**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

**ИНСТИТУТ ЦИФРОВОГО РАЗВИТИЯ**

# Отчёт по лабораторной работе №2.22 по дисциплине:

# Основы программной инженерии

Выполнила:
студент группы ПИЖ-б-о-20-1
Лазарева Дарья Олеговна

Проверил:
доцент кафедры инфокоммуникаций
Романкин Р.А.

Ставрополь, 2022 г.

ВЫПОЛНЕНИЕ:

1. Пример тестирования приложения без framework’a

```
PS D:\УЧЕБА\ОПИ\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
```

2. Пример тестирования приложения с использованием unittest

```
PS D:\УЧЕБА\ОПИ\2.22> py -m unittest utest_calc.py
setUpClass
==========
Set up for [Add operation test]
id: utest_calc.CalcTest.test_add
Tear down for [Add operation test]

.Set up for [Div operation test]
id: utest_calc.CalcTest.test_div
Tear down for [Div operation test]

.Set up for [Mul operation test]
id: utest_calc.CalcTest.test_mul
Tear down for [Mul operation test]

.Set up for [Sub operation test]
id: utest_calc.CalcTest.test_sub
Tear down for [Sub operation test]

.==========
tearDownClass

----------------------------------------------------------------------
Ran 4 tests in 0.007s
```

3. Запуск с запросом расширенной информации (ключ –v)

```
ok
test_div (utest_calc.CalcTest)
Div operation test ... Set up for [Div operation test]
id: utest_calc.CalcTest.test_div
Tear down for [Div operation test]

ok
test_mul (utest_calc.CalcTest)
Mul operation test ... Set up for [Mul operation test]
id: utest_calc.CalcTest.test_mul
Tear down for [Mul operation test]

ok
test_sub (utest_calc.CalcTest)
Sub operation test ... Set up for [Sub operation test]
id: utest_calc.CalcTest.test_sub
Tear down for [Sub operation test]

ok
==========
tearDownClass

----------------------------------------------------------------------
Ran 4 tests in 0.012s

OK
```

4. Методы, позволяющие собирать информацию о самом тесте

```
28      def tearDown(self):
29          """Tear down for test"""
30          print("Tear down for [" + self.shortDescription() + "]")
31          print("")
32
33      def test_add(self):
34          """Add operation test"""
35          print("id: " + self.id())
36          self.assertEqual(calc.add(1, 2), 3)
37
38      def test_sub(self):
39          """Sub operation test"""
40          print("id: " + self.id())
41          self.assertEqual(calc.sub(4, 2), 2)
42
43      def test_mul(self):
44          """Mul operation test"""
45          print("id: " + self.id())
46          self.assertEqual(calc.mul(2, 5), 10)
47
48      def test_div(self):
49          """Div operation test"""
```

CalcTest › test_div()

Terminal: Local + ⌄

```
tearDownClass

----------------------------------------------------------------------
Ran 4 tests in 0.015s

OK
```

5. Класс TestSuite

```python
import unittest
import calc_tests


calcTestSuite = unittest.TestSuite()
calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcTest))

runner = unittest.TextTestRunner(verbosity=2)
runner.run(calcTestSuite)

