

Open in app ↗

Sign up

Sign in

Medium

Search



Unit Testing in Python: A Comprehensive Guide for Beginners



Sachinsoni · Follow

7 min read · Mar 2, 2024



Listen



Share

Unit Testing is a technique in which particular module is tested to check by developer himself whether there are any errors. The primary focus of unit testing is test an individual unit of system to analyze, detect, and fix the errors.

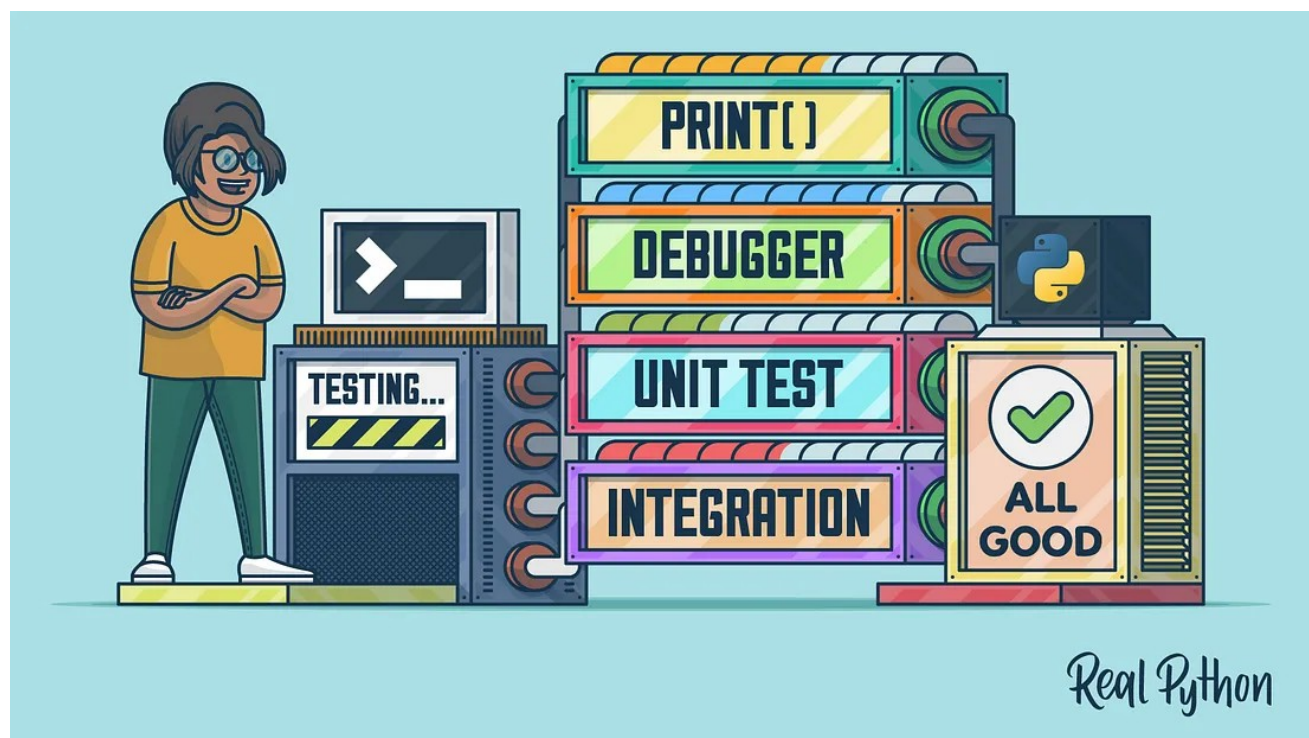


IMAGE by Real Python

Python provides the **unittest module** to test the unit of source code. The unittest plays an essential role when we are writing the huge code, and it provides the facility to check whether the output is correct or not.

Normally, we print the value and match it with the reference output or check the output manually. This process takes lots of time. To overcome this problem, Python introduces the **unittest** module.

What are Test Cases ?

Test cases in unit testing are like **mini tests** for your code. They check individual parts of your code (like functions) to make sure they work as expected with different inputs. This helps **catch bugs early** and keep your code **reliable and working smoothly**.

Types of test cases in unit testing :

There are 2 types of test cases which are :

1. Positive Test Cases

- **Purpose:** Verify that your code functions as expected when provided with valid inputs.
- **Focus:** These tests aim to confirm that the code produces the correct output or exhibits the desired behavior with typical, correct data.

2. Negative Test Cases

- **Purpose:** Test how your code handles invalid inputs or unexpected scenarios. The goal is to ensure the code responds gracefully and produces appropriate error messages or exceptions.
- **Focus:** These tests deliberately try to break your code (in a controlled way). They cover edge cases, boundary values, and incorrect data types.

