unittest framework

Testing framework in python

Introduction

- A unit test is a test that operates on an individual unit of software. A unit test aims to validate that the tested unit works as designed.
- A unit is often a small part of a program that takes a few inputs and produces an output. Functions, methods, and other callables are good examples of units that you'd need to test.
- Python standard library has a testing framework named "unittest"
- It is used to write automated tests for your code.
- The unittest package has an object-oriented approach where test cases derived from a base class, which has several useful methods.

Unittest framework

- The framework uses an object-oriented approach and supports some essential concepts that facilitate test creation, organization, preparation, and automation:
  - **Test case:** An individual unit of testing. It examines the output for a given input set.
  - **Test suite:** A collection of test cases, test suites, or both. They're grouped and executed as a whole.
  - **Test fixture:** A group of actions required to set up an environment for testing. It also includes the teardown processes after the tests run.
  - **Test runner:** A component that handles the execution of tests and communicates the results to the user.

Note : the test scripts written in python must start with "test_"

TestCase class

- The unittest package defines the TestCase class, which is primarily designed for writing unit tests.
- To start writing your test cases, you just need to import the class and subclass it.
- Then, you'll add methods whose names should begin with test.
- These methods will test a given unit of code using different inputs and check for the expected results.
- Verify the example beside , to know the usage of TestCase class
    - **abs() :** it returns absolute positive value of an integer
    - .**assertEqual()** method of TestCase class used to verify the values passed to it are equal are not

```
import unittest

class TestAbsFunction(unittest.TestCase):
    def test_positive_number(self):
        self.assertEqual(abs(10), 10)

    def test_negative_number(self):
        self.assertEqual(abs(-10), 10)

    def test_zero(self):
        self.assertEqual(abs(0), 0)
```

Running unittest tests

- Once you've written the tests, you need a way to run them. You'll have at least two standard ways to run tests with unittest:
  - Make the test module executable
    - To make a test module executable in unittest, you can add the following code to the end of the module:
    - The main() function from unittest allows you to load and run a set of tests

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

  - Use the command-line interface of unittest
    - you can run the module as a regular Python script

```
$python3  test_absfunction.py
.........
----------------------------------------------------------------------
Ran 9 tests in 0.000s

OK
```

verbosity argument of main()

- the main() function takes the verbosity argument as one. With this argument, you can tweak the output's verbosity, which has three possible values:
  - 0 for quiet
  - 1 for normal
  - 2 for detailed
- If you want to make the detailed output more descriptive visible in unittest output , then you can add docstrings to your tests like in the following code snippet:

```
def test_upper(self):
        """Test that 'foo' is converted to 'FOO'."""
        self.assertEqual('foo'.upper(), 'FOO')
```

```
vlab@lochu:~/Desktop/unittest_py$ python3 test_string.py
test_upper (__main__.TestStringMethods)
Test that 'foo' is converted to 'FOO'. ... ok

----------------------------------------------------------------------
Ran 1 tests in 0.000s

OK
```

Sample code

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        """Test that 'foo' is converted to 'FOO'."""
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        """Test string case checking."""
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        """Test string splitting."""
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

Skipping tests

- The unittest framework also supports skipping individual test methods and even whole test case classes.
- Skipping tests allows you to temporarily bypass a test case without permanently removing it from your test suite.
- The following decorators will help you with the goal of skipping tests during your test running process:

Decorator	Description
@unittest.skip(reason)	Skips the decorated test
@unittest.skipIf(condition,reason)	Skips the decorated test if condition is true
@unittest.skipUnless(condition,reason)	Skips the decorated test unless condition is true

