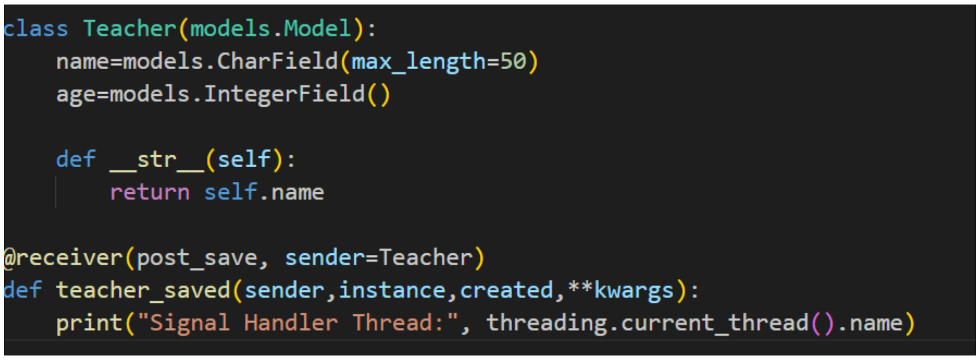
This output conclusively demonstrates that Django signals are executed synchronously. The total time taken to create the Student object includes the time spent in the signal handler, proving that the main flow of execution waits for the handler to finish. If the signals were asynchronous, the view would not have waited for the handler to complete, and the total time would have been much shorter, excluding the 30-second delay.

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, Django signals run in the same thread as the caller by default. When a signal is triggered, it is handled synchronously in the same thread that initiated the signal. We can prove this using a simple code snippet where we print the thread information inside both the caller and the signal handler.

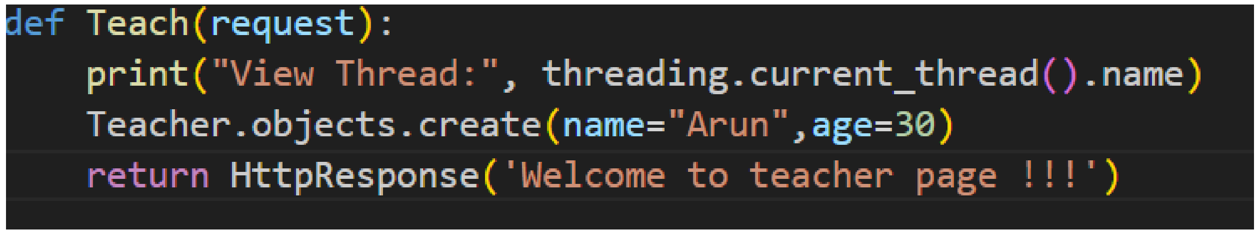
Models.py:



```
class Teacher(models.Model):
    name=models.CharField(max_length=50)
    age=models.IntegerField()

    def __str__(self):
        return self.name

@receiver(post_save, sender=Teacher)
def teacher_saved(sender,instance,created,**kwargs):
    print("Signal Handler Thread:", threading.current_thread().name)
```

Accuknox Assignment

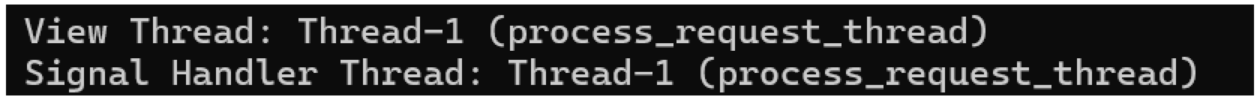
Views.py:



```python
def Teach(request):
    print("View Thread:", threading.current_thread().name)
    Teacher.objects.create(name="Arun",age=30)
    return HttpResponse('Welcome to teacher page !!!')
```

In this code, we have a Teacher model with name and age fields. A signal handler teacher_saved is connected to the post_save signal, which fires whenever a new Teacher instance is saved. Both in the view function Teach and in the signal handler, we print the current thread using threading.current_thread().name to check whether they are running in the same thread.

Output:



```
View Thread: Thread-1 (process_request_thread)
Signal Handler Thread: Thread-1 (process_request_thread)
```

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.

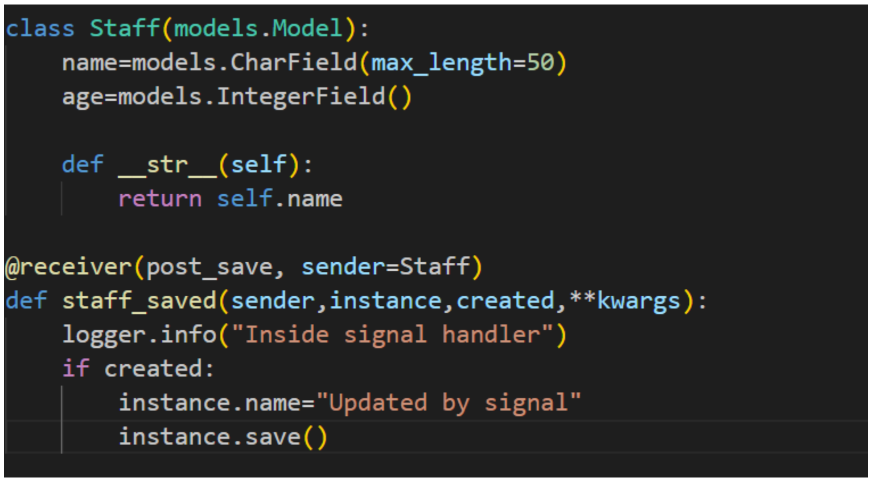
Answer:

By default, Django signals are executed within the same database transaction as the caller. This means that if a signal is triggered as part of a database operation, the signal handler operates within the same transaction and will be affected by any commit or rollback that happens to the caller's transaction.