*Why Both Are Important

Positive and negative test cases work hand-in-hand to provide a comprehensive testing strategy:

- **Positive test cases** ensure the core functionality of your code works under normal conditions.
- **Negative test cases** protect your code against potential errors, making your

application more robust and resilient to unexpected user input or edge cases.

How to write unit test for our Python modules ?

Here's a guide on how to write a test case module in Python:

1. Choose a test framework:

- **unittest** : The built-in Python module, ideal for beginners due to its simplicity.
- **pytest** : A popular framework known for its flexibility, advanced features, and readable style.

2. Structure your module:

Organize your test cases into a clear structure for maintainability:

- **Create a Python file:** Name it `test_{module_name}.py`, where `{module_name}` is the actual module you're testing.
- **Import necessary modules:** Include `unittest` (or your chosen framework) and the module you're testing.
- **Define a test class:** Inherit from `unittest.TestCase` (or its equivalent in your chosen framework) to create a class for your test cases.

3. Write individual test cases:

Within your test class, define separate methods for each test case:

- **Test method names:** Use descriptive names that clearly indicate what's being tested (e.g., `test_add_positive_numbers` or `test_invalid_file_format`).
- **Test case structure:** Follow this common pattern:

i). **Set up test data:** Create any data or objects needed for the test (e.g., sample input values).

ii). **Execute the unit under test:** Call the function, method, or class you're testing using the setup data.

iii). **Assert expected behavior:** Use assertions provided by the framework (e.g., `assertEqual` , `assertTrue`) to verify that the output or behavior matches your expectations.

4. Example using `unittest` :

I have a sample.py module in which some mathematical operations are shown below :

```
class Operation:

    def __init__(self):
        pass

    def add(self,a,b):
        if not isinstance(a, (int, float)):
            raise ValueError("Value must be either an integer or a float.")

        if not isinstance(b, (int, float)):
            raise ValueError("Value must be either an integer or a float.")
        return a+b

    def minus(self,a,b):
        if not isinstance(a, (int, float)):
            raise ValueError("Value must be either an integer or a float.")

        if not isinstance(b, (int, float)):
            raise ValueError("Value must be either an integer or a float.")
        return a-b

    def mul(self,a,b):
        if not isinstance(a, (int, float)):
            raise ValueError("Value must be either an integer or a float.")

        if not isinstance(b, (int, float)):
            raise ValueError("Value must be either an integer or a float.")
        return a*b

    def div(self,a,b):
        if not isinstance(a, (int, float)):
            raise ValueError("Value must be either an integer or a float.")
        if not isinstance(b, (int, float)):
            raise ValueError("Value must be either an integer or a float.")

        if b==0:
            raise ValueError("Zero division error")
```

```
return a/b
```

Now, it's time to write our unit test file for test_sample.py file.

```
import unittest
import sample

class TestSample(unittest.TestCase):

    def __init__(self, methodName='runTest'):
        super().__init__(methodName)

        # Test data
        self.a1 = 20
        self.a2 = 'Ram'
        self.b1 = 10
        self.b2 = 0
        self.checker = sample.Operation() # Creating the object

    def test_add(self):
        # Positive test case
        result = self.checker.add(self.a1, self.b1)
        self.assertEqual(result, 30)

        # Negative test case for data type
        with self.assertRaises(ValueError) as context:
            self.checker.add(self.a2, self.b1)
        self.assertEqual(str(context.exception), "Value must be either an in

    def test_minus(self):
        # Positive test case
        result = self.checker.minus(self.a1, self.b1)
        self.assertEqual(result, 10)

        # Negative test case for data type
        with self.assertRaises(ValueError) as context:
            self.checker.minus(self.a2, self.b1)
        self.assertEqual(str(context.exception), "Value must be either an in

    def test_mul(self):
        # Positive test case
        result = self.checker.mul(self.a1, self.b1)
        self.assertEqual(result, 200)

        # Negative test case for data type
        with self.assertRaises(ValueError) as context:
            self.checker.mul(self.a2, self.b1)
```

```
        self.assertEqual(str(context.exception), "Value must be either an in

def test_div(self):
    # Positive test case
    result = self.checker.div(self.a1, self.b1)
    self.assertEqual(result, 2)

    # Negative test case for data type
    with self.assertRaises(ValueError) as context:
        self.checker.div(self.a2, self.b1)
    self.assertEqual(str(context.exception), "Value must be either an in

    # Negative test case for zero division
    with self.assertRaises(ValueError):
        self.checker.div(self.a1, self.b2)

if __name__ == "__main__":
    unittest.main()
```

and for running the above test script is by the following command :

```
python -m unittest <test_script_name>
```

And, you will get the following output :

```
C:\Users\Sachin\Desktop\tert>python -m unittest test_sample.py
....
-----
Ran 4 tests in 0.001s

OK
```

