Государственное профессиональное образовательное учреждение
Тульской области «Донской политехнический колледж»
Задания для выполнения самостоятельной работы

студентами отделения 10.02.05 Обеспечение информационной безопасности автоматизированных систем
по *ОП 03 Основы алгоритмизации и программирования*

Практическая работа №1

Тема: Тестирование с использованием unittest.

Задание: Провести тестирование с использованием стандартной библиотеки `unittest`.

Имеется Python-файл с двумя функциями (рисунок 1.1):

```
# файл calc.py

def add(x, y):

    return x + y



def is_positive(x):

    return x > 0
```

Рисунок 1.1 – Python-файл с двумя функциями

Требуется написать тесты для этого файла (модуля), который проверит правильность разработанных функций.

Для этого создадим файл tests.py, в котором будет класс с методом для тестирования. Важно, чтобы все методы с тестами начинались с `test_<что-то>`, иначе Python не поймет, что тестировать. После наследования от

класса `TestCase` из `unittest` будут доступны методы, которые проверяют на соответствие ожидаемому результату.

Напишем несколько проверок для функций вышеприведённого модуля (рисунок 1.2):

```
# файл tests.py

import unittest

import calc # тестируемый модуль



class TestCalc(unittest.TestCase):

    # начинается с test_

    def test_add(self):

        self.assertEqual(calc.add(3, 6), 9)



    # начинается с test_

    def test_is_positive(self):

        self.assertTrue(calc.is_positive(1))
```

Рисунок 1.2 – Проверка для функций

Здесь два теста: один проверяет на равенство, другой на истину. Кроме того, имеются и другие виды проверок, которые показаны на рисунке ниже

(рисунок 1.3). Так, например, для сравнений чисел с плавающей точкой (float) рекомендуется использовать `assertAlomstEqual`.

Method	Checks that	New in
`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`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Рисунок 1.3 – Список методов для проверки на ожидаемое соответствие

Запуск unit-тестов

Для запуска тестов нужно написать в командой строке следующее (рисунок 1.4):

```
$ python -m unittest tests.py
```

Рисунок 1.4 – Запуск тестов

На экране выведется сообщение о проведении тестов и их статус. Каждый пройденный тест обозначается точкой `.`, а проваленной буквой `F`. В нашем случае получили две точки и статус `OK` (рисунок 1.5):

```
Ran 2 tests in 0.000s

OK
```

Рисунок 1.5 – Результат и статус о проведении теста

Кроме того, чтобы не прописывать всю вышеприведённую строчку, можно добавить в Python-файл с тестами вызов функции `unittest.main` в конце в блоке `__main__` (рисунок 1.6):

```
# файл tests.py

class TestCalc(unittest.TestCase)

  # остальной код



if __name__ == "__main__":

    unittest.main()
```

Рисунок 1.6 – Вызов функции unittest.main

Тогда запуск осуществляется простой командой (рисунок 1.7):

```
$ python tests.py
```

Рисунок 1.7 – Запуск простой команды

Проваленные тесты

Попробуем заменить некоторые значения так, что ожидаемый результат не будет сходиться с вычислениями, например, изменим тест с проверкой на положительное число :(рисунок 1.8)

```
def test_is_positive(self):
```

```
self.assertTrue(calc.is_positive(-1))
```

Рисунок 1.8 – Тест с проверкой на положительное число

Ниже показано сообщение. В результате мы провалили один тест и получили `.F` и статус `Failed` (рисунок 1.9).

```
.F

======================================================================

FAIL: test_is_positive (__main__.TestCalc)

----------------------------------------------------------------------

Traceback (most recent call last):

  File "tests.py", line 10, in test_is_positive

    self.assertTrue(calc.is_positive(-1))

AssertionError: False is not true

----------------------------------------------------------------------

Ran 2 tests in 0.001s

FAILED (failures=1)
```

Рисунок 1.9 – Проваленный тест

Если мы заменим в первом методе с проверкой на равенство ожидаемый результат на другое число, то провалим оба тести и получим `FF` (рисунок 1.10).

```
def test_add(self):

    self.assertEqual(calc.add(3, 6), 10)

FF

======================================================================

FAIL: test_add (__main__.TestCalc)

----------------------------------------------------------------------

AssertionError: 9 != 10

======================================================================

FAIL: test_is_positive (__main__.TestCalc)

----------------------------------------------------------------------

AssertionError: False is not true

FAILED (failures=2)
```

Рисунок 1.10 - Замена в первом методе с проверкой на равенство ожидаемый результат на другое число

Тестирование исключений

На практике может возникнуть ситуация, когда функция содержит исключение, срабатываемое при определенных условиях. Их тоже можно проверять, ведь, если функция Python поднимает исключение при ожидаемом результате, значит, она работает корректно.

Добавим в файл с вычислениями ещё одну функцию деления одного числа на другое. Понятно, что делить на 0 невозможно, поэтому функция поднимает соответствующее исключение (рисунок 1.11):

```
# Файл calc.py

def divide(x, y):

    if y == 0:

        raise ZeroDivisionError

    return x / y
```

Рисунок 1.11 – Добавление функции деления одного числа на другое

Тогда для проверки исключений рекомендуется использовать контекстный менеджер Python, внутри которого просто вызвать проверяемую функцию (рисунок 1.12):

```
def test_divide(self):

    with self.assertRaises(ZeroDivisionError):

        calc.divide(10, 0)
```

Рисунок 1.12 – Контекстный менеджер Python

Операции перед проведением тестов

Порой требуется выполнить операции перед каждым тестом, особенно, если требуется протестировать методы класса, не создавая постоянно экземпляров класса. Более того, это позволит соблюсти принцип DRY (Don’t Repeat Youreself — не повторяйся).

Пусть в файле person.py имеется класс `Person`, который хранит информацию о имени, фамилии и e-mail (рисунок 1.13):

```
class Person:

    def __init__(self, first_name, last_name):

        self.first = first_name

        self.last = last_name



    @property

    def email(self):

        return f"{self.last}@school.ru"

    @property

    def full(self):

        return f"{self.first} {self.last}"
```

Рисунок 1.13 – Класс Person

Воспользуемся методом `setUp`, который перед каждым тестом будет создавать экземпляр класса Person. В итоге, проверим правильность введенного e-mail и полного имени (рисунок 1.14).

```
class TestPerson(unittest.TestCase):

    @classmethod

    def setUpClass(cls):

        cls.p = Person("Vasya", "Frolov")



    def test_email(self):

        self.assertEqual(self.p.email, "Frolov@school.ru")



    def test_fullname(self):

        self.assertEqual(self.p.full, "Vasya Frolov")
```

Рисунок 1.14 – Проверка введенного e-mail и полного имени

Помимо `setUp`, имеется метод `tearDown`, который, наоборот, запускается в конце каждого теста. Эти методы также пригодятся для открытия и закрытия файлов.

Источники:

1. https://realpython.com/python-unittest/
2. https://python-school.ru/blog/python/unit-tests/