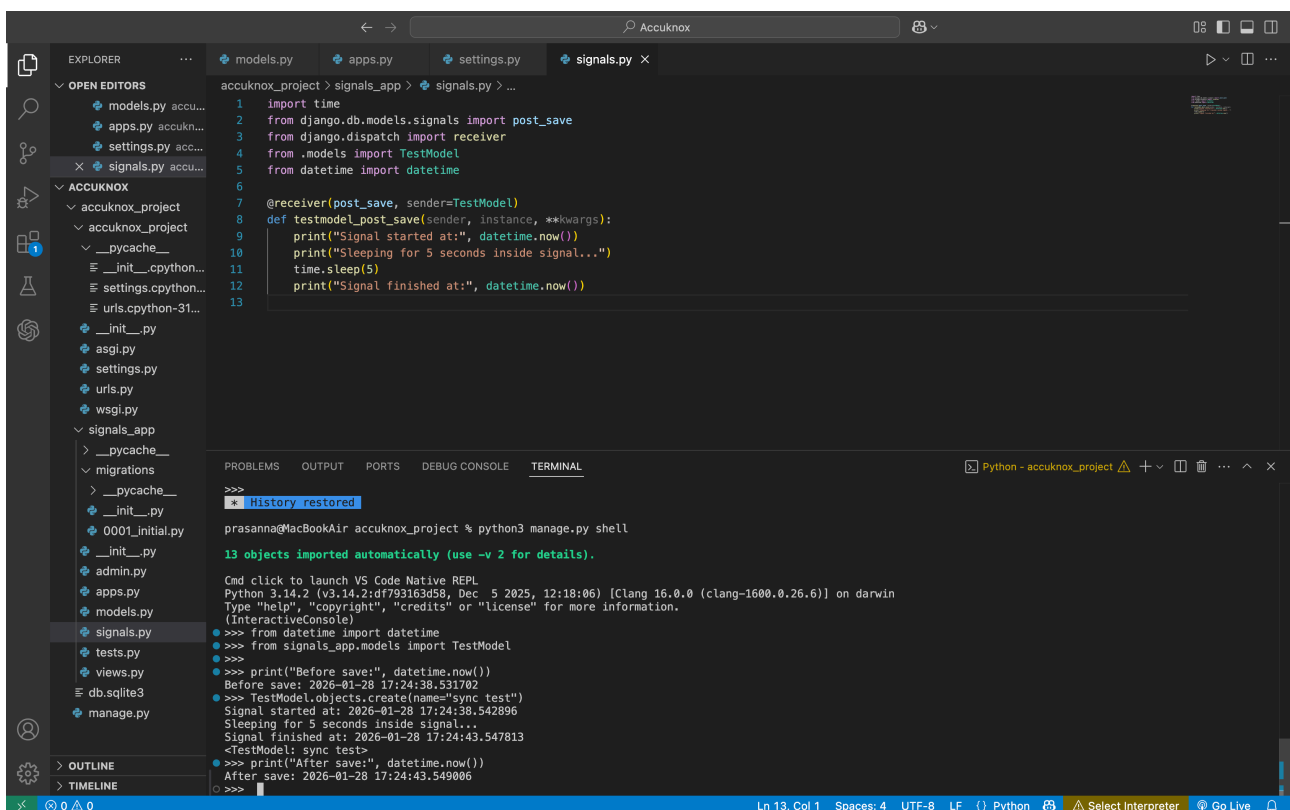


Accuknox Django Trainee Assignment: Signals & Python

Topic: Django signals

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.

Answer: By default, Django signals are executed **synchronously**. This means that when a signal is triggered, the caller waits until the signal handler finishes execution before continuing further execution. Django does not execute signals in the background or in a separate asynchronous process unless explicitly implemented by the developer.