To demonstrate this, we can use a code snippet that includes a transaction management setup to show how the signal handler and the main operation are part of the same transaction.

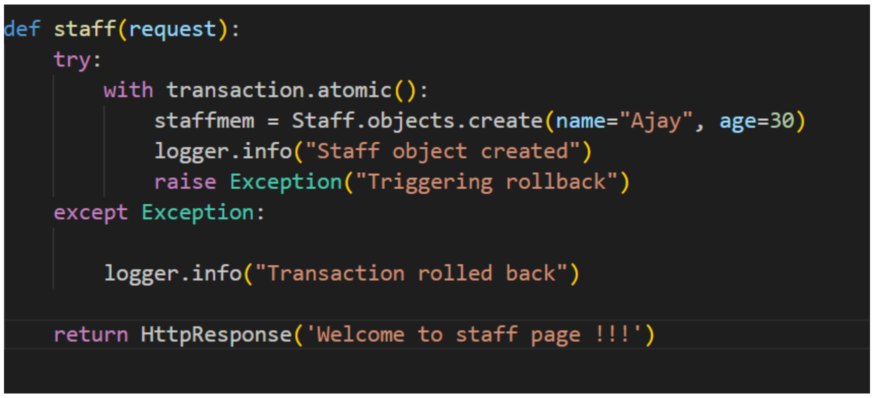
Accuknox Assignment

Models.py:



```python
class Staff(models.Model):
    name=models.CharField(max_length=50)
    age=models.IntegerField()

    def __str__(self):
        return self.name

@receiver(post_save, sender=Staff)
def staff_saved(sender,instance,created,**kwargs):
    logger.info("Inside signal handler")
    if created:
        instance.name="Updated by signal"
        instance.save()
```

Views.py:



```python
def staff(request):
    try:
        with transaction.atomic():
            staffmem = Staff.objects.create(name="Ajay", age=30)
            logger.info("Staff object created")
            raise Exception("Triggering rollback")
    except Exception:

        logger.info("Transaction rolled back")

    return HttpResponse('Welcome to staff page !!!')
```

In models.py, a Teacher model is defined with a signal handler named staff_saved connected to the post_save signal. This signal handler updates the name of the Teacher instance and saves it again. The crucial aspect here is that any changes made by this signal handler will be rolled back if the transaction initiated by the caller is rolled back.

In views.py, the Staff view function demonstrates this concept. The view creates a Teacher object within a transaction managed by transaction.atomic(). After the object is created, the view deliberately raises an exception to trigger a rollback of the entire transaction. This approach ensures that all database changes, including those made by the signal handler, will be rolled back.

If the transaction is rolled back, the creation of the Staff object and any modifications made by the signal handler will not be committed to the database. This outcome is significant because it shows that the signal handler operates within the same transaction as the caller.

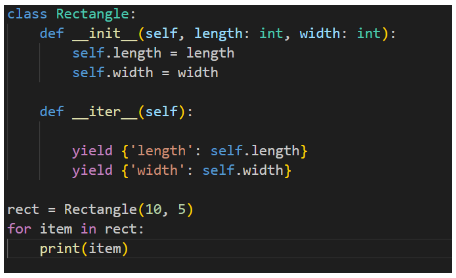
When the Staff view is accessed, it leads to the creation of the Staff object, which triggers the post_save signal and executes the staff_saved handler. The handler attempts to update and save the Staff object. However, due to the exception, the entire transaction is rolled back. As a result, none of the changes made by the initial object creation or by the signal handler are committed to the database.

This example effectively demonstrates that Django signals run in the same database transaction as the caller. The rollback of the transaction affects both the main operation and the signal handler, confirming that signal handling is synchronous with respect to database transactions by default.

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>}**

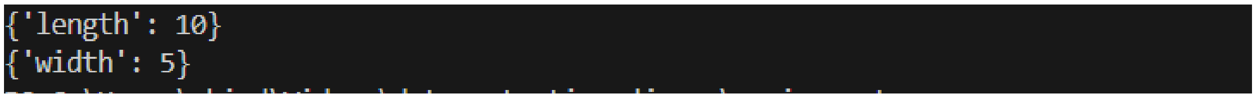
Rectangle.py:



```python
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}

rect = Rectangle(10, 5)
for item in rect:
    print(item)
```

Accuknox Assignment

Output:



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