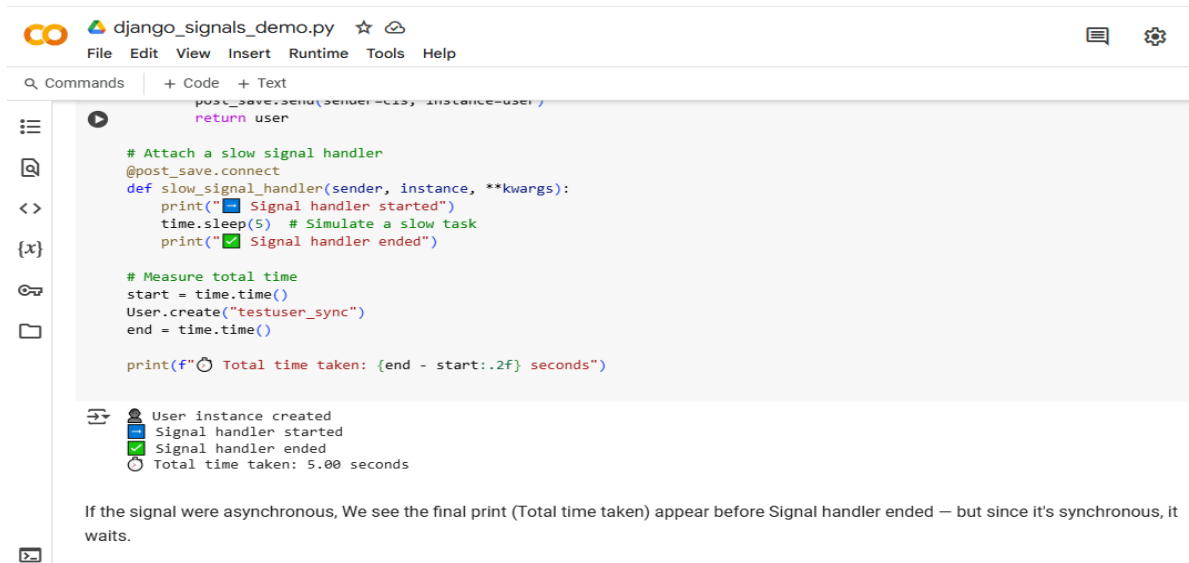


TOPIC : Django Signal

Question 1: By default are Django signals executed synchronously or asynchronously?

✓ Answer: Yes, Django signals are synchronous by default. That means the signal handler runs immediately when the signal is sent, and the main function waits for the signal handler to finish.



The screenshot shows a code editor with a file named `django_signals_demo.py`. The code defines a `post_save` signal handler for the `User` model. The handler, `slow_signal_handler`, prints "Signal handler started", sleeps for 5 seconds, and then prints "Signal handler ended". The main code measures the total time taken to create a user and run the handler, printing "Total time taken: 5.00 seconds". The output console shows the execution flow: "User instance created", "Signal handler started", "Signal handler ended", and "Total time taken: 5.00 seconds". A note at the bottom states: "If the signal were asynchronous, We see the final print (Total time taken) appear before Signal handler ended — but since it's synchronous, it waits."

```
django_signals_demo.py
File Edit View Insert Runtime Tools Help

Commands | + Code + Text

# Attach a slow signal handler
@post_save.connect
def slow_signal_handler(sender, instance, **kwargs):
    print("🔵 Signal handler started")
    time.sleep(5) # Simulate a slow task
    print("✅ Signal handler ended")

# Measure total time
start = time.time()
User.create("testuser_sync")
end = time.time()

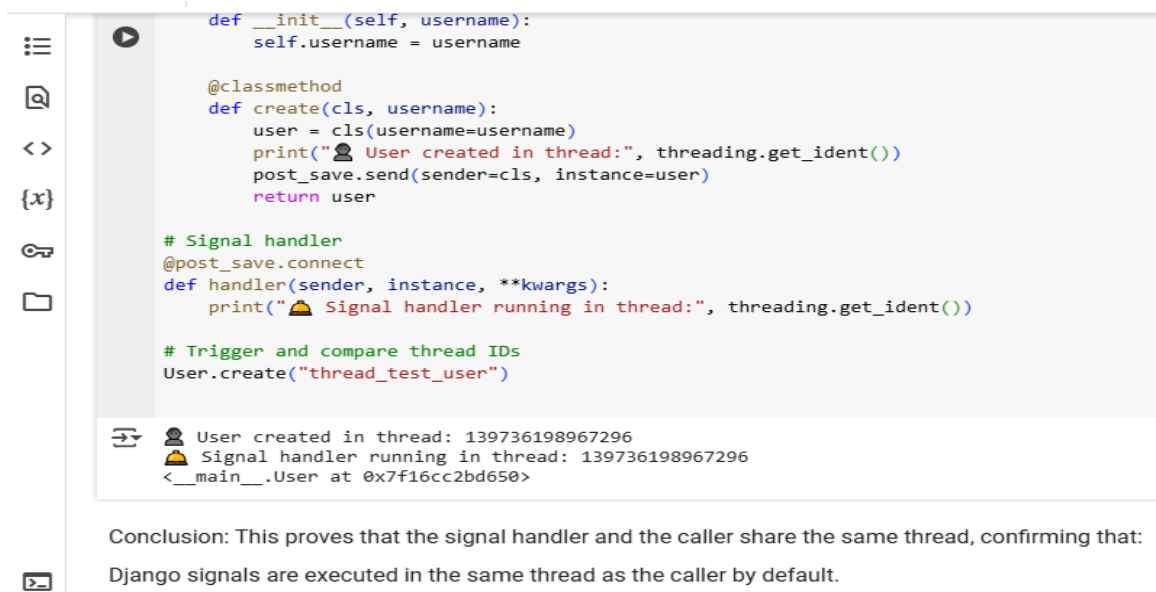
print(f"🕒 Total time taken: {end - start:.2f} seconds")

👤 User instance created
🔵 Signal handler started
✅ Signal handler ended
🕒 Total time taken: 5.00 seconds

If the signal were asynchronous, We see the final print (Total time taken) appear before Signal handler ended — but since it's synchronous, it waits.
```

Question 2: Do django signals run in the same thread as the caller?