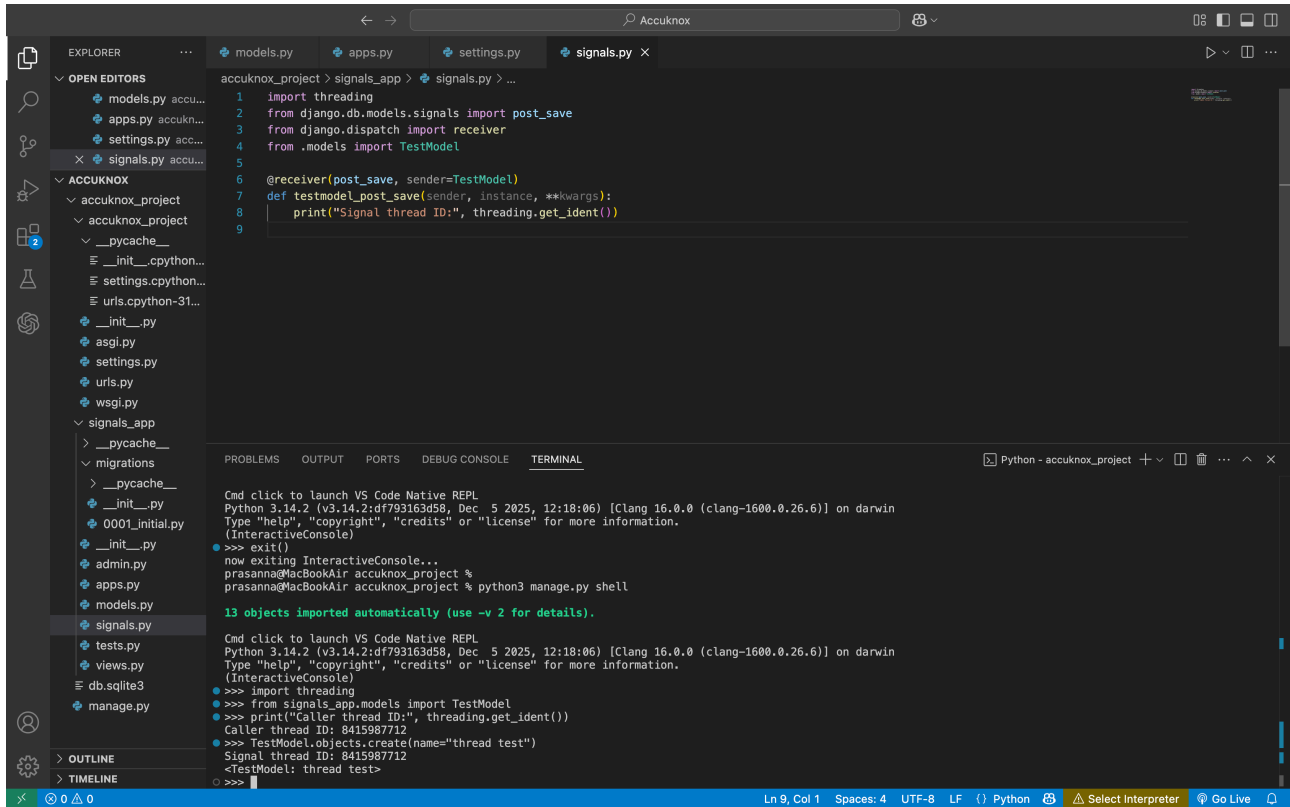


```
1 import time
2 from django.db.models.signals import post_save
3 from django.dispatch import receiver
4 from .models import TestModel
5 from datetime import datetime
6
7 @receiver(post_save, sender=TestModel)
8 def testmodel_post_save(sender, instance, **kwargs):
9     print("Signal started at:", datetime.now())
10    print("Sleeping for 5 seconds inside signal...")
11    time.sleep(5)
12    print("Signal finished at:", datetime.now())
13
```

```
>>>
* History restored
prasanna@MacBookAir accuknox_project % python3 manage.py shell
13 objects imported automatically (use -v 2 for details).
Cnd click to launch VS Code Native REPL
Python 3.14.2 (v2.14.2:df793163d58, Dec 5 2025, 12:18:06) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from datetime import datetime
>>> from signals_app.models import TestModel
>>>
>>> print("Before save:", datetime.now())
Before save: 2026-01-28 17:24:38.531702
>>> TestModel.objects.create(name="sync test")
Signal started at: 2026-01-28 17:24:38.542896
Sleeping for 5 seconds inside signal...
Signal finished at: 2026-01-28 17:24:43.547813
<TestModel: sync test>
>>> print("After save:", datetime.now())
After save: 2026-01-28 17:24:43.549006
>>>
```

The screenshot shows Django shell output with timestamps printed before saving the model, during signal execution, and after saving. A deliberate delay was introduced inside the `post_save` signal using `time.sleep(5)`. The After save timestamp appears only after the signal handler finishes execution, clearly demonstrating that the code following `.create()` executes **only after** the signal completes. This confirms that Django signals are executed synchronously by default.