```

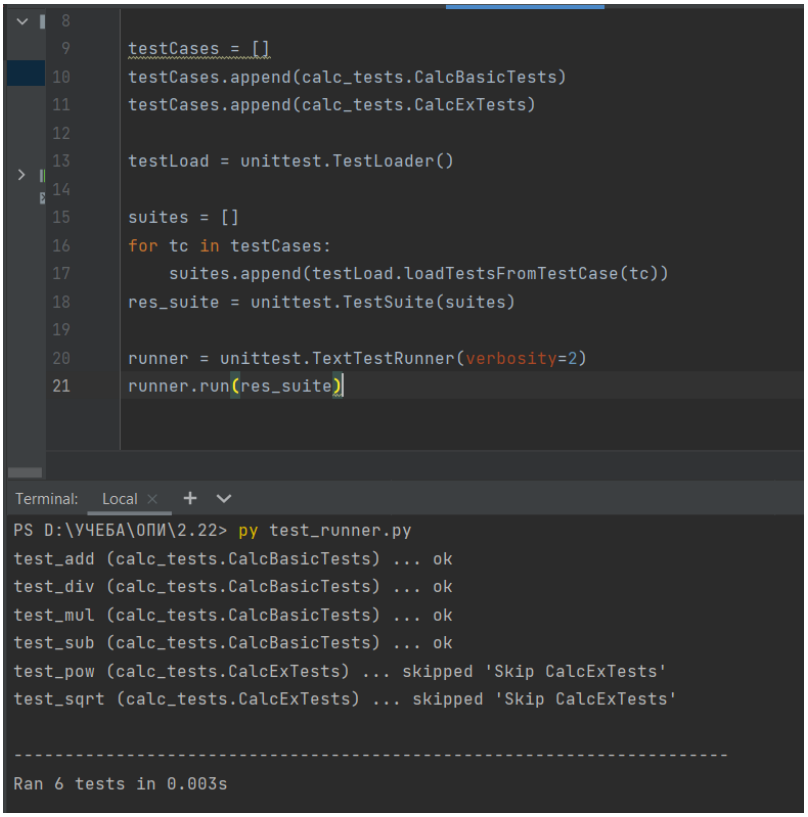
```
Terminal:  Local
PS D:\УЧЕБА\ОПИ\2.22> py test_runner.py

----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

6. Класс TestResult

```python
testLoad = unittest.TestLoader()
suites = testLoad.loadTestsFromModule(calc_tests)
testResult = unittest.TestResult()
runner = unittest.TextTestRunner(verbosity=1)
testResult = runner.run(suites)
print("errors")
print(len(testResult.errors))
print("failures")
print(len(testResult.failures))
print("skipped")
print(len(testResult.skipped))
print("testsRun")
print(testResult.testsRun)
```

```
Terminal:  Local
PS D:\УЧЕБА\ОПИ\2.22> py test_runner.py
....ss
----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK (skipped=2)
errors
0
failures
0
skipped
2
testsRun
6
```

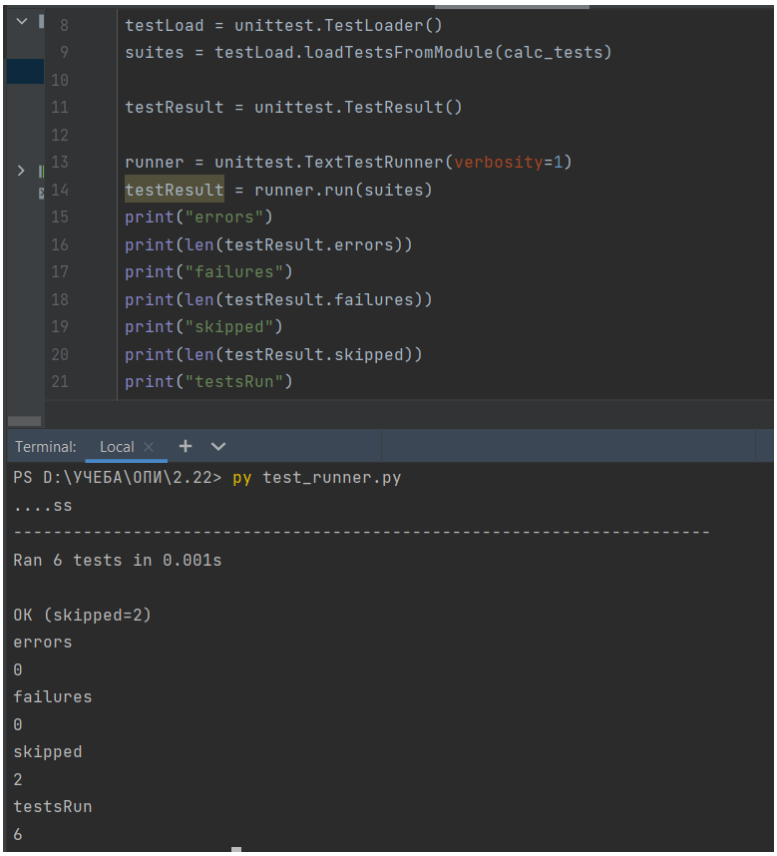
7. Пропуск отдельных классов в тесте