Terminal output after running test python script

Running Specific Test Methods/Class Using Python's unittest Module :

```
# For functions/Methods
python -m unittest <script_name>.<class_name>.<function_name>
Example -> python -m unittest test_sample.TestSample.test_add

# For Classes
```

```
python -m unittest <script_name>.<class_name>
Example -> python -m unittest test_sample.TestSample
```

Running All Test Scripts in a Specific Folder :

```
# Suppose in testing folder all my test scripts are written
python -m unittest discover -s testing/

# Here's what each part of the command does:

# python: Specifies that you want to run a Python script or module.
# -m unittest: This runs the unittest module as a script.
# discover: This tells the unittest module to automatically discover and run
# -s testing/: This specifies the directory where the test modules are located
```

About python -m unittest :

The `python -m unittest` command is used to run unit tests when your test cases are organized into a package or module structure and you want to run all the tests in that package or module.

Python Basic Functions and Unit Test Output

The unittest module produces three possible outcomes. Below are the potential outcomes.

1. **OK** — If all tests are passed, it will return OK.
2. **Failure** — It will raise an `AssertionError` exception, if any of tests is failed.
3. **Error** — If any errors occur instead of Assertion error.

Let's see the following basic functions.

Method	Description
<code>.assertEqual(a, b)</code>	<code>a == b</code>
<code>.assertTrue(x)</code>	<code>bool(x)</code> is True
<code>.assertFalse(x)</code>	<code>bool(x)</code> is False
<code>.assertIs(a, b)</code>	<code>a</code> is <code>b</code>
<code>.assertIsNone(x)</code>	<code>x</code> is None
<code>.assertIn(a, b)</code>	<code>a</code> in <code>b</code>
<code>.assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>.assertNotIn(a, b)</code>	<code>a</code> not in <code>b</code>
<code>.assertNotIsInstance(a,b)</code>	<code>not isinstance(a, b)</code>
<code>.assertIsNot(a, b)</code>	<code>a</code> is not <code>b</code>

image by javatpoint

So, this is the way through which you can write your test script. For more details regarding unit testing [click here](#) for unit test documentation!

Running Unit Tests with Coverage Analysis

Coverage analysis, an essential part of software testing, serves to identify untested sections of code by revealing which lines remain unexecuted during test runs. By pinpointing these gaps in test coverage, developers gain insights into areas of the codebase that require additional testing attention. This process not only enhances the overall quality of the test suite but also contributes to the early detection of potential bugs or vulnerabilities, ultimately bolstering the reliability and robustness of the software.

In short we can say that coverage analysis tells us which parts of our code are tested by our tests and which parts are not. This helps us make sure we're testing everything we should be testing.

Applying coverage on our unit testing :

```
# First you need to install coverage package by  
pip install coverage
```



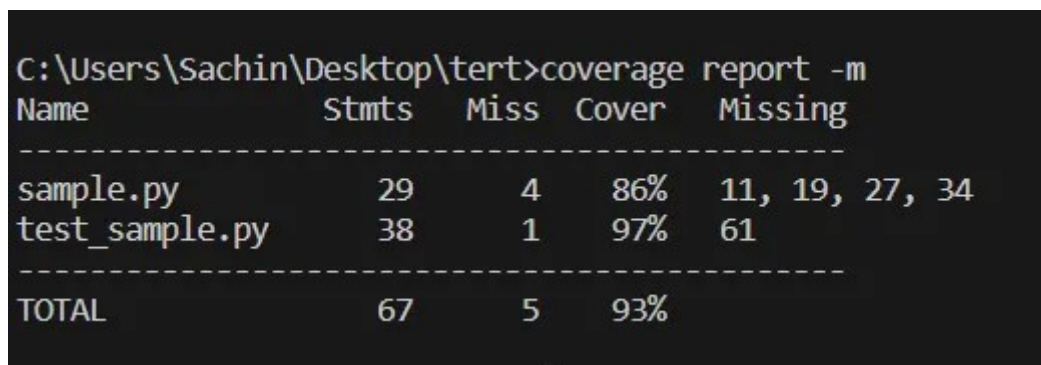
```
# Then run the following command :
coverage run --source=src -m unittest discover testing/

# In this command:

# --source=src specifies that coverage analysis should be applied to code wi
# -m unittest discover -s tests/ runs the unit tests located in the tests di
# By using --source, you can focus coverage analysis on the parts of your
# codebase that you want to evaluate, which is particularly useful in larger
# projects where you may have multiple directories containing source code.
```

```
# Now run this command to generate report of coverage analysis
coverage report -m

# you will get the following output:
```



```
C:\Users\Sachin\Desktop\tert>coverage report -m
Name                Stmts  Miss  Cover   Missing
-----
sample.py           29      4    86%    11, 19, 27, 34
test_sample.py      38      1    97%      61
-----
TOTAL                67      5    93%
```

coverage report

Upon reviewing the report, it is clear that there are no test cases written in the **test_sample.py** file for lines 11, 19, 27, and 34 in the **sample.py** file. This approach helps identify which lines require test cases, guiding developers to write comprehensive tests and enhance their efficiency in software development. You can also generate a xml file for your coverage by the following command :

```
coverage xml

# For html
coverage html
```

