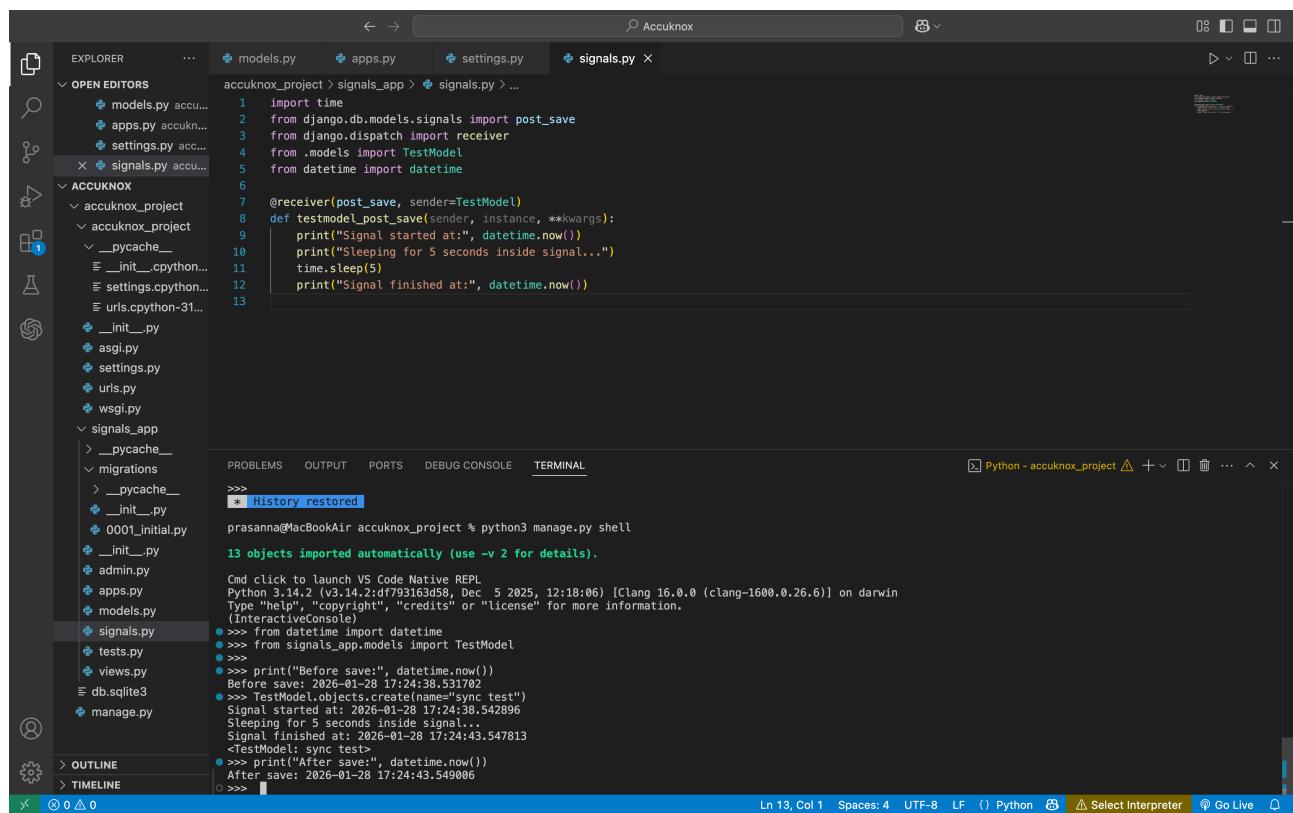


# Accuknox Django Trainee Assignment: Signals & Python

## 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.



The screenshot shows a VS Code interface with the following details:

- Explorer View:** Shows the project structure with files like `models.py`, `apps.py`, `settings.py`, and `signals.py` in the `signals\_app` directory.
- Code Editor:** Displays a Python file `signals.py` containing the following code:

```

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