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.



The screenshot shows a VS Code editor with a Django project named 'accuknox'. The file explorer on the left shows the project structure, including 'signals_app'. The main editor window shows the 'signals.py' file with the following code:

```
1 import threading
2 from django.db.models.signals import post_save
3 from django.dispatch import receiver
4 from .models import TestModel
5
6 @receiver(post_save, sender=TestModel)
7 def testmodel_post_save(sender, instance, **kwargs):
8     print("Signal thread ID:", threading.get_ident())
9
```

The terminal at the bottom shows the output of the Django shell. It displays the thread ID of the caller (8415987712) and the thread ID of the signal handler (8415987712), proving that they are the same.

```
Cmd click to launch VS Code Native REPL
Python 3.14.2 (v3.14.2:df793163d58, Dec 5 2025, 12:18:06) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> exit()
now exiting InteractiveConsole...
prasananna@MacBookAir accuknox_project % python3 manage.py shell
prasananna@MacBookAir accuknox_project % python3 manage.py shell

13 objects imported automatically (use -v 2 for details).

Cmd click to launch VS Code Native REPL
Python 3.14.2 (v3.14.2:df793163d58, Dec 5 2025, 12:18:06) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> import threading
>>> from signals_app.models import TestModel
>>> print("Caller thread ID:", threading.get_ident())
Caller thread ID: 8415987712
>>> TestModel.objects.create(name="thread test")
Signal thread ID: 8415987712
<TestModel: thread test>
>>>
```

Answer: The screenshot shows the thread ID printed in the Django shell before saving the model and the thread ID printed inside the signal handler. Both values are identical, proving that Django signals execute in the same thread as the caller.

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.

```
models.py  apps.py  settings.py  signals.py

OPEN EDITOR... 1 unsaved
  models.py accu...
  apps.py accukn...
  settings.py accu...
  signals.py accu...

ACCUKNOX
  accuknox_project
    accuknox_project
      __pycache__
      __init__.cpython...
      settings.cpython...
      urls.cpython-31...
      __init__.py
      asgi.py
      settings.py
      urls.py
      wsgi.py
    signals_app
      __pycache__
      migrations
      __pycache__
      __init__.py
      0001_initial.py
      __init__.py
      admin.py
      apps.py
      models.py
      signals.py
      tests.py
      views.py
      db.sqlite3
      manage.py

PROBLEMS  OUTPUT  PORTS  DEBUG CONSOLE  TERMINAL

Python - accuknox_project

SyntaxError: invalid syntax
>>> from django.db import transaction
>>> from signals_app.models import TestModel
>>> try:
... with transaction.atomic():
  File "<console>", line 2
    with transaction.atomic():
        ^^^^^
IndentationError: expected an indented block after 'try' statement on line 1
>>> from django.db import transaction
>>> from signals_app.models import TestModel
>>> try:
... with transaction.atomic():
...     TestModel.objects.create(name="transaction test")
... except Exception as e:
...     print("Exception caught:", e)
...
print("Objects in DB:", TestModel.objects.all())
Signal executed - raising exception to test transaction
Exception caught: Forcing rollback from signal
>>> print("Objects in DB:", TestModel.objects.all())
Objects in DB: <QuerySet [<TestModel: first test>, <TestModel: thread test>]>
>>>
```