Example code

```
import sys
import unittest

class SkipTestExample(unittest.TestCase):
    @unittest.skip("Unconditionally skipped test")
    def test_unimportant(self):
        self.fail("The test should be skipped")

    @unittest.skipIf(sys.version_info < (3, 12), "Requires
Python >= 3.12")
    def test_using_calendar_constants(self):
        import calendar

        self.assertEqual(calendar.Month(10),
calendar.OCTOBER)

    @unittest.skipUnless(sys.platform.startswith("win"),
"Requires Windows")
    def test_windows_support(self):
        from ctypes import WinDLL, windll

        self.assertIsInstance(windll.kernel32, WinDLL)

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

vlab@lochu:~/Desktop/unittest_py$ python3.12 skip_test.py
test_unimportant (__main__.SkipTestExample.test_unimportant) ... **skipped**
'Unconditionally skipped test'
test_using_calendar_constants
(__main__.SkipTestExample.test_using_calendar_constants) ... ok
test_windows_support (__main__.SkipTestExample.test_windows_support) ... **skipped**
'Requires Windows'

Ran 3 tests in 0.002s

OK (skipped=2)

subTest() method

- **Case :** to check whether a number is even or odd
- If we want to test with a large input dataset we use .subTest() method

even.py

```
def is_even(number):
    return number % 2 == 0
```

```
import unittest

from even import is_even

class TestIsEven(unittest.TestCase):
    def test_even_number(self):
        for number in [2, 4, 6, -8, -10, -12]:
            with self.subTest(number=number):
                self.assertEqual(is_even(number), True)

    def test_odd_number(self):
        for number in [1, 3, 5, -7, -9, -11]:
            with self.subTest(number=number):
                self.assertEqual(is_even(number), False)

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

Assert methods

- The TestCase class provides a set of assert methods. You can use these methods to check multiple conditions while writing your tests.
- They let you compare single values, such as numbers and Booleans, and collections, such as lists, tuples, dictionaries, and more.

Comparing values

- Comparing the result of a code unit with the expected value is a common way to check whether the unit works okay. The TestCase class defines a rich set of methods that allows you to do this type of check:

Method	Comparison
`.assertEqual(a, b)`	`a == b`
`.assertNotEqual(a, b)`	`a != b`
`.assertTrue(x)`	`bool(x) is True`
`.assertFalse(x)`	`bool(x) is False`

Comparing Objects by Their Identity

- TestCase also implements methods that are related to the identity of objects.
- An object's identity is the memory address where the object lives. This identity is a unique identifier that distinguishes one object from another.
- It is a read-only property, which means that you can't change an object's identity once you've created the object.
- To check an object's identity, you'll use the **is and is not operators.**

Method	Comparison
`.assertIs(a, b)`	`a is b`
`.assertIsNot(a, b)`	`a is not b`
`.assertIsNone(x)`	`x is None`
`.assertIsNotNone(x)`	`x is not None`

Comparing Collections

- Another common need when writing tests is to compare collections, such as lists, tuples, strings, dictionaries, and sets. The TestCase class also has shortcut methods for these types of comparisons
- These methods run equality tests between different collection types.

Method	Comparison
`.assertSequenceEqual(a, b)`	Equality of two sequences
`.assertMultiLineEqual(a, b)`	Equality of two strings
`.assertListEqual(a, b)`	Equality of two lists
`.assertTupleEqual(a, b)`	Equality of two tuples
`.assertDictEqual(a, b)`	Equality of two dictionaries
`.assertSetEqual(a, b)`	Equality of two sets

Running Membership Tests

- A membership test is a check that allows you to determine whether a given value is or is not in a collection of values. You'll run these tests with the in and not in operators

Method	Check
`.assertIn(a, b)`	`a in b`
`.assertNotIn(a, b)`	`a not in b`

Checking for an Object's Type

- Checking the type of the object that a function, method, or callable returns may be another common requirement in testing.

Method	Comparison
`.assertIsInstance(a, b)`	`isinstance(a, b)`
`.assertNotIsInstance(a, b)`	`not isinstance(a, b)`

- These two methods are based on the built-in isinstance() function, which you can use to check whether the input object is of a given type.

Testing for exceptions

- Sometimes, you'll need to check for exceptions. Yes, sometimes your own code will raise exceptions as part of its behavior.

Method	Check
`.assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` raises `exc`
`.assertRaisesRegex(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises `exc` and the message matches regex `r`

- The first method allows checking for explicit exceptions without considering the associated error message, and the second method checks for exceptions and considers the associated message using regular expressions.

Contd...

