

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
Федеральное государственное автономное  
образовательное учреждение высшего образования  
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Кафедра инфокоммуникаций

**Отчет по лабораторной работе № 17**

**Тестирование в Python [unittest]**

**По дисциплине «Технологии программирования и алгоритмизация»**

Выполнил студент группы ИВТ-б-о-20-1

Бобров Н. В. « » \_\_\_\_\_ 20\_\_ г.

Подпись студента \_\_\_\_\_

Работа защищена « » \_\_\_\_\_ 20\_\_ г.

Проверил Воронкин Р. А. \_\_\_\_\_

(подпись)

Ставрополь 2022

**Цель работы:** приобретение навыков написания автоматизированных тестов на языке программирования Python версии 3.x.

### **Ход работы:**

1. Создал общедоступный репозиторий и клонировал его на локальный сервер.
2. Изучил теоретический материал и приступил к выполнению задания.
3. Для детального ознакомления проработал пример.



```
7 def test_add():
8     if calc.add(1, 2) == 3:
9         print("Test add(a, b) is OK")
10    else:
11        print("Test add(a, b) is Fail")
12
13
14 def test_sub():
15     if calc.sub(4, 2) == 2:
16         print("Test sub(a, b) is OK")
17     else:
18         print("Test sub(a, b) is Fail")
19
20
21 def test_mul():
22     if calc.mul(2, 5) == 10:
23         print("Test mul(a, b) is OK")
24     else:
25         print("Test mul(a, b) is Fail")
26
27
28 def test_div():
29     if calc.div(8, 4) == 2:
30         print("Test div(a, b) is OK")
31     else:
32         print("Test div(a, b) is Fail")
```

Рисунок 1 – Код который нужно проверить

```

1  import unittest
2  import calc
3
4
5  class CalcTest(unittest.TestCase):
6      def test_add(self):
7          self.assertEqual(calc.add(1, 2), 3)
8
9      def test_sub(self):
10         self.assertEqual(calc.sub(4, 2), 2)
11
12     def test_mul(self):
13         self.assertEqual(calc.mul(2, 5), 10)
14
15     def test_div(self):
16         self.assertEqual(calc.div(8, 4), 2)
17
18
19  if __name__ == '__main__':
20      unittest.main()

```

Рисунок 2 – Модуль для проверки кода с помощью unittest

```

✓ Tests passed: 4 of 4 tests – 9 ms

C:\ProgramData\Anaconda3\envs\2.22\python
Testing started at 15:38 ...
Launching unittests with arguments python

Ran 4 tests in 0.006s

OK

Process finished with exit code 0

```

Рисунок 3 – Вывод консоли

4. Приступил к выполнению индивидуального задания, целью которого является выполнить проверку операций по работе с БД.

```

def test_create_db(self):
    self.cursor.execute(
        """
        SELECT name FROM sqlite_master WHERE type = 'table' AND name = 'students' OR name = 'student_name'
        """
    )
    table = self.cursor.fetchall()
    self.assertEqual(table, [('student_name',), ('students',)])

def test_add_student(self):
    individual.add_student(self.fullpath, 'text', 3, 'text')
    self.cursor.execute(
        """
        SELECT student_name.name, students.groupt, students.grade
        FROM students
        INNER JOIN student_name ON student_name.student_id = students.student_id
        WHERE students.student_id = (SELECT MAX(student_id) FROM students)
        """
    )
    rows = self.cursor.fetchall()
    self.last = [
        {
            "name": row[0],
            "groupt": row[1],
            "grade": row[2]
        }
    ]

def test1_select_student_1(self):
    self.assertEqual(self.result, [{'name': 'Bobrov N.V', 'groupt': 1, 'grade': '5 5 5 5 5'}])

def test1_select_student_2(self):
    self.assertNotEqual(self.result, [{'name': 'Ivanov I.I', 'groupt': 2, 'grade': '4 4 4 4 4'}])

def tearDown(self):
    self.conn.close()
    self.tmp.cleanup()

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

```

Рисунок 4 – Код для проверки операций

5. Запустил тест для проверки.

```

✓ Tests passed: 4 of 4 tests – 122 ms

C:\ProgramData\Anaconda3\envs\2.22\p
Testing started at 15:45 ...
Launching unittests with arguments p

Process finished with exit code 0

Ran 4 tests in 0.126s

OK

```

Рисунок 5 – Вывод тестов

### Контрольные вопросы:

1. Для чего используется автономное тестирование?