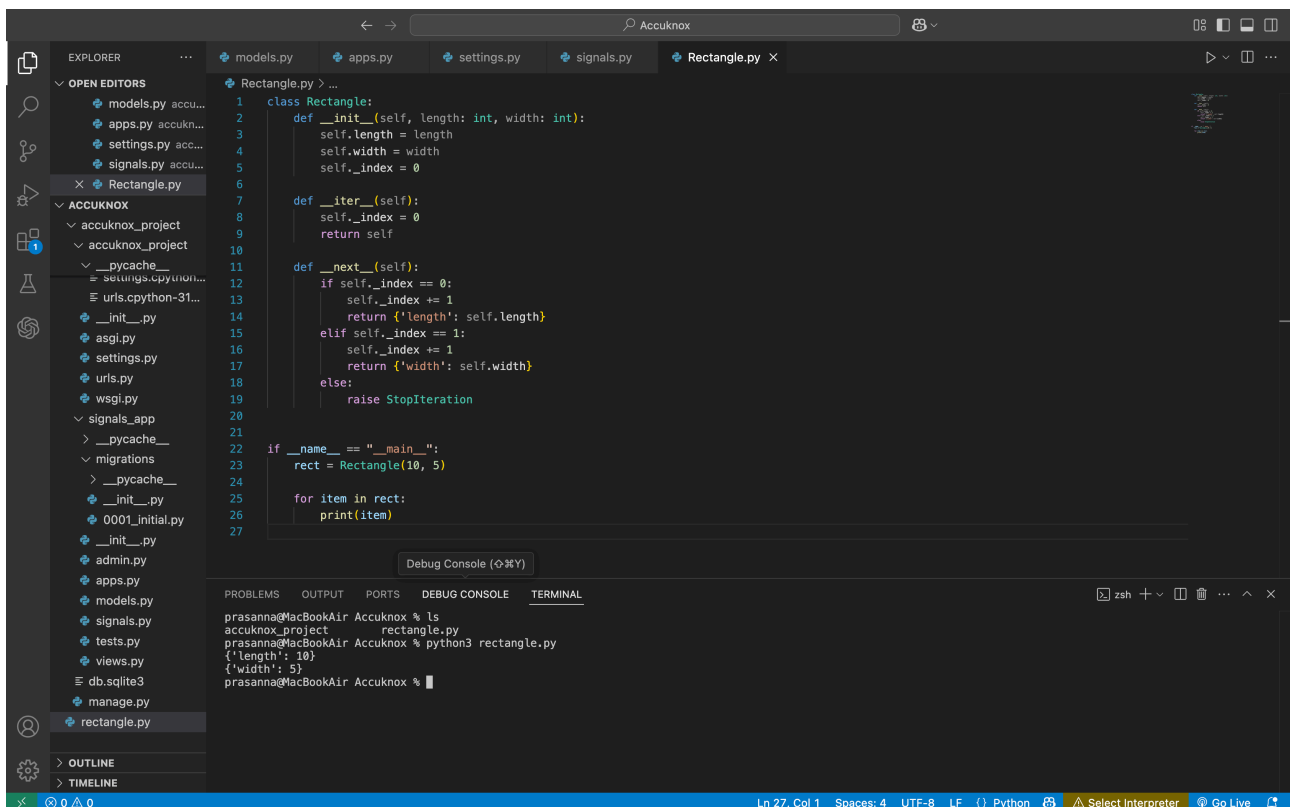
Answer: The screenshot shows Django shell execution where a model instance is created inside a `transaction.atomic()` block. During signal execution, an exception is raised intentionally, which causes the transaction to roll back. When querying the database afterward, the newly created object is not present, confirming that the signal execution and the model save occur within the same database transaction.

Topic: Custom Classes in Python

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>}**

Answer: The objective is to create a **Rectangle** class that requires **length** and **width** during initialization and supports iteration. When iterated, the instance should return the length first in dictionary format, followed by the width in dictionary format.



```
1 class Rectangle:
2     def __init__(self, length: int, width: int):
3         self.length = length
4         self.width = width
5         self._index = 0
6
7     def __iter__(self):
8         self._index = 0
9         return self
10
11     def __next__(self):
12         if self._index == 0:
13             self._index += 1
14             return {'length': self.length}
15         elif self._index == 1:
16             self._index += 1
17             return {'width': self.width}
18         else:
19             raise StopIteration
20
21 if __name__ == "__main__":
22     rect = Rectangle(10, 5)
23
24     for item in rect:
25         print(item)
26
27
```

Debug Console (Python)

```
prasan@MacBookAir Accuknox % ls
accuknox_project
prasan@MacBookAir Accuknox % python3 rectangle.py
{'length': 10}
{'width': 5}
prasan@MacBookAir Accuknox %
```

The **Rectangle** class successfully fulfills all the specified requirements by enforcing initialization with **length** and **width** and implementing Python's iterator protocol. By defining the **__iter__()** and **__next__()** methods, the class allows seamless iteration over its instances, returning the length first and the width next in the required dictionary format. This implementation demonstrates a clear understanding of custom class design and controlled iteration in Python while maintaining simplicity and correctness.