- The TestCase class also provides some additional assert methods that help you with warnings and logs:

Method	Check
`.assertWarns(warn, fun, *args, **kwds)`	`fun(*args, **kwds)` raises `warn`
`.assertWarnsRegex(warn, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises `warn` and the message matches regex `r`
`.assertLogs(logger, level)`	The `with` block logs on `logger` with minimum `level`
`.assertNoLogs(logger, level)`	The `with` block does not log on `logger` with minimum `level`

Using unittest From the Command Line

- The unittest package also provides a command-line interface (CLI) that you can use to discover and run your tests.
- With this interface, you can run tests from modules, classes, and even individual test methods.
    - **$ python -m unittest test_module1 test_module2**
    - **$ python -m unittest test_module.TestCase**
    - **$ python -m unittest test_module.TestCase.test_method**

Discovering Tests Automatically

- The unittest framework supports test discovery. The test loader can inspect each module in a given directory looking for classes derived from TestCase.
    - $ python -m unittest discover
- This command locates all tests in the current directory, groups them in a test suite, and finally runs them. You can use the python -m unittest as a shortcut for the above command.
- You can use the -s or --start-directory command-line options with the discover subcommand to specify the directory where your tests reside. Other command-line options of discover include:

Option	Description
`-v, --verbose`	Produces a verbose output
`-p, --pattern`	Allows for using glob patterns and defaults to `test*.py`
`-t, --top-level-directory`	Defines the top-level directory of a project

Grouping Your Tests With the TestSuite Class

- The unittest framework has a class called TestSuite that you can use to create groups of tests and run them selectively. Test suites can be useful in many situations, including the following:
  - **Complex projects:** In complex projects with many features, test suites help you organize tests into manageable and logical groups.
  - **Different testing levels:** Test suites allow you to organize your tests according to their testing levels, including unit tests, integration tests, and system tests.
  - **Selective testing:** Test suites allow you to create logical groups of tests that you can run selectively, saving time and resources.
  - **Environment-specific testing:** Test suites allow group tests that are supposed to run on specific platforms, such as Windows, Linux, macOS, or others

Example code

calci.py

```
import math
from collections import Counter

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def mean(data):
    return sum(data) / len(data)

def median(data):
    n = len(data)
    index = n // 2
    if n % 2:
        return sorted(data)[index]
    return sum(sorted(data)[index - 1 : index + 1]) / 2
```

test_calci.py

```
import unittest

from calculations import (
    add,
    mean,
    median,
    subtract,
)

class TestArithmeticOperations(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(10, 5), 15)
        self.assertEqual(add(-1, 1), 0)

    def test_subtract(self):
        self.assertEqual(subtract(10, 5), 5)
        self.assertEqual(subtract(-1, 1), -2)
class TestStatisticalOperations(unittest.TestCase):
    def test_mean(self):
        self.assertEqual(mean([1, 2, 3, 4, 5, 6]), 3.5)

    def test_median(self):
        self.assertEqual(median([1, 3, 3, 6, 7, 8, 9]), 6)
```

Usage of TestSuite and TextTestRunner

- Suppose you need a way to run the arithmetic and statistical tests separately. In this case, you can create test suites.
- The TestSuite class allows you to create test suites.
  - This class constructor takes the tests argument that must be an iterable of tests or other test suites.
  - manage and run related tests as a single unit
  - organizing large test sets or running specific subsets of tests.
- The “TextTestRunner” is a runner class that executes tests in testsuite
  - It outputs results to the console in a formatted way, showing test successes, failures, and errors.

Creation of test suite : .run() , .addTest() methods

- Create and return a test suite using the **TestSuite()** constructor with the list of tests as an argument.
- To run the suite, you create a **TextTestRunner** and pass the suite to its .**run()** method.

```
def arithmetic_suite():
    arithmetic_tests = [
        TestArithmeticOperations("test_add"),
        TestArithmeticOperations("test_subtract"),
    ]

return unittest.TestSuite(tests=arithmetic_tests)

if __name__ == "__main__":
    suite1 = make_suite()
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite1)
```

(or)