If you want to perform entire coverage process automatically, follow these steps :

1. Make a bash file with name coverage.sh or give the name as you want.
2. Inside this file write following commands :

```
coverage run --source=src -m unittest discover testing/  
coverage report -m # this command shows the result at terminal  
coverage html      # generating html file  
coverage xml       # generating xml file
```

3. Now run this bash file by using :

```
bash coverage.sh
```

For more information regarding coverage, [click here](#) for the coverage documentation !

I hope this blog has deepened your understanding of unit testing and coverage concepts. If you've found value in this content, consider following me for more insightful posts. Thank you! for investing your time in reading this article.



Follow

Written by Sachinsoni

581 Followers · 19 Following

Responses (1)



Write a response

What are your thoughts?



Cypress Godwin Adebayo

3 days ago



```
import requests
```

```
import telebot
```

```
import sys
```

```
import subprocess
```

```
# Ensure required modules are installed
```

```
def install_package(package):
```

```
try:
```

```
    __import__(package)
```

```
except ImportError:
```

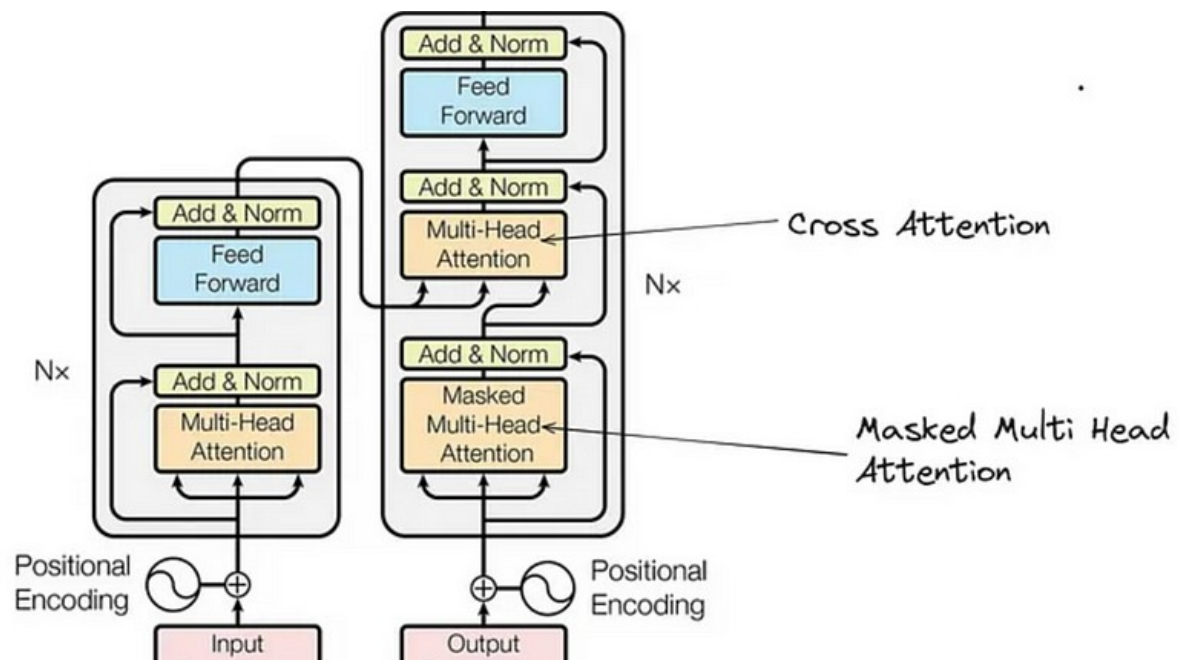
```
    subprocess.check_call([sys.executable, "-m", "pip", "install", package])
```

```
install_pack... more
```



[Reply](#)

