**Question no 1.**

By default Django signals are executed synchronously, here we use **post\_save** signal for a model and add a time delay.

**Creating a model,**

#models.py

from django.db import models

class book(models.Model):

title = models.CharField(max\_length=200)

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

**then create a signals.py file**

# signals.py

from django.db.models.signals import post\_save

from django.dispatch import receiver

from .models import Book

import time

def book\_save(sender, instance, \*\*kwargs):

print("started")

time.sleep(5) # Delay for 5 seconds to simulate a long task

print("finished")

**Test this(Djnago shell):**

from myapp.models import Book

import time

start = time.time()

book = Book(title="Test Book", author="Author Name")

book.save() # This will trigger the post\_save signal

print(f"completed in {time.time() - start:.2f} sec")

**Question 2.**

By default, Django signals execute in the same thread as the caller. This means that when a signal is triggered, all associated signal handlers are run immediately and synchronously within the same thread as the code that triggered the signal.

Example:

**# models.py**

from django.db import models

class Book(models.Model):

title = models.CharField(max\_length=200)

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

We will set up a signal handler that prints the thread ID for both the signal handler and the main thread to show that they are identical.

**# signals.py**

from django.db.models.signals import post\_save

from django.dispatch import receiver

from .models import Book

import threading

def on\_book\_save(sender, instance, \*\*kwargs):

print(f"running in thread: {threading.current\_thread().name}")

**To test this, run the following in the Django shell:**

from myapp.models import Book

import threading

print(f"Main thread: {threading.current\_thread().name}") # Print main thread info

book = Book(title="Test Book", author="Author Name")

book.save() # This will trigger the post\_save signal

**Question 3**

By default, Django signals are executed within the same database transaction as the triggering code, ensuring transactional consistency. If a signal is raised during a transaction, its handlers will operate within the same context, reflecting any changes made up to that point.

Let's use two models: Book and Log.

**# models.py**

from django.db import models

class Book(models.Model):

title = models.CharField(max\_length=200)

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

class Log(models.Model):

message = models.CharField(max\_length=255)

**# signals.py**

Create a signal handler that logs a message when a Book instance is saved.

from django.db.models.signals import post\_save

from django.dispatch import receiver

from .models import Book, Log

def log\_book\_save(sender, instance, \*\*kwargs):

Log.objects.create(message=f"Book saved: {instance.title}")

**Test this:**

from myapp.models import Book, Log

from django.db import transaction

# Check initial state

print("Initial Log count:", Log.objects.count()) # Should be 0

# Start a transaction

with transaction.atomic():

book = Book(title="Test Book", author="Author Name")

book.save() # This will trigger the post\_save signal and create a log entry

# Check if the Log entry is created within the transaction

print("Log count inside transaction:", Log.objects.count()) # Should show 1

# After the transaction is committed, check the Log count again

print("Log count after transaction commit:", Log.objects.count())

**Topic: Custom Classes in Python**

class Rectangle:

    def \_\_init\_\_(self, length, width):

        self.length = length

        self.width = width

    def \_\_iter\_\_(self):

        yield {'length': self.length}

        yield {'width': self.width}

rect = Rectangle(5, 3)

for item in rect:

    print(item)