```
- Terminal:** Shows the output of the Django shell command `python3 manage.py shell`:

```

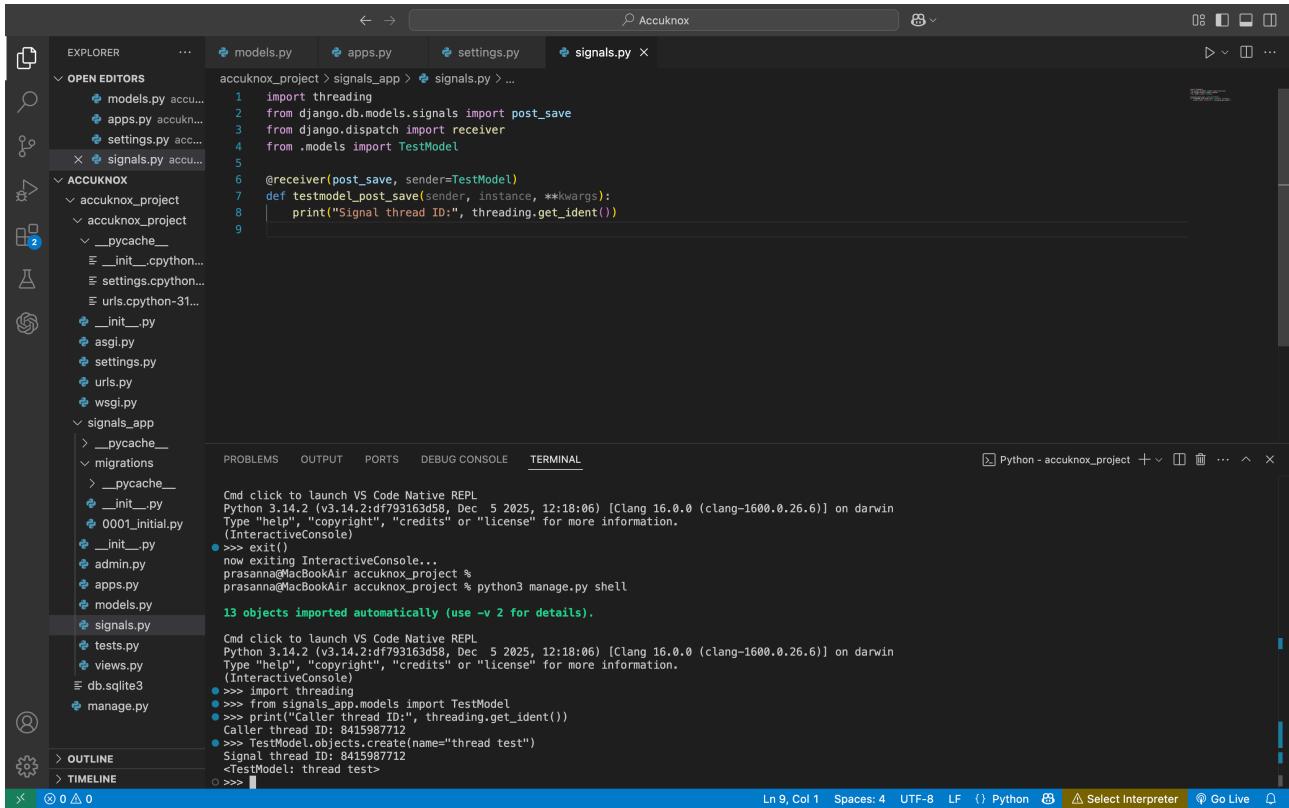
* History restored
prasanna@MacBookAir accuknox_project % python3 manage.py shell
13 objects imported automatically (use -v 2 for details).

Cmd click to launch VS Code Native REPL
Python 3.14.2 (v3.14.2:dd793163d58, 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 the VS Code interface with the following details:

- EXPLORER:** Shows the project structure for 'accuknox\_project'. The 'signals.py' file is open in the editor.
- EDITOR:** The code in 'signals.py' is:import threading
from django.db.models.signals import post\_save
from django.dispatch import receiver
from .models import TestModel

@receiver(post\_save, sender=TestModel)
def testmodel\_post\_save(sender, instance, \*\*kwargs):
 print("Signal thread ID:", threading.get\_ident())
- TERMINAL:** Shows two sessions of the Python interactive shell. The first session shows the import of 'threading' and the definition of the signal receiver. The second session shows the execution of the receiver, printing the thread ID '8415987712'.
- STATUS BAR:** Shows the current file is 'Python - accuknox\_project', line 9, column 1, and the terminal has 4 spaces.

**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.

The screenshot shows a code editor interface with the following details:

- EXPLORER:** Shows the project structure of `accuknox_project` with files like `models.py`, `apps.py`, `settings.py`, and `signals.py`.
- models.py:** Contains code for a `TestModel` and a signal receiver `testmodel_post_save`.
- signals.py:** Contains a signal receiver `@receiver(post_save, sender=TestModel)` that prints a message and raises an exception.
- TERMINAL:** Shows a Django shell session with the following content:
 