More from Sachinsoni

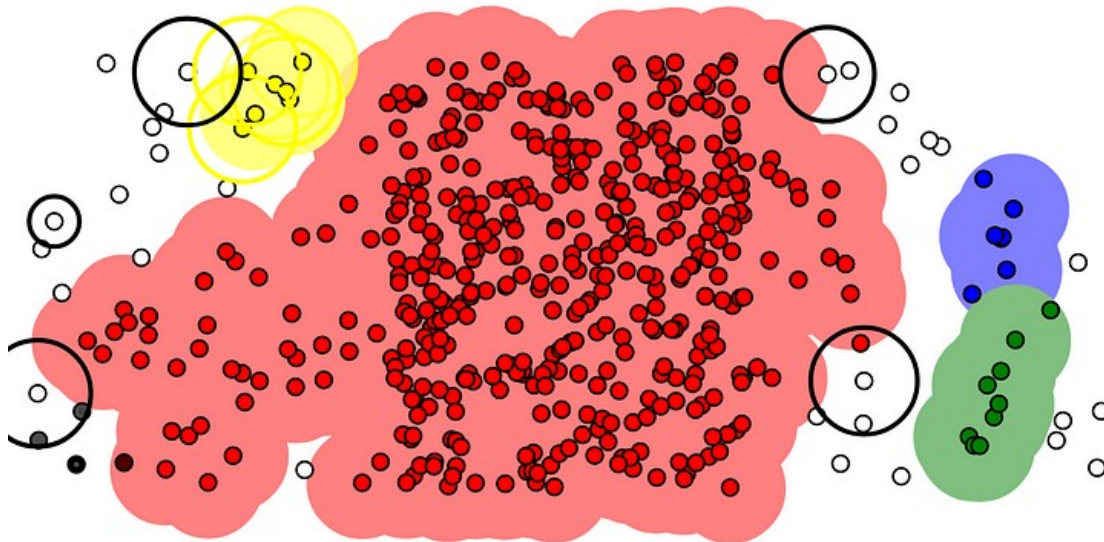


GS Sachinsoni

Cross Attention in Transformer

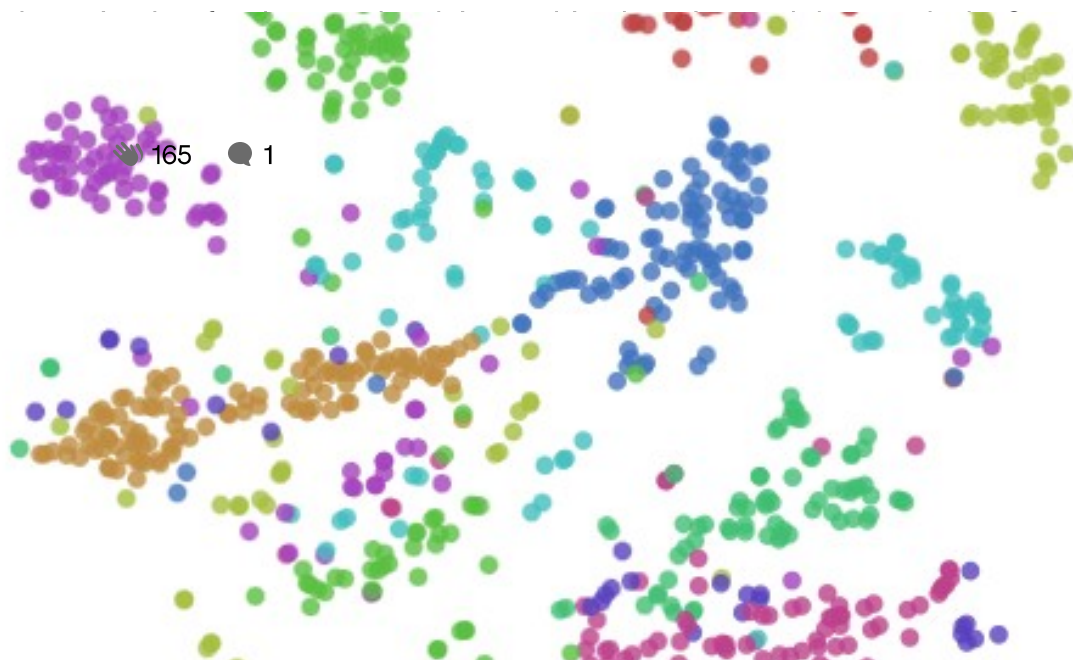
Cross attention is a key component in transformers, where a sequence can attend to another sequence's information, making it essential for...

