

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт перспективной инженерии
Департамент цифровых, робототехнических систем и электроники

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №2.22
дисциплины «Объектно-ориентированное программирование»

Выполнил:
Магомедов Имран Борисович
3 курс, группа «Программная инженерия»,
направленность (профиль) «Разработка
и сопровождение программного
обеспечения», очная форма обучения

(подпись)

Руководитель практики:
Воронкин Р.А., доцент департамента
цифровых, робототехнических систем и
электроники

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

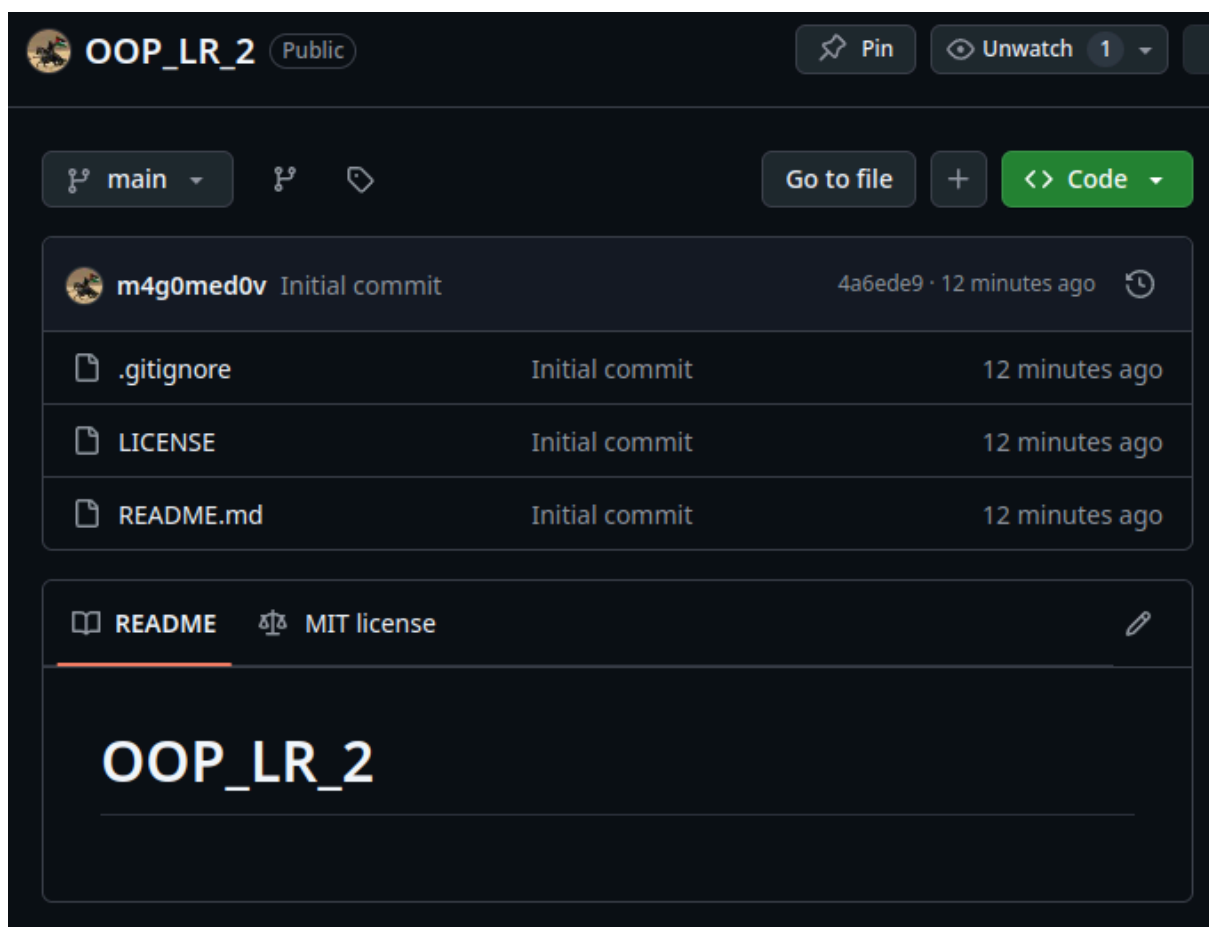
Ставрополь, 2024 г.

Тема: Тестирование в Python [unittest].

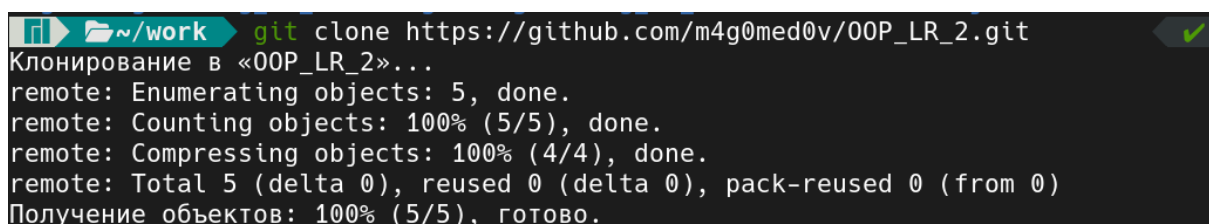
Цель работы: приобретение навыков написания автоматизированных тестов на языке программирования Python версии 3.x.

Методика выполнения работы

1. Изучил теоретический материал работы.
2. Создал общедоступный репозиторий на Github, в котором использовал лицензию MIT и язык программирования Python.



