Unit testing
или
модульное тестирование

Основной идеей модульного тестирования является проверка корректности работы модулей или функций проекта по отдельности.

Особенности такого подхода к тестированию:

- \+ автоматизация процесса тестирования
- \+ можно быстро локализовать и поправить ошибку
- \+ уверенность в работоспособности проекта, при хорошем покрытии тестами
- \+ хорошо написанные тесты могут служить своеобразной документацией
- \- в больших проектах объем кода тестов превышает код самого проекта
- \- невозможность проверки работы функций которые не имеют точного результата работы или имеют огромное пространство параметров
- \- тяжело тестировать работоспособность проекта под нагрузкой

Важно понимать, что модульное тестирование,как правило, не исключает необходимость в тестировании проекта в около боевых условиях. Всегда можно что-то забыть, полного покрытия тестами добиться зачастую очень сложно.

Перед написанием непосредственно тестов нужно понять следующие вещи:

- расположение тестов:
  - рядом с тестируемыми функциями
  - в отдельном модуле
  - возможно стоит заводить отдельные папки для хранения тестовых модулей
- фреймворк для тестирования
- общие стили написание тестов

Пример очень простого(бесполезного) теста:

```
import unittest

def inc(a):
    return a + 1

class Test(unittest.TestCase):
    def test_inc(self):
        self.assertEqual(inc(7), 8)
        self.assertEqual(inc(-10), -9)


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

Результат работы теста выглядит так:

```
zaber@zaber-PC:~/Документы/utests$ python3 test1.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Если с простым кодом особых проблем при написании тестов не возникает, то что же делать когда тестируемая функция что то тянет снаружи? Например обращается к БД или сетевым ресурсам, использует другие функции или модули?

В таких случаях используют подделки. Их разделяют на два вида стабы(stubs) и моки(mocks). Из себя они представляют этот «внешний» класс/функцию внутри которых либо возвращается статичное значение подобранное под тест, либо другие нехитрые вещи, все ограниченно только вашей фантазией.

Простой пример, функция для удаления файла:

```
import os

def rm(filename):
    os.remove(filename)
```

И тест для нее с использованием mock, без реального удаления файла:

```
import unittest
import mock

from filerm import rm

class Test(unittest.TestCase):

    @mock.patch('filerm.os')
    def test_rm(self, mock_os):
        rm("test_file")
        mock_os.remove.assert_called_with("test_file")

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

Следующий логичный вопрос: нужно запускать все тестовые файлы руками?

Конечно ответ — нет. Для python3 я нашел такую утилиту, как nose. Она сама ищет тестовые файлы по заданному шаблону, запускает их и агрегирует результаты тестов. Так же там есть поддержка разных фич например расчет покрытия кода тестами, но это другая история.

```
zaber@zaber-PC:~/Документы/utests$ nosetests3 test* -v
test_inc (test1.Test) ... ok
test_rm (test2.Test) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
```