Sep 6, 2024 45 1



 Sachinsoni

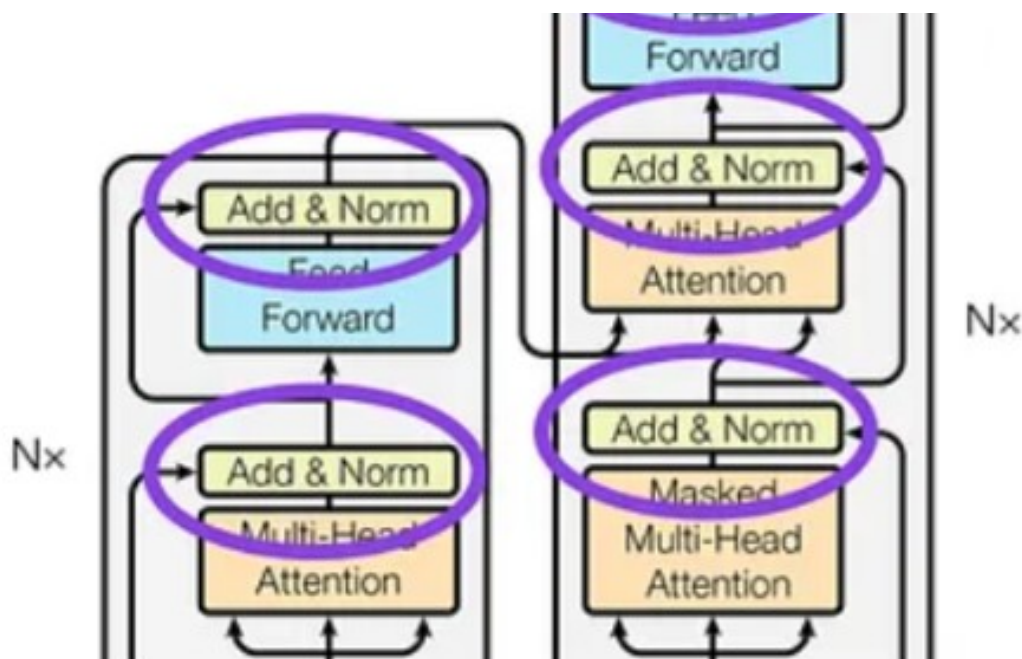
Clustering Like a Pro: A Beginner's Guide to DBSCAN

 Sachinsoni

Mastering t-SNE(t-distributed stochastic neighbor embedding)

A better dimensionality reduction technique as compared to PCA (Principal Component Analysis)


Feb 11, 2024  201  3



 Sachinsoni

Layer Normalization in Transformer

Layer normalization is a crucial technique in transformer models that helps stabilize and accelerate training by normalizing the inputs to...


Sep 4, 2024  87



See all from Sachinsoni

Recommended from Medium



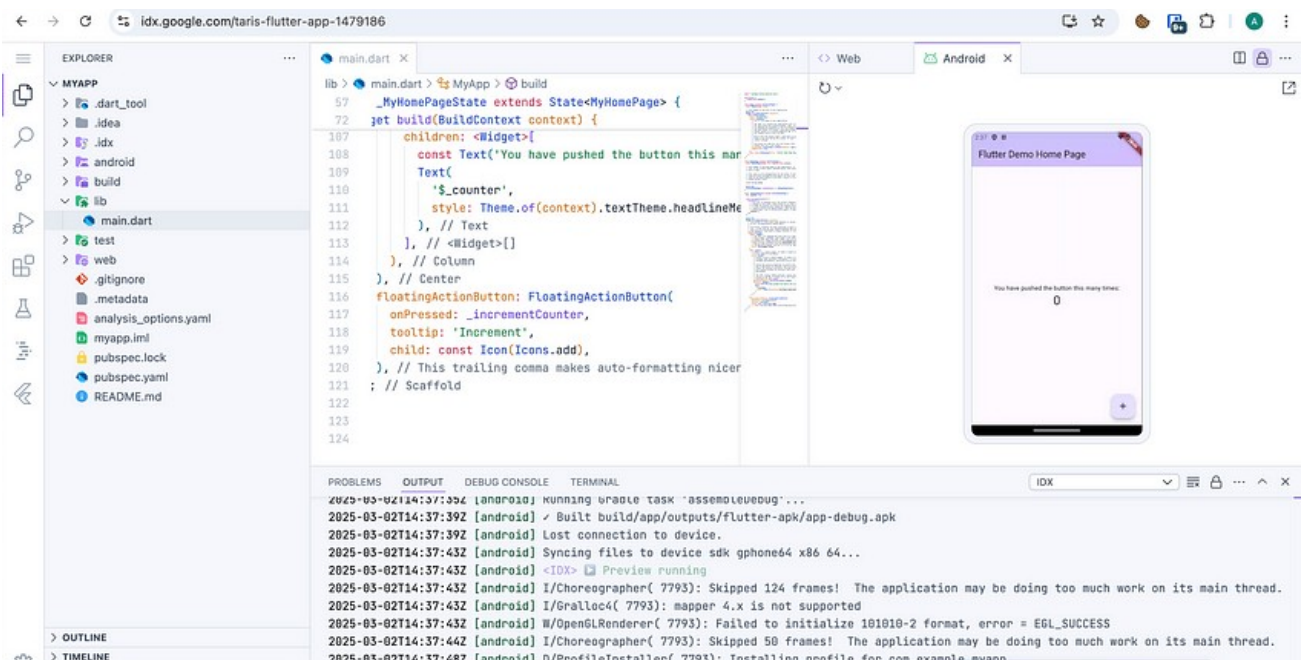
 In Python in Plain English by Dhruv Ahuja


How I Learned to Love `__init__.py`: A Simple Guide 😊

💡 Heads Up! Click here to unlock this article for free if you're not a Medium member!