Сущность вашей программы. Автономное тестирование ещё называют модульным или *unit*-тестированием (*unit-testing*). Здесь и далее под словом тестирование будет пониматься именно автономное тестирование.

Важной характеристикой *unit*-теста является его повторяемость, т. е. результат его работы не зависит от окружения (внешнего мира), если же приходится обращаться к внешнему миру в процессе выполнения теста, то необходимо предусмотреть возможность подмены “мира” какой-то статичной сущностью.

2. Какие фреймворки Python получили наибольшее распространение для решения задач автономного тестирования?

- unittest
- nose
- pytest

3. Какие существуют основные структурные единицы модуля *unittest*?

### **Test fixture**

*Test fixture* – обеспечивает подготовку окружения для выполнения тестов, а также организацию мероприятий по их корректному завершению (например очистка ресурсов). Подготовка окружения может включать в себя создание баз данных, запуск необходим серверов и т.п.

### **Test case**

*Test case* – это элементарная единица тестирования, в рамках которой проверяется работа компонента тестируемой программы (метод, класс, поведение и т.п.). Для реализации этой сущности используется класс *TestCase*.

### **Test suite**

*Test suite* – это коллекция тестов, которая может в себя включать как отдельные *test case*’ы так и целые коллекции (т.е. можно создавать коллекции коллекций). Коллекции используются с целью объединения тестов для совместного запуска.

### **Test runner**

*Test runner* – это компонент, которые оркестрирует (координирует взаимодействие) запуск тестов и предоставляет пользователю результат их выполнения. *Test runner* может иметь графический интерфейс, текстовый интерфейс или возвращать какое-то заранее заданное значение, которое будет описывать результат прохождения тестов.

4. Какие существуют способы запуска тестов *unittest*?

Запуск тестов можно сделать как из командной строки, так и с помощью графического интерфейса пользователя (GUI).

5. Каково назначение класса *TestCase*?

Он представляет собой класс, который должен являться базовым для всех остальных классов, методы которых будут тестировать те или иные автономные единицы исходной программы.

6. Какие методы класса *TestCase* выполняются при запуске и завершении работы тестов?

*setUp()*

Метод вызывается перед запуском теста. Как правило, используется для подготовки окружения для теста.

*tearDown()*

Метод вызывается после завершения работы теста. Используется для “приборки” за тестом.

*setUpClass()*

Метод действует на уровне класса, т.е. выполняется перед запуском тестов класса. При этом синтаксис требует наличие декоратора *@classmethod*.

*tearDownClass()*

Запускается после выполнения всех методов класса, требует наличия декоратора *@classmethod*.

*skipTest(reason)*

Данный метод может быть использован для пропуска теста, если это необходимо.

7. Какие методы класса *TestCase* используются для проверки условий и генерации ошибок?

***TestCase* класс предоставляет набор *assert*-методов для проверки и генерации ошибок:** `assertEqual(a, b)`, `assertNotEqual(a, b)`, `assertTrue(x)`, `assertFalse(x)`, `assertIs(a, b)`, `assertIsNot(a, b)`, `assertIsNone(x)`, `assertIsNotNone(x)`, `assertIn(a, b)`, `assertNotIn(a, b)`, `assertIsInstance(a, b)`, `assertNotIsInstance(a, b)`.