✓ Answer: Yes, by default Django signals run in the same thread as the function that triggered the signal.



The screenshot shows a code editor with a file named `thread_test.py`. The code defines a `post_save` signal handler for the `User` model. The handler, `handler`, prints "Signal handler running in thread:" followed by the thread ID. The main code creates a user and prints "User created in thread:" followed by the thread ID. The output console shows the execution flow: "User created in thread: 139736198967296", "Signal handler running in thread: 139736198967296", and "<__main__.User at 0x7f16cc2bd650>". A note at the bottom states: "Conclusion: This proves that the signal handler and the caller share the same thread, confirming that: Django signals are executed in the same thread as the caller by default."

```
def __init__(self, username):
    self.username = username

@classmethod
def create(cls, username):
    user = cls(username=username)
    print("👤 User created in thread:", threading.get_ident())
    post_save.send(sender=cls, instance=user)
    return user

# Signal handler
@post_save.connect
def handler(sender, instance, **kwargs):
    print("🔵 Signal handler running in thread:", threading.get_ident())

# Trigger and compare thread IDs
User.create("thread_test_user")

👤 User created in thread: 139736198967296
🔵 Signal handler running in thread: 139736198967296
<__main__.User at 0x7f16cc2bd650>

Conclusion: This proves that the signal handler and the caller share the same thread, confirming that:
Django signals are executed in the same thread as the caller by default.
```

3rd question explanation and logic below

Question 3: By default do django signals run in the same database transaction as the caller?

Answer: Yes, by default Django signals are executed inside the same database transaction as the operation that triggered them. So if the transaction is rolled back, any changes made in the signal handler will also be rolled back.



```
django_signals_demo.py
File Edit View Insert Runtime Tools Help

Commands + Code + Text

if raise_exception:
    raise Exception("Simulated error after signal")
    return user

# Run transaction and trigger rollback
print("Starting transaction...")
try:
    with atomic():
        create_user("txn_test_user", raise_exception=True)
except:
    print("Transaction failed.")

print("\nFinal Logs in 'DB':", FAKE_DB["logs"])

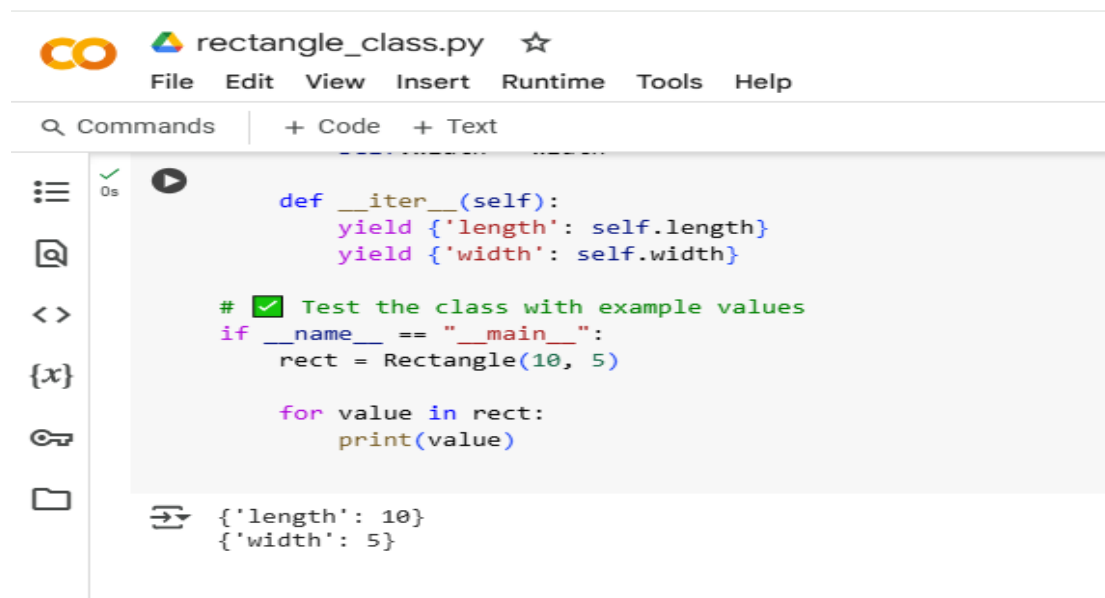
Starting transaction...
Creating log in signal handler...
Exception occurred, rolling back logs...

Final Logs in 'DB': []
```

Conclusion: Even though the signal handler created a log, it was rolled back when the main operation failed. This proves that Django signal handlers execute within the same transaction as the caller.

Topic: Custom Classes in Python

Rectangle.py



```
rectangle_class.py
File Edit View Insert Runtime Tools Help

Commands + Code + Text

def __iter__(self):
    yield {'length': self.length}
    yield {'width': self.width}

# Test the class with example values
if __name__ == "__main__":
    rect = Rectangle(10, 5)

    for value in rect:
        print(value)

{'length': 10}
{'width': 5}
```