★ Feb 3  1.5K  13





 In Coding Beauty by Tari Ibaba

This new IDE from Google is an absolute game changer

This new IDE from Google is seriously revolutionary.



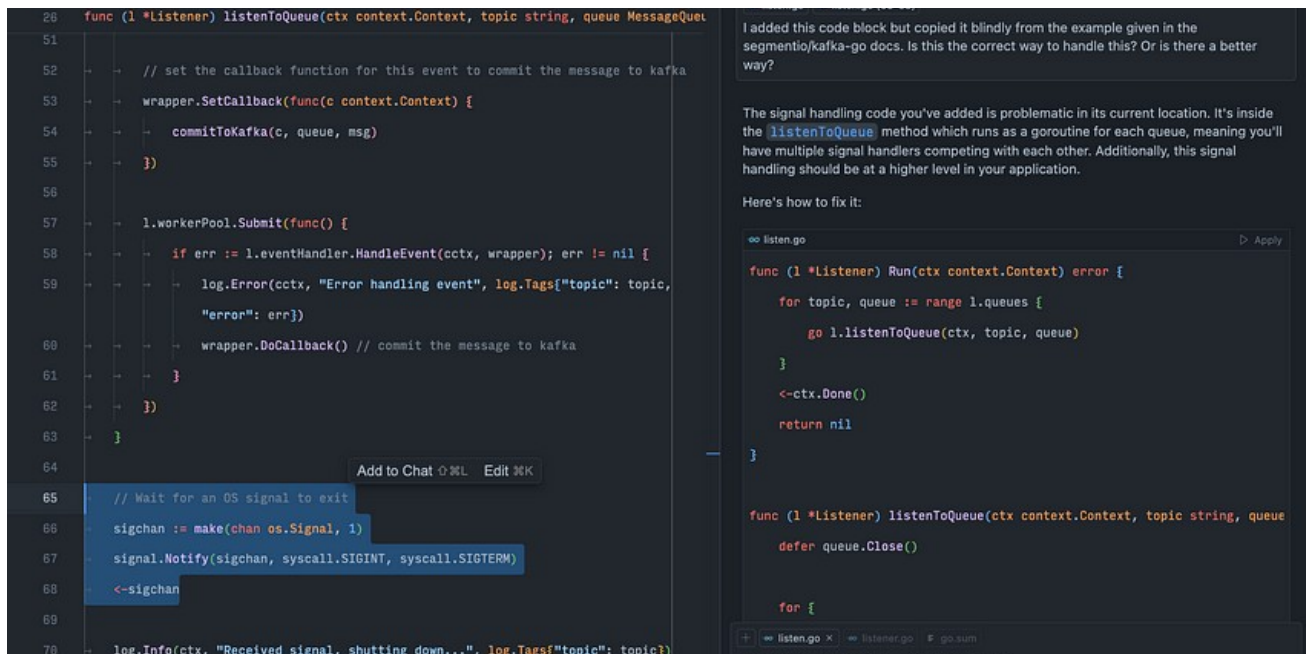
Mar 11



2.5K



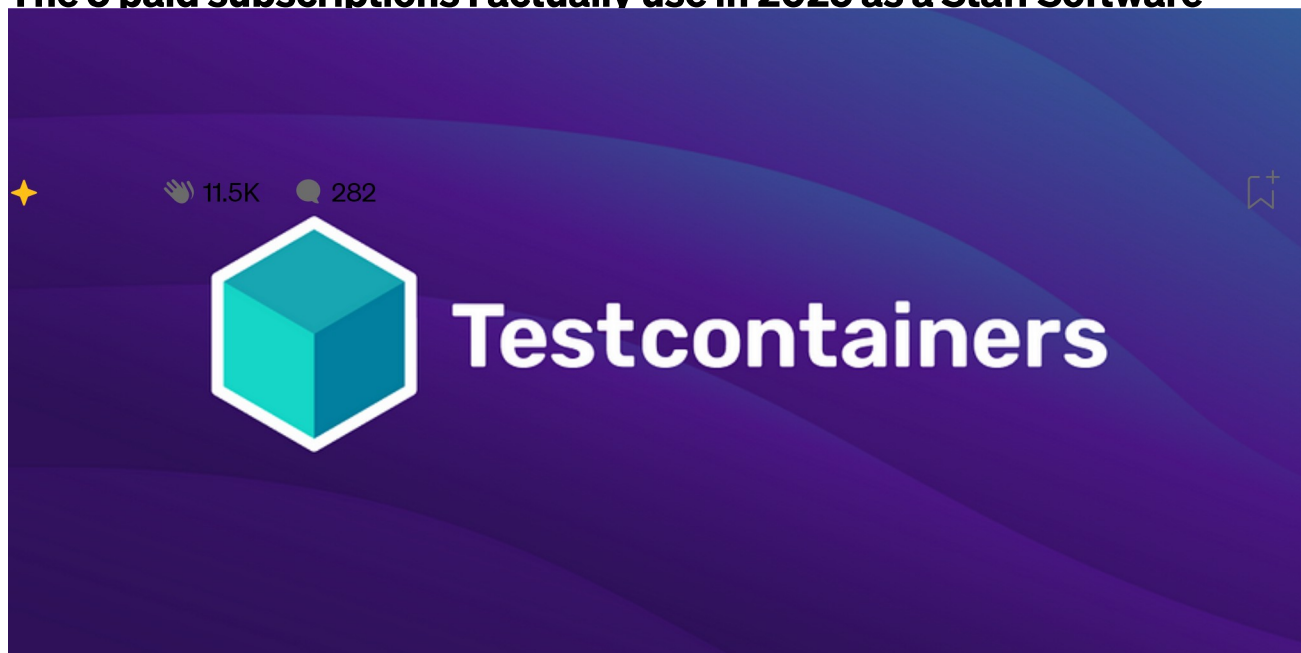
146





In Level Up Coding by Jacob Bennett

The 5 paid subscriptions I actually use in 2025 as a Staff Software



Jesús

Testcontainers in Python

In modern software development, writing tests for applications that interact with external services like databases, message queues, or even...