```

SyntaxError: invalid syntax
  >>> from django.db import transaction
  >>> from signals_app.models import TestModel
  >>> try:
  ...     with transaction.atomic():
  ...         print("transaction.atomic() called")
  ...         with transaction.atomic():
  ...             print("nested transaction.atomic() called")
  ...             TestModel.objects.create(name="transaction test")
  ...         print("transaction.atomic() finished")
  ...     print("transaction.atomic() finished")
  ... 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: first test, <TestModel: thread test>>
  >>> 
```

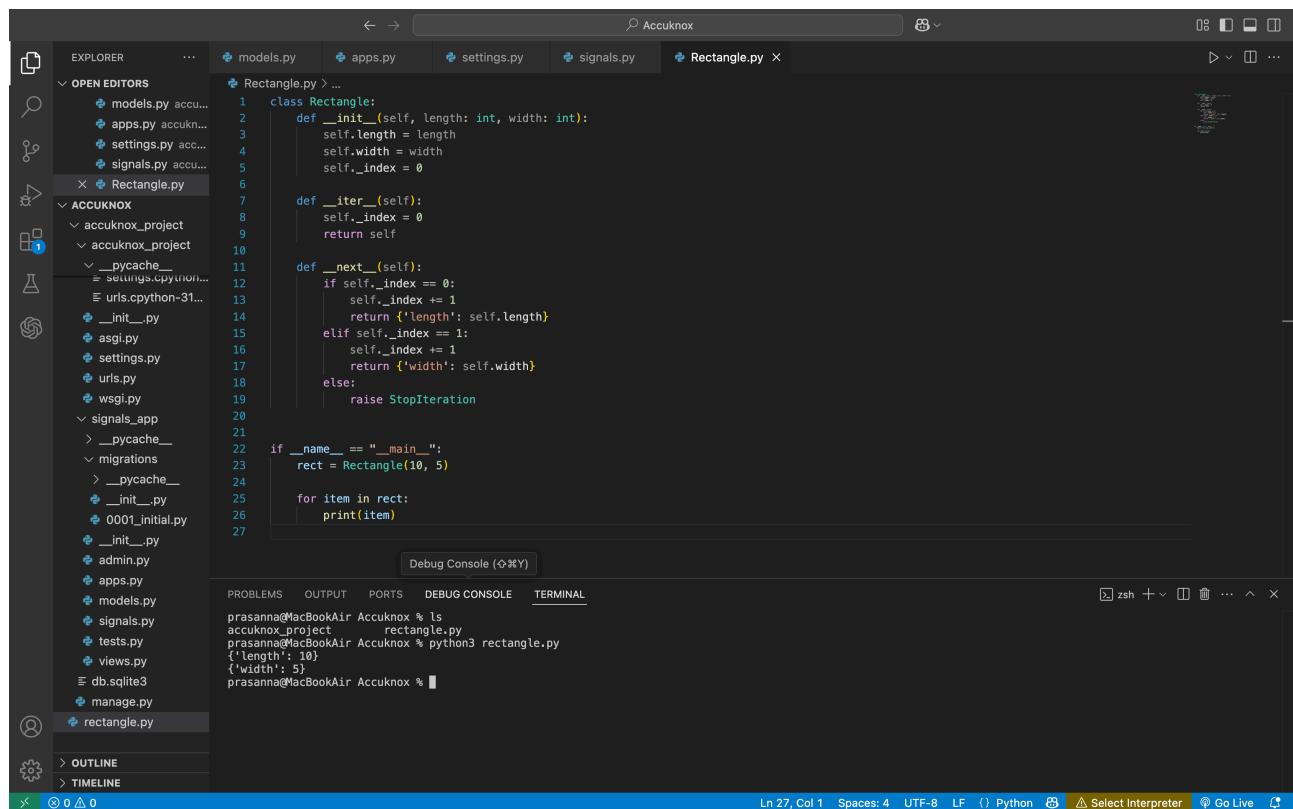
**Answer:** The screenshot shows Django shell execution where a model instance is created inside a `transaction.atomic()` block. During signal execution, an exception is raised intentionally, which causes the transaction to roll back. When querying the database afterward, the newly created object is 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.



The screenshot shows a code editor with the `Rectangle.py` file open. The code defines a `Rectangle` class with `length` and `width` attributes, and implements the iterator protocol with `__iter__` and `__next__` methods. The `__next__` method returns a dictionary with 'length' and 'width' keys. The code is run in a terminal, showing the output of the `rectangle.py` script when run with `python3`, which prints the dictionary representation of the rectangle's dimensions.

```
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
22     if __name__ == "__main__":
23         rect = Rectangle(10, 5)
24
25         for item in rect:
26             print(item)
```

PROBLEMS OUTPUT PORTS DEBUG CONSOLE TERMINAL

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

Ln 27, Col 1 Spaces: 4 UTF-8 LF {} Python Select Interpreter Go Live

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.