```
testCases = []
testCases.append(calc_tests.CalcBasicTests)
testCases.append(calc_tests.CalcExTests)

testLoad = unittest.TestLoader()

suites = []
for tc in testCases:
    suites.append(testLoad.loadTestsFromTestCase(tc))
res_suite = unittest.TestSuite(suites)

runner = unittest.TextTestRunner(verbosity=2)
runner.run(res_suite)
```

```
Terminal:  Local
PS D:\УЧЕБА\ОПИ\2.22> py test_runner.py
test_add (calc_tests.CalcBasicTests) ... ok
test_div (calc_tests.CalcBasicTests) ... ok
test_mul (calc_tests.CalcBasicTests) ... ok
test_sub (calc_tests.CalcBasicTests) ... ok
test_pow (calc_tests.CalcExTests) ... skipped 'Skip CalcExTests'
test_sqrt (calc_tests.CalcExTests) ... skipped 'Skip CalcExTests'

----------------------------------------------------------------------
Ran 6 tests in 0.003s
```

8. Пропуск классов



```
testLoad = unittest.TestLoader()
suites = testLoad.loadTestsFromModule(calc_tests)

testResult = unittest.TestResult()

runner = unittest.TextTestRunner(verbosity=1)
testResult = runner.run(suites)
print("errors")
print(len(testResult.errors))
print("failures")
print(len(testResult.failures))
print("skipped")
print(len(testResult.skipped))
print("testsRun")
```