Oct 12, 2024



5

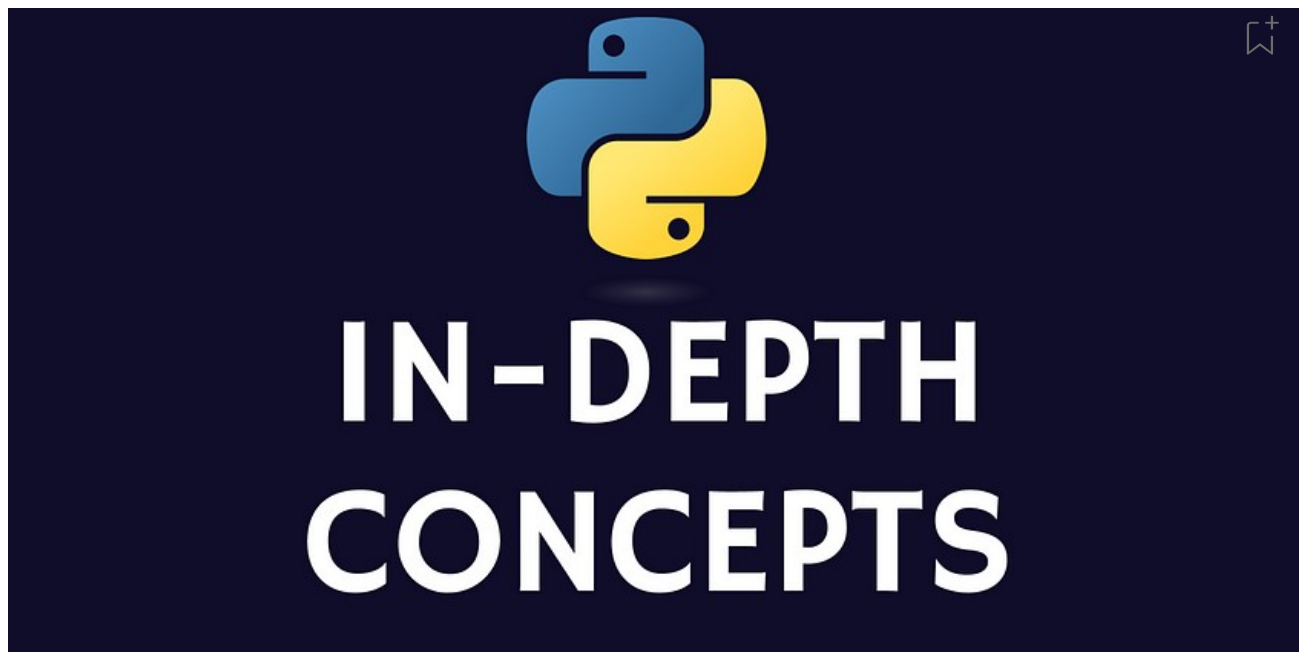




1



Best Practices for Implementing Configuration Class in Python

Managing configurations for development (dev), user acceptance testing (uat), and production (prod) environments is a critical part of...



 Sabrina Carpenter 

If You Can Answer These 7 Concepts Correctly, You're Decent at Python

Perfect for anyone wanting to prove their Python expertise!

★ Feb 3 🖱 931 💬 13



See more recommendations