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 executed synchronously**, meaning they run in the same process and thread as the sender for ex:-

```
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_signal_handler(sender, instance, **kwargs):
    print("Signal started execution")
    time.sleep(5)  # Simulating a delay
    print("Signal execution completed")

User.objects.create(username="testuser")
print("User creation complete")
```

Output:- By using a post_save signal and adding time.sleep() inside the signal handler. If the signal were asynchronous, it wouldn't block the main thread.

```
Signal started execution
(Sleeps for 5 seconds)
Signal execution completed
User creation complete
```

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:- Django signals run in the same thread as the caller. We can verify this by printing the thread ID inside the signal and the caller function. For ex:-

```
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_signal_handler(sender, instance, **kwargs):
            print(f"Signal Thread ID: {threading.get_ident()}")
```

```
# Creating a User instance
print(f"Main Thread ID: {threading.get_ident()}")
User.objects.create(username="testuser")
```

Output:- Since both the main function and the signal print the same thread ID, it proves that Django signals run in the same thread as the caller.

```
Main Thread ID: 140736420345728
Signal Thread ID: 140736420345728  (Same as main 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.

Ans:- Django signals run in the same database transaction by default. If the transaction rolls back, the signal's execution will be rolled back as well. For ex:- We will create a user inside a transaction and raise an exception **after** the post_save signal fires. If the transaction is rolled back, the user should not be saved.

```
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

@receiver(post_save, sender=User)
def my_signal_handler(sender, instance, **kwargs):
    print("Signal executed.")

try:
    with transaction.atomic():
        user = User.objects.create(username="testuser")
        print("User saved in DB")
        raise Exception("Rolling back transaction")

except Exception as e:
    print(f"Exception occurred: {e}")

# Checking if the user exists in the database
print("User exists:", User.objects.filter(username="testuser").exists())
```

Output:-

```
Signal executed.
```

```
User saved in DB
Exception occurred: Rolling back transaction
User 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**

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>}

Ans:-

```
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}

# Usage
rectangle = Rectangle(10, 20)

for dimension in rectangle:
  print(dimension)
```

Output:-

```
{'length': 10}
{'width': 20}
```