Python Tutorial: Unit Testing Your Code with the unittest Module

Video

https://youtu.be/6tNS--WetLI?si=IHFM6ad8Tf5vlojB

Notes

- You first need to create a new module (file) to store the unit tests
    - The naming convention is "test_<name_of_module_to_be_tested>.py"
        - For example, "test_calc.py"
- The unittest module is built-in, so doesn't need to be installed
    - At the top of the file write `import unittest`
- Next, import the thing you want to test
    - `import calc`
    - `from employee import Employee`
- Create a test class
    - It should inherit from `unittest.TestCase`
    - For example, `class TestCalc(unittest.TestCase):`
- The methods should start with `test_` followed by the name of the function, in the module, to be tested.
    - For example, `def test_add(self):`

- unittest uses assertion statements to determine if a test passed or failed

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

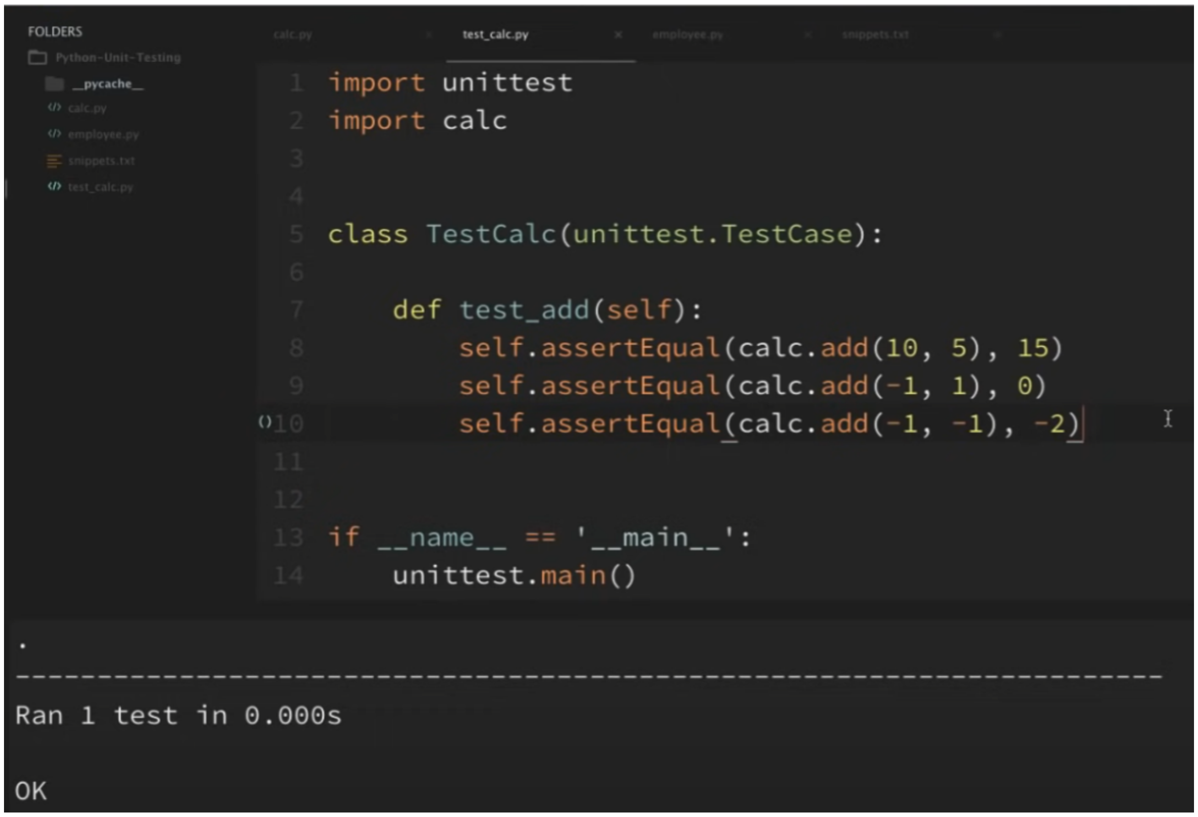
- Create some tests

```
class TestCalc(unittest.TestCase):
  def test_add(self):
      self.assertEqual(calc.add(10, 5), 15)
      self.assertEqual(calc.add(-1, 1), 0)
      self.assertEqual(calc.add(-1, -1), -2)
```

- The tests can be run from the command line
  - To do this you need to run unittest as the main module and pass in your test module
  - For example, `python -m unittest test_calc.py`
- To run it from the editor or with just `python test_calc.py` (from the command line)
  - Add this to the bottom of the file

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

- If you have multiple assertion statements within one test method, it will count as one test in the output



```
import unittest
import calc


class TestCalc(unittest.TestCase):

    def test_add(self):
        self.assertEqual(calc.add(10, 5), 15)
        self.assertEqual(calc.add(-1, 1), 0)
        self.assertEqual(calc.add(-1, -1), -2)


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

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

- You can test that a function raises a particular exception using `assertRaises`
  - For example, `self.assertRaises(ValueError, calc.divide, 10, 0)`
  - The value for `calc.divide` must be passed separately by default
  - If you want to call it ‘normally’ within the test module, you need to use context managers

```
with self.assertRaises(ValueError):
    calc.divide(10, 0)
```

- The tests do not run in the order that they appear in the file, so do not assume they do when creating the tests
    - Tests should be isolated from each other

setUp & tearDow

- These two methods allow you to do things before and after each test. This is useful for writing DRY tests/code to improve maintainability.

```
def setUp(self):
        print('setUp')
        self.emp_1 = Employee('Corey', 'Schafer', 50000)
        self.emp_2 = Employee('Sue', 'Smith', 60000)

    def tearDown(self):
        print('tearDown\n')
```

- Create a `setUp` and `tearDown` method at the top of the test class
    - `setUp` will run before every single test
    - `tearDown` will run after every single test
- If you are testing a class, create the instances of the class within `setUp` as instance attributes of your test class so that you do not repeat yourself in each individual test by creating the instances of the class to be tested inside the test methods.
    - In the tests where you require an instance of the class, you can call the instance attribute, such as `self.emp_1`
- `tearDown` could be used for tests that require the creation/modification of any files. It could delete them or revert them for you automatically

setUpClass & tearDownClass (Class Methods)

- These allow you to run code before and after EVERYTHING rather than just before and after each individual test

```
@classmethod
    def setUpClass(cls):
        print('setupClass')

    @classmethod
    def tearDownClass(cls):
        print('teardownClass')
```

- Could be used to create a database to be used across each test

Mock

- If you have a function that, for example, relies on retrieving some information from a website but the website is down, that function will fail, which will also make your test fail
    - However, tests should only fail if there is something wrong with your code, not if a website out of your control is down
    - You can get around this with “mocking”
- At the top of the test file (after `import unittest`, write `from unittest.mock import patch`
- `patch` can be used as a decorator or context manager
- Mock allows you to create ‘mock objects’ to substitute in, for your tests

```
def test_monthly_schedule(self):
        with patch('employee.requests.get') as mocked_get:
            mocked_get.return_value.ok = True
            mocked_get.return_value.text = 'Success'
```

- In the above code, the context manager is there to get a mock of what could be returned when the `requests.get` is called in the `employee` module. Rather than relying on the website being up and returning ‘ok’ and some text, we can create a mock return value so that if the website is down, our tests will not be affected (which is important as the test should not fail just because a website, that is out of our control, is down)

Resources

- unittest - https://docs.python.org/3/library/unittest.html#unittest.TestCase.debug
- Repo From Video - https://github.com/CoreyMSchafer/code_snippets/tree/master/Python-Unit-Testing