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 are executed synchronously. When a signal is sent, the connected signal handlers are executed immediately, in the same thread as the code that sent the signal.

Code example:

class MyModel(models.Model):

name = models.CharField(max\_length=100)

@receiver(post\_save, sender=MyModel)

def my\_signal\_handler(instance, created, \*\*kwargs):

print(f'Signal received: {instance.name}, created: {created}')

def create\_instance():

instance = MyModel(name='Test')

instance.save()

create\_instance()

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.

Code example:

class MyModel(models.Model):

name = models.CharField(max\_length=100)

@receiver(post\_save, sender=MyModel)

def my\_signal\_handler(instance, created, \*\*kwargs):

current\_thread = threading.current\_thread()

print({current\_thread.name})

def create\_instance():

print({threading.current\_thread().name})

instance = MyModel(name='Test')

instance.save()

if \_\_name\_\_ == "\_\_main\_\_":

create\_instance()

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, by default, Django signals run in the same database transaction as the caller. If a signal is triggered during a database operation, any database changes made by the signal handlers are part of the same transaction and can be rolled back if an exception occurs.

Code example:

class MyModel(models.Model):

name = models.CharField(max\_length=100)

@receiver(post\_save, sender=MyModel)

def my\_signal\_handler(instance, created, \*\*kwargs):

if created:

raise ValidationError("An error occurred in the signal handler.")

def create\_instance():

try:

with transaction.atomic():

instance = MyModel(name='Test')

instance.save()

except ValidationError as e:

print(Caught an error: {e})

if \_\_name\_\_ == "\_\_main\_\_":

create\_instance()

**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}

rect = Rectangle(5, 10)

for dimension in rect:

print(dimension)