**Assert'ы для контроля выбрасываемых исключений и warning'ов:** `assertRaises(exc, fun, *args, **kwds)`, `assertRaisesRegex(exc, r, fun, *args, **kwds)`, `assertWarns(warn, fun, *args, **kwds)`, `assertWarnsRegex(warn, r, fun, *args, **kwds)`.

**Assert'ы для проверки различных ситуаций:** `assertAlmostEqual(a, b)`, `assertNotAlmostEqual(a, b)`, `assertGreater(a, b)`, `assertGreaterEqual(a, b)`, `assertLess(a, b)`, `assertLessEqual(a, b)`, `assertRegex(s, r)`, `assertNotRegex(s, r)`, `assertCountEqual(a, b)`

**Типо-зависимые assert'ы, которые используются при вызове `assertEqual()`.** Приводятся на тот случай, если необходимо использовать конкретный метод: `assertMultiLineEqual(a, b)`, `assertSequenceEqual(a, b)`, `assertListEqual(a, b)`, `assertTupleEqual(a, b)`, `assertSetEqual(a, b)`, `assertDictEqual(a, b)`.

Дополнительно хотелось бы отметить метод **`fail()`**: `fail(msg=None)`. Этот метод сигнализирует о том, что произошла ошибка в тесте.

8. Какие методы класса *TestCase* позволяют собирать информацию о самом тесте?

***countTestCases()***

Возвращает количество тестов в объекте класса-наследника от *TestCase*.

***id()***

Возвращает строковый идентификатор теста. Как правило, это полное имя метода, включающее имя модуля и имя класса.

***shortDescription()***

Возвращает описание теста, которое представляет собой первую строку *docstring*'а метода, если его нет, то возвращает *None*.

9. Каково назначение класса *TestSuite*? Как осуществляется загрузка тестов?

Класс *TestSuite* используется для объединения тестов в группы, которые могут включать в себя как отдельные тесты так и заранее созданные группы. Помимо этого, *TestSuite* предоставляет интерфейс, позволяющий *TestRunner* 'у, запускать тесты. Разберем более подробно методы класса *TestSuite*.

***addTest(test)***

Добавляет *TestCase* или *TestSuite* в группу.

***addTests(tests)***

Добавляет все *TestCase* и *TestSuite* объекты в группу, итеративно проходя по элементам переменной *tests*.

***\*run(result)\****

Запускает тесты из данной группы.

***countTestCases()***

Возвращает количество тестов в данной группе (включает в себя как отдельные тесты, так и подгруппы).

10. Каково назначение класса *TestResult*?

Класс *TestResult* используется для сбора информации о результатах прохождения тестов

11. Для чего может понадобиться пропуск отдельных тестов?

*unittest* предоставляет нам инструменты для удобного управления процессом пропуска тестов. Это может быть ещё полезно в том плане, что информацию о пропущенных тестах (их количестве) можно дополнительно получить через специальный *API*, предоставляемый классом *TestResult*

12. Как выполняется безусловный и условных пропуск тестов? Как выполнить пропуск класса тестов?

Для безусловного пропуска тестов применяется декоратор:

***@unittest.skip(reason)***



Для условного пропуска тестов применяются следующие декораторы:

`@unittest.skipIf(condition, reason)`

Тест будет пропущен, если условие (*condition*) истинно:

`@unittest.skipUnless(condition, reason)`

Условный пропуск тестов можно использовать в ситуациях, когда те или иные тесты зависят от версии программы, например: в новой версии уже не поддерживается часть методов; или тесты могут быть платформозависимые, например: ряд тестов могут выполняться только под операционной системой *MS Windows*. Условие записывается в параметр *condition*, текстовое описание – в *reason*.

Для пропуска классов используется декоратор: `@unittest.skip(reason)`, который записывается перед объявлением класса. В результате все тесты из данного класса не будут выполнены.

**Вывод:** в ходе выполнения лабораторной работы были приобретены навыки написания автоматизированных тестов на языке программирования Python версии 3.x.