```
Terminal:  Local
PS D:\УЧЕБА\ОПИ\2.22> py test_runner.py
....ss
----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK (skipped=2)
errors
0
failures
0
skipped
2
testsRun
6
```

ВОПРОСЫ

1. Для чего используется автономное тестирование?

Для тестирования функций, классов, методов и т.д. с целю выявления ошибок в работе в этих отдельных единицах общей программы.

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

В мире Python существуют три framework'а, которые получили наибольшее распространение:

- unittest
- nose
- pytest

**unittest**

unittest – это framework для тестирования, входящий в стандартную библиотеку языка Python. Его архитектура выполнена в стиле xUnit. xUnit представляет собой семейство framework'ов для тестирования в разных языках программирования, в Java – это JUnit, C# – NUnit и т.д.

**nose**

Девизом nose является фраза "nose extends unittest to make testing easier", что можно перевести как "nose расширяет unittest, делая тестирование проще". nose идеален, когда нужно сделать тесты "по-быстрому", без предварительного планирования и выстраивания архитектуры приложения с тестами. Функционал nose можно расширять и настраивать с помощью плагинов.

**pytest**

pytest довольно мощный инструмент для тестирования, и многие разработчики оставляют свой выбор на нем. pytest по "духу" ближе к языку Python нежели unittest. Как было сказано выше, unittest в своей базе – xUnit, что накладывает определенные обязательства при разработке тестов (создание классов-наследников от unittest.TestCase, выполнение определенной процедуры запуска тестов и т.п.). При разработке на pytest ничего этого делать не нужно, вы просто пишете функции, которые должны начинаться с "test_" и используете assert'ы, встроенные в Python (unittest используется свои).

3. Какие существуют основные структурные единицы модуля unittest?

Основными структурными элемента каркаса unittest являются:

**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. Для того, чтобы метод класса выполнялся как тест, необходимо, чтобы он начинался со слова test. Несмотря на то, что методы framework'a unittest написаны не в соответствии с PEP 8 (ввиду того, что идейно он наследник xUnit), мы все же рекомендуем следовать правилам стиля для Python везде, где это возможно. Поэтому имена тестов будем начинать с префикса test_. Далее, под словом тест будем понимать метод класса-наследника от TestCase, который начинается с префикса test_.

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

К этим методам относятся:

**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. Заметим, что методы setUp() и tearDown() вызываются для всех тестов в рамках класса, в котором они переопределены. По умолчанию, эти методы ничего не делают. Если их добавить в utest_calc.py, то перед [после] тестов test_add(), test_sub(), test_mul(), test_div() будут выполнены setUp() [tearDown()].

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

TestCase класс предоставляет набор assert-методов для проверки и генерации ошибок:

| Метод | Описание |
|---|---|
| `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) |

Assert’ы для контроля выбрасываемых исключений и warning’ов:

| Метод | Описание |
| --- | --- |
| `assertRaises(exc, fun, *args, **kwds)` | Функция `fun(*args, **kwds)` вызывает исключение exc |
| `assertRaisesRegex(exc, r, fun, *args, **kwds)` | Функция `fun(*args, **kwds)` вызывает исключение exc, сообщение которого совпадает с регулярным выражением r |
| `assertWarns(warn, fun, *args, **kwds)` | Функция `fun(*args, **kwds)` выдает сообщение warn |
| `assertWarnsRegex(warn, r, fun, *args, **kwds)` | Функция `fun(*args, **kwds)` выдает сообщение warn и оно совпадает с регулярным выражением r |

Assert’ы для проверки различных ситуаций:

| Метод | Описание |
| --- | --- |
| `assertAlmostEqual(a, b)` | round(a-b, 7) == 0 |
| `assertNotAlmostEqual(a, b)` | round(a-b, 7) != 0 |
| `assertGreater(a, b)` | a > b |
| `assertGreaterEqual(a, b)` | a >= b |
| `assertLess(a, b)` | a < b |
| `assertLessEqual(a, b)` | a <= b |
| `assertRegex(s, r)` | r.search(s) |
| `assertNotRegex(s, r)` | not r.search(s) |
| `assertCountEqual(a, b)` | a и b имеют одинаковые элементы (порядок неважен) |

Типо-зависимые assert’ы, которые используются при вызове assertEqual() . Приводятся на тот случай, если необходимо использовать конкретный метод.

| Метод | Описание |
| --- | --- |
| `assertMultiLineEqual(a, b)` | строки (strings) |
| `assertSequenceEqual(a, b)` | последовательности (sequences) |
| `assertListEqual(a, b)` | списки (lists) |
| `assertTupleEqual(a, b)` | кортежи (tuplse) |
| `assertSetEqual(a, b)` | множества или неизменяемые множества (frozensets) |
| `assertDictEqual(a, b)` | словари (dicts) |

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.

Метод *run(result)* запускает тесты из данной группы.

Начнем с класса TestLoader. Этот класс используется для создания групп из классов и модулей. Среди методов TestLoader можно выделить:

loadTestsFromTestCase(testCaseClass), возвращающий группу со всеми тестами из класса testCaseClass. Напоминаем, что под тестом понимается модуль, начинающийся со слова “test”. Используя этот метод, можно создать список групп тестов, где каждая группа создается на базе классов-наследников от TestCase, объединенных предварительно в список.

10. Каково назначение класса TestResult?

Класс TestResult используется для сбора информации о результатах прохождения тестов.

11. Для чего может понадобиться пропуск отдельных тестов?

Во избежание ошибок тестирования, так как некоторые тесты могут давать заведомо неправильный результат в зависимости от какого-либо условия. Для этого такие тесты необходимо пропускать.

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

Безусловный пропуск: @unittest.skip(reason) записывается перед объявлением теста.

Условный пропуск:

1) @unittest.skipIf(condition, reason) – Тест будет пропущен, если условие (condition) истинно.
2) @unittest.skipUnless(condition, reason) – Тест будет пропущен если, условие (condition) не истинно.

Пропуск класса тестов: @unittest.skip(reason) записывается перед объявлением класса.

13. Самостоятельно изучить средства по поддержке тестов unittest в PyCharm. Приведите обобщенный алгоритм проведения тестирования с помощью PyCharm.

В PyCharm есть встроенная поддержка unit тестов, которая позволяет создавать шаблон класса для тестирования и его дальнейшей настройки.

1) Необходимо создать класс для тестирования.

2) Написание кода тестов в классе для тестирования частей программы.

3) Запуск тестов

4) Debug тестов при необходимости.

5) Автоматизация тестов. PyCharm поддерживает автоматизацию тестов – установив её, вы можете сфокусироваться в написании кода самой

программы, а IDE будет в автоматическом режиме проводить тестированиепо мере изменения кода.