- Use the **.addTest()** method to add individual tests to an existing suite. To do this, you can do something like the following:

```
def arithmetic_suite():
    arithmetic_suite = unittest.TestSuite()

arithmetic_suite.addTest(TestArithmeticOperations(
"test_add"))

arithmetic_suite.addTest(TestArithmeticOperations(
"test_subtract"))

    return arithmetic_suite

if __name__ == "__main__":
    suite1 = arithmetic_suite()
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite1)
```

.addTests() method

- The TestSuite class also has a **.addTests()** method that you can use to add several tests in one go.
- This method takes an iterable of test cases, test suites, or a combination of them.

```
def statistical_suite():
    statistical_tests = [
        TestStatisticalOperations("test_mean"),
        TestStatisticalOperations("test_median"),
    ]
    statistical_suite = unittest.TestSuite()
    statistical_suite.addTests(statistical_tests)

    return statistical_suite

if __name__ == "__main__":
    suite2 = statistical_suite()
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite2)
```

```
vlab@lochu:~/Desktop/unittest_py$ python3.12 test_calci.py
test_add (__main__.TestArithmeticOperations.test_add) ... ok
test_subtract (__main__.TestArithmeticOperations.test_subtract) ... ok
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
test_mean (__main__.TestStatisticalOperations.test_mean) ... ok
test_median(__main__.TestStatisticalOperations.test_median) ... ok
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Creating Suites With the load_tests() Function

- Adding tests to a suite manually can be a big task , also be error-prone and represent a maintenance burden.
- Therefore , unittest has other tools that can help you create test suites quickly.
- The load_tests() function is one of these tools. The function is a hook that unittest provides for customizing test loading and suite creation, either for modules or packages of tests.
- The function takes three mandatory arguments. Here's the signature:
    - **def load_tests(loader, standard_tests, pattern)**

Contd...

- The load_tests() function defined in the module gets called automatically, and unittest takes care of passing in the required arguments.
- The arguments of load_tests() are :
  - **loader:** A TestLoader instance, which helps discover and load test cases.
  - **standard_tests:** Preloaded tests from the module or package.
  - **pattern:** A glob pattern that selects specific tests.

```
def load_tests(loader, standard_tests, pattern):
    suite = unittest.TestSuite()
    suite.addTests(loader.loadTestsFromTestCase(TestArithmeticOperations))
    suite.addTests(loader.loadTestsFromTestCase(TestStatisticalOperations))
     (or)
    suite.addTests(loader.standard_tests)
    return suite

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

Test Fixtures

- A test fixture is a preparation that you perform before and after running one or more tests.
- The preparations before the test run are known as setup, while the tasks that you perform after the test run are called teardown.
- The setup process may involve the creation of temporary files, objects, databases, dataframes, network connections, and so on.
- In contrast, the teardown phase may require releasing resources, removing temporary files, closing connections, and similar tasks.

Setup and teardown fixtures

- The unittest framework allows you to create setup and teardown fixtures in your test cases classes by overriding the following methods in your TestClass subclasses:

Method	Description
`.setUp()`	An instance method that `unittest` calls before running each test method in a test case class.
`.tearDown()`	An instance method that `unittest` calls after running each test method in a test case class.
`.setUpClass()`	A class method that `unittest` calls before running the tests in a test case class.
`.tearDownClass()`	A class method that `unittest` calls after running the tests in a test case class.

- The last two methods are class methods, which means that you need to use the @classmethod decorator to create them

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        ...

