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.

Ans : Django signals are, by default, executed in a synchronous manner. This means that once a signal is sent, Django will wait for all associated signal handlers to complete their execution before continuing with the next steps.

To confirm this behaviour, we can set up a signal handler that includes a delay. This will help us see if Django waits for the handler to finish before proceeding.

```
import time
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 my_handler(sender, instance, **kwargs):
    print("Signal received, starting delay...")
    time.sleep(5)
    print("Signal handler finished execution.")
  user = User.objects.create(username="test_user")
  print("User created, signal should have executed by now.")
```

output:

```
Signal received, starting delay...
[5-second delay]
Signal handler finished execution.
User created, signal should have been executed by now.
```

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.

ANS : Yes, Django signals, by default, operate in the same thread as the code that triggers them. This means that when a signal is fired, any connected signal handlers are executed within the same thread context as the sender.

We can confirm this by using Python's `threading` module to capture and compare the thread IDs in both the calling code and the signal handler.

```
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 my_handler(sender, instance, **kwargs):

    print("Signal handler thread ID:", threading.get_ident())

print("Main thread ID:", threading.get_ident())

user = User.objects.create(username="test_user")
```

Output:

Main thread ID: <some_thread_id>

Signal handler thread ID: <same_thread_id_as_main>

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.

ANS:

By default, Django signals are executed within the same database transaction as the code that triggers them. This means that if the caller's transaction is rolled back, any database changes made within the signal handler will also be rolled back.

To demonstrate this, we can set up a signal handler that performs a database operation, then force a rollback in the caller's transaction. If the changes made in the signal handler are also rolled back, this will confirm that the signal runs within the same transaction context.

```
class Profile(models.Model):

    user = models.OneToOneField(User, on_delete=models.CASCADE)

    bio = models.TextField(default="")

@receiver(post_save, sender=User)

def create_profile(sender, instance, **kwargs):
```

```
    Profile.objects.create(user=instance)

    print("Profile created for user:", instance.username)

def test_transaction():

    try:

        with transaction.atomic():

            user = User.objects.create(username="test_user")

            print("User created:", user.username)

                    raise Exception("Forcing rollback")

    except Exception as e:

        print("Transaction rolled back:", e)

test_transaction()print("Profile exists:",
Profile.objects.filter(user__username="test_user").exists())
```

OUTPUT:

```
User created: test_user

Profile created for user: test_user

Transaction rolled back: Forcing rollback

Profile exists: False
```

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>}`**

```
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(1, 3)
for dimension in rect:
    print(dimension)
```

OUTPUT

{'length': 1}

{'width': 3}