<u>Answer 1:</u> Django signals are synchronous by default. This means that when a signal is triggered, Django waits for the signal handler to complete before continuing execution.
For more understanding I give you Code below -

```
from django.dispatch import Signal
import time
my_signal = Signal()
def receiver_1(sender, **kwargs):
    print("Receiver 1 called")
    time.sleep(1)
def receiver_2(sender, **kwargs):
    print("Receiver 2 called")
my_signal.connect(receiver_1)
my_signal.connect(receiver_2)
my_signal.send(sender=__name__)
```

<u>Answer 2:</u> Yes, Django signals always run in the same thread as the caller, meaning that when you trigger a signal within a specific thread, all connected signal handlers will also execute within that same thread.
Code for understanding:

```
from django.dispatch import receiver
from django.db.models.signals import post_save
from threading import Thread

def my_signal_handler(sender, instance, **kwargs):
    print(threading.current_thread().name)
    post_save.connect(my_signal_handler)

def my_function_that_triggers_signal():
    print(threading.current_thread().name)
    my_model_instance = MyModel.objects.create()

thread = Thread(target=my_function_that_triggers_signal)
thread.start()
thread.join()
```

<u>Answer 3:</u> No, by default Django signals do not run within the same database transaction , they are executed when each database operation is committed immediately, outside of the current transaction.
Code below -

```
from django.db import models, transaction
from django.dispatch import receiver
from django.db.models.signals import post_save

class MyModel(models.Model):
    name = models.CharField(max_length=255)

def my_signal_handler(sender, instance, created, **kwargs):
    print("Signal triggered - updating related data")
    # Do some operation on the database here

def create_model_with_signal(data):
    with transaction.atomic():
        my_object = MyModel(name=data['name'])
        my_object.save()
```

<u>Topic Solution</u>:

A custom class in Python is a user-defined blueprint for creating objects.

```
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}

length = int(input("Enter the length of the rectangle: "))
width = int(input("Enter the width of the rectangle: "))

rect = Rectangle(length,width)
for attribute in rect:
    print(attribute)
```