    @classmethod
    def tearDownClass(cls):
        ...
```

- These methods only take the current test class as an argument. Remember that they run only once per class

Class-Level Fixtures

- If you use the .**setUpClass() and .tearDownClass()** class methods, then you can create class-level fixtures.
- This type of fixture only **runs once per test case class**.
- The .setUpClass() method runs before the test methods, and .tearDownClass() runs after all the test methods have run
- This behavior is known as **shared fixtures** because all the test methods depend on a single setup and teardown run.
- Note that shared fixtures break test isolation. In other words, the results of a test will depend on previously run tests. So, they should be used with care.

Module-level fixtures

- These fixtures **run once per module**. The setup fixture runs before all the test cases in the module, and the teardown fixture runs after all the test cases in the module have run.
- If an exception occurs in the **setUpModule()** function, then none of the tests in the module run, and the **tearDownModule()** function won't run either.
- If the raised exception is a **SkipTest** exception, then the module will be reported as skipped instead of an error.
- To create module-level fixtures, you need to use module-level functions rather than methods on a TestCase subclass. The required functions are the following:

Function	Description
`setUpModule()`	Runs before all test cases in the containing module
`tearDownModule()`	Runs after all test cases have run

Testing With Fake Objects: unittest.mock

What Is Mock?

- Mock is a general-purpose class in unittest.mock that lets you create a "fake" object. You can configure this fake object to mimic a real object by defining attributes like .return_value, methods, or properties that it should expose.

How Does Mock Work?

- A Mock object has the following capabilities:
    1. **Can Replace Real Objects :** You can use it in place of a real object (like a function, method, or class) to simulate its behavior.
    2. **Tracks Calls :** A Mock object keeps track of how it was called (e.g., arguments passed, number of calls, etc.). This is useful for verifying behavior during a test.
    3. **Configurable Behavior :** You can specify what the Mock should do when called, such as:
        - Returning a specific value with .return_value
        - Raising exceptions with .side_effect
        - Simulating methods or attributes

Use of unittest.mock

1. Basic Mock

```
from unittest.mock import Mock

# Create a mock object
mock_func = Mock()

# Set the mock's return value
mock_func.return_value = "Hello,
Mock!"

# Call the mock as if it were a
function
result = mock_func()

# Output
print(result)  # Output: "Hello,
Mock!"
```

2. Mocking a method

```
from unittest.mock import Mock

# Create a mock object
mock_func = Mock()

# Set the mock's return value
mock_func.return_value = "Hello,
Mock!"

# Call the mock as if it were a
function
result = mock_func()

# Output
print(result)  # Output: "Hello,
Mock!"
```

3. Mocking a attribute

```
from unittest.mock import Mock

# Create a mock object
mock_func = Mock()

# Set the mock's return value
mock_func.return_value = "Hello,
Mock!"

# Call the mock as if it were a
function
result = mock_func()

# Output
print(result)  # Output: "Hello,
Mock!"
```

Returning a Specific Value with .return_value

- You can use the **.return_value** property to specify what a mock should return when called.

```
from unittest.mock import Mock

# Create a mock object
mock_function = Mock()

# Configure the mock to return a specific value
mock_function.return_value = "Hello, Mock!"

# Call the mock function
result = mock_function()

# Output
print(result)  # Output: "Hello, Mock!"
```

Raising Exceptions with .side_effect

- Use .side_effect to make the mock raise an exception or return different results.

1: Raising an Exception

```
from unittest.mock import Mock

# Create a mock object
mock_function = Mock()

# Configure the mock to raise an exception
mock_function.side_effect = ValueError("An error
occurred!")

# Call the mock function (will raise the
exception)
try:
    mock_function()
except ValueError as e:
    print(e)  # Output: "An error occurred!"
```

2: Returning Different Values Sequentially

```
from unittest.mock import Mock

# Create a mock object
mock_function = Mock()

# Configure the mock to return different values
on each call
mock_function.side_effect = [1, 2, 3]

# Call the mock function multiple times
print(mock_function())  # Output: 1
print(mock_function())  # Output: 2
print(mock_function())  # Output: 3
```

Tracking Calls using unittest.mock

- Mocks keep track of how they are called. You can verify calls during testing.
  - **call_count** tells you how many times the mock was called.
  - **call_args_list** keeps track of all calls and their arguments.

```
from unittest.mock import Mock

# Create a mock object
mock_function = Mock()

# Call the mock multiple times
mock_function(1, 2)
mock_function(3, 4)

# Check the number of calls
print(mock_function.call_count)  # Output: 2

# Check the arguments passed to the first call
print(mock_function.call_args_list[0])  # Output: call(1, 2)

# Check all call arguments
print(mock_function.call_args_list)  # Output: [call(1, 2), call(3, 4)]
```

References:

- https://github.com/lochu-55/Company_training/tree/main/unittest_py
- https://realpython.com/python-unittest/#grouping-your-tests-with-the-testsuite-class

THANK YOU