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.

Ans: By default, Django signals execute synchronously.In Django, signals are sent and received synchronously, meaning that the code that sends the signal will wait for the signal handlers to complete before continuing execution. This can potentially slow down your application if the signal handlers perform time-consuming tasks.

Consider the code snippet

```
from django.http import HttpResponse
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.shortcuts import render
import time
class Student(models.Model):
    Name=models.CharField(max_length=100)
    Place=models.CharField(max_length=50)

@receiver(post_save,sender=Student)
def signal_handling(sender,instance,**kwargs):
    print(f"Application received for {instance.Name}")
    time.sleep(2)
    print(f"Application process is completed for
{instance.Name}")

def Application_submission(request):
    print("Registration stared")

    Student.objects.create(Name='sachin',Place='chennai')
    print("Application submited succesfully")
    return HttpResponse("Registration Completed")
```

This code snippet demonstrates how django signals are executed synchronously by default.There is a student model having Name and Place fields. This signal handler listens for `post_save` signals from the `Student` model. When a `Student` instance is saved, the handler prints a message, waits for 2 seconds to simulate processing time, and then prints a completion message. The`Application_submission`

`Function` creates a `Student` instance, which triggers the `post_save` signal. It prints a start message, creates a student, and then prints a success message.

The `signal_handling` function is invoked synchronously as part of the `post_save` signal. It prints "Application received for sachin." The function then sleeps for 2 seconds to simulate processing time.

After the delay, it prints "Application process is completed for sachin." Only after the signal handler completes (including the sleep time) does the view function proceed to print "Application submitted successfully." Finally, the HTTP response "Registration Completed" is returned to the user.

The key point here is that the view waits for the `signal_handling` function to complete before it prints "Application submitted successfully" and returns the HTTP response. This demonstrates that Django signals execute synchronously by default, as the view does not proceed until the signal handler has finished its execution.

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

ANS.Yes by default django signals run synchronously in the same thread as the sender (caller).so when a signal is sent, all of its connected receiver functions are executed one after another, and the caller must wait for all of these to complete before proceeding.Django also supports asynchronous handling if you want them to run asynchronously, you need to manage that explicitly using Python's `asyncio`, threading, or other asynchronous tools. So Django signals are not required to run in the same thread as the caller, but by default, they do. To make them run asynchronously, you need to manually handle the asynchronous execution.

Consider the code snippet

```
from django.http import HttpResponse
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.shortcuts import render
import time
import threading
class Student(models.Model):
    Name=models.CharField(max_length=100)
    Place=models.CharField(max_length=50)

@receiver(post_save,sender=Student)
def signal_handling(sender,instance,**kwargs):
    print(f"Application received for {instance.Name}")
    # Printing the current thread's ID
    print(f"The thread id : {threading.current_thread().ident}")
    time.sleep(2)
    print(f"Application process is completed for {instance.Name}")

def Application_submission(request):
    print("Registration stared")
    # Creating an instance of Student
    Student.objects.create(Name='sachin',Place='chennai')
    # Printing the current thread's ID
    print(f"The thread id : {threading.current_thread().ident}")
    print("Application submited succesfully")
    return HttpResponse("Registration Completed")
```

In this code the `Student` model has two fields : `Name` and `Place`. Signal_handling function is a receiver for the `post_save` signal, which is sent every time a `Student` instance is saved. When triggered, it prints the message that the application has been received, along with the current thread ID using `threading.current_thread().ident`. It then pauses for 2 seconds to simulate a delay using `time.sleep(2)` and prints another message indicating that the application process is complete.

The Application_submission view function is triggered when a user makes a request . It prints "Registration started" to indicate the beginning of the process. A new `Student` object is created with the `Name` set to "sachin" and `Place` set to "chennai". This action triggers the `post_save` signal. The view function prints the thread ID, indicating the thread in which the

view function is running. After creating the `Student` object, it prints "Application submitted successfully" and returns an HTTP response with the text "Registration Completed".The thread ID printed by `signal_handling` and the thread ID printed by `Application_submission` are the same. This confirms that the signal handler (`signal_handling`) is executed in the same thread as the view function

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.

Ans: Yes, by default, Django signals run in the same database transaction as the caller. When a signal is sent, Django executes the signal handlers within the same database transaction as the code that sent the signal. This means that if the signal handlers perform database operations, they will be part of the same transaction as the original code. If the transaction is rolled back, any changes made by the signal handlers will also be rolled back. This ensures data consistency and integrity.

Consider this code

```
from django.http import HttpResponse
from django.db import models, transaction
from django.db.models.signals import post_save
from django.dispatch import receiver
flag = None
class Student(models.Model):
    Name = models.CharField(max_length=100)
    Place = models.CharField(max_length=50)
@receiver(post_save, sender=Student)
def signal_handling(sender, instance, **kwargs):
    global flag  # using global variable to prevent recursion
    if flag is not None and flag > 0:
        return
    print(f"Application received for {instance.Name}")
    # updating flag to prevent recursive calls
    flag = 1
    instance.Place = "Updated by Signal"
    instance.save()  # This will not trigger recursion due to the flag check
    print(f"Application processed for {instance.Name}")
    flag = None
def Application_submission(request):
    print("Registration started")
    try:
        with transaction.atomic():

            student = Student.objects.create(Name='Sachin', Place='Chennai')
            print("Student created.")
            # Intentionally raise an exception to force a rollback
            raise Exception("raising an error to verify transaction rollback
behavior")
    except Exception as e:
        print(f"Exception caught: {e}")
    # Check if the Student object still exists in the database
    student_exists = Student.objects.filter(Name='Sachin').exists()
    print(f"Does the student exist after rollback? {student_exists}")
    return HttpResponse("Registration Completed")
```

This code demonstrates that Django signals run in the same database transaction as the caller by using the `post_save` signal and `transaction.atomic()`. The `post_save` signal is connected to the `signal_handling` function, which is triggered after a `Student` instance is saved. The `transaction.atomic()` context manager is used to group database operations into a single transaction. If an exception is raised within this block, all changes made in the transaction are rolled back. When `Student.objects.create(Name='Sachin', Place='Chennai')` is called, a `post_save` signal is triggered. The `signal_handling`

function updates the `Place` field of the `Student` instance and attempts to save it again. An exception is deliberately raised within the `transaction.atomic()` block to force a rollback. Due to this rollback, any changes made within the transaction, including those made by the signal handler, are undone. This signal handler updates the `Place` field and saves the instance again. Since the signal is processed within the same transaction, any changes it makes will be rolled back if the transaction is rolled back. This function creates a `Student` instance, which triggers the `post_save` signal. Due to the `transaction.atomic()` block, when the exception is raised, the database operation is rolled back. The output will show that the `Student` instance does not exist after the rollback, proving that both the signal's database operations and the main transaction operations are part of the same transaction. The code demonstrates that Django signals run in the same database transaction as the caller. When an exception within a `transaction.atomic()` block causes a rollback, any changes made by signal handlers are also rolled back, proving that signals are part of the same transaction.

4. 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': } followed by the width {width:}

Ans:

```
class Rectangle:
    def __init__(self, length: int, width: int):
        self.length = length
        self.width = width
    def __iter__(self):
        # Use a generator to yield the length and
width
        yield {'length': self.length}
        yield {'width': self.width}
rect = Rectangle(20, 10)
# Iterating over the Rectangle instance
for item in rect:
    print(item)
```