

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

Кафедра инфокоммуникаций

**Отчет по лабораторной работе №2.22**

**Тестирование в Python [unittest]**

**По дисциплине «Теории программирования и алгоритмизации»**

Выполнил студент группы ИВТ-б-о-20-1

Плотников Д. 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 новый собственный репозиторий. Ссылка на репозиторий: [https://github.com/Dmitry-15/2.22\\_laba](https://github.com/Dmitry-15/2.22_laba).
2. С помощью команды `git clone` клонировал удаленный репозиторий на свой ПК. Дополнил файл `.gitignore` необходимыми правилами для работы с IDE PyCharm.
3. После ознакомления с теоретическим материалом приступил к выполнению заданий.

### Пример

1. Для начала выполнил пример.

```
PS C:\Users\Plotnikov\PycharmProjects\laba_2.22> python test_calc.py
Test add(a, b) is OK
Test sub(a, b) is OK
Test mul(a, b) is OK
Test div(a, b) is OK
PS C:\Users\Plotnikov\PycharmProjects\laba_2.22>
```

Рисунок 1 – Проверка корректности работы функций из `calc.py`

```
✓ Tests passed: 4 of 4 tests – 10 ms
C:\tools\Anaconda3\envs\laba_2.22\python.exe "C:\Program Files\JetBrains\PyCharm
Testing started at 16:36 ...

Ran 4 tests in 0.011s

OK
```

Рисунок 2 – Тестирование с помощью `unittest`

2. Приступил к выполнению индивидуального задания.

### Индивидуальное задание

1. Условие задания: для индивидуального задания лабораторной работы 2.21 добавьте тесты с использованием модуля `unittest`, проверяющие операции по работе с базой данных.

```
PS C:\Users\Plotnikov\PycharmProjects\laba_2.22> python individ1.py add -n "Плотников Д. В." -z "Лев" -yr "15.08.2002"
PS C:\Users\Plotnikov\PycharmProjects\laba_2.22> python individ1.py add -n "Бобров Н. В." -z "Стрелец" -yr "19.12.2002"
```

### Рисунок 3 – Добавление людей

2. Затем перешел в базу данных для проверки добавления двух таблиц с информацией о людях.

Таблица: human\_name

	human_id	name
	Фильтр	Фильтр
1	1	Плотников Д. В.
2	2	Бобров Н. В.

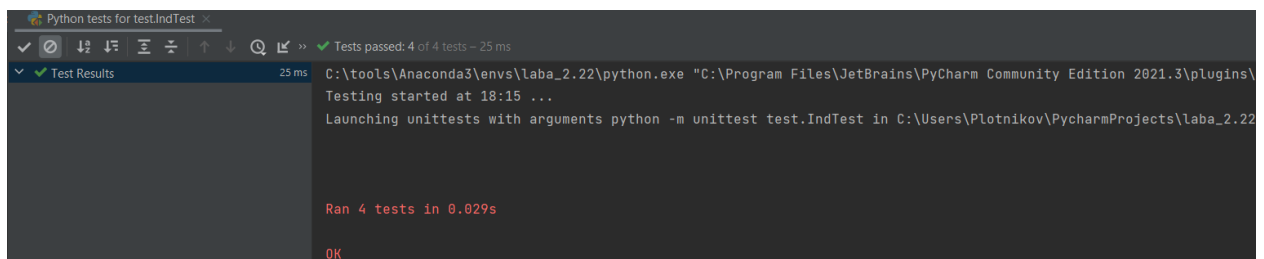
### Рисунок 4 – Первая таблица

Таблица: people

	human_id	zodiac	year
	Фильтр	Фильтр	Фильтр
1	1	Лев	15.08.2002
2	2	Стрелец	19.12.2002

### Рисунок 5 – Вторая таблица

3. Далее выполнил тест.



### Рисунок 6 – Выполнение теста

#### 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, **kws)`, `assertRaisesRegex(exc, r, fun, *args, **kws)`, `assertWarns(warn, fun, *args, **kws)`, `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'a* 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 успешно были приобретены навыки написания автоматизированных тестов на языке программирования Python версии 3.x.