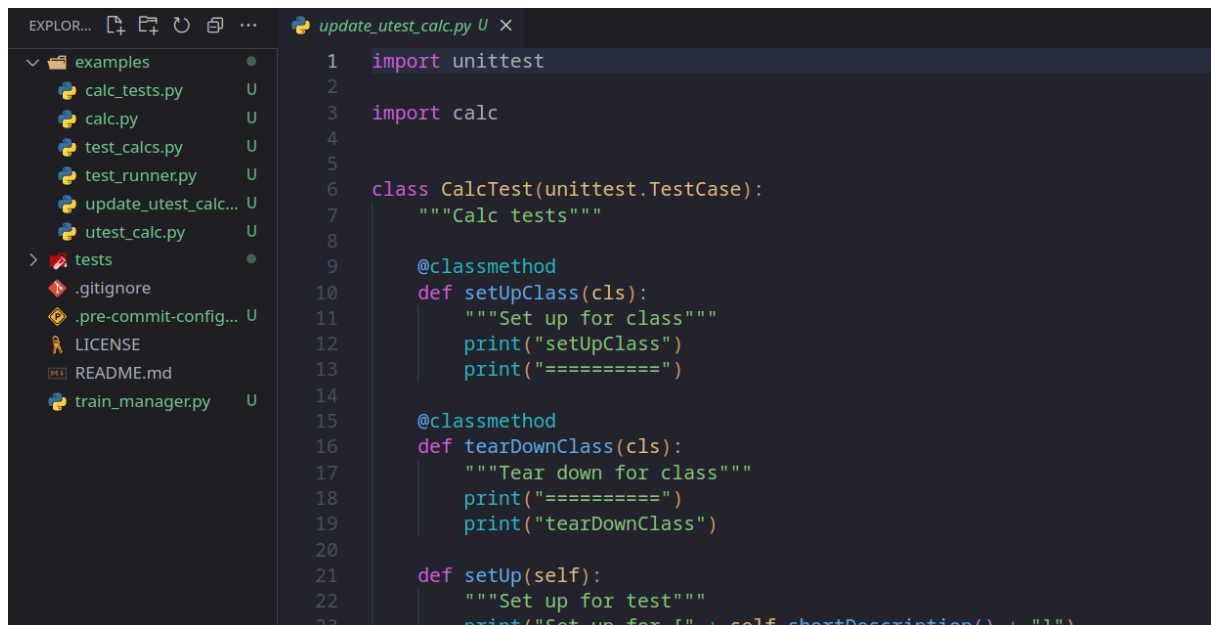
3. Выполните клонирование созданного репозитория.



4. Организовал свой репозиторий в соответствие с моделью ветвление git-flow.

```
~/w/OOP_LR_2 main git branch develop ✓
~/w/OOP_LR_2 main git switch develop ✓
Переключились на ветку «develop»
```

5. Проработал примеры лабораторной работы.



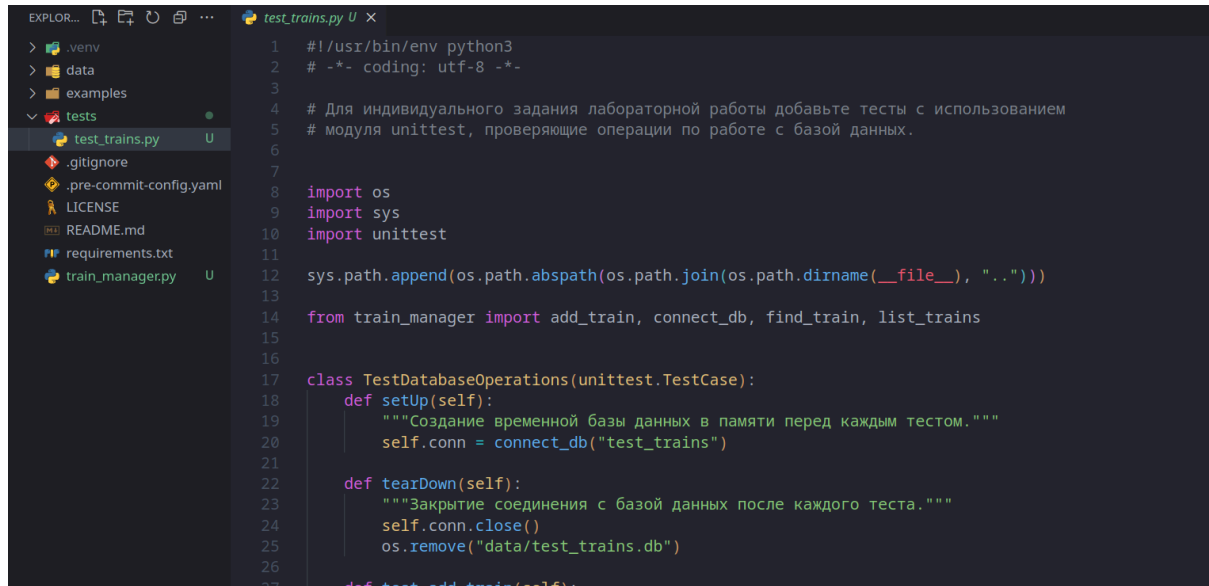
```
1 import unittest
2
3 import calc
4
5
6 class CalcTest(unittest.TestCase):
7     """Calc tests"""
8
9     @classmethod
10    def setUpClass(cls):
11        """Set up for class"""
12        print("setUpClass")
13        print("=====")
14
15    @classmethod
16    def tearDownClass(cls):
17        """Tear down for class"""
18        print("=====")
19        print("tearDownClass")
20
21    def setUp(self):
22        """Set up for test"""
23        print("Set up for [" + self.shortDescription() + "]\n")
```

6. Зафиксируйте сделанные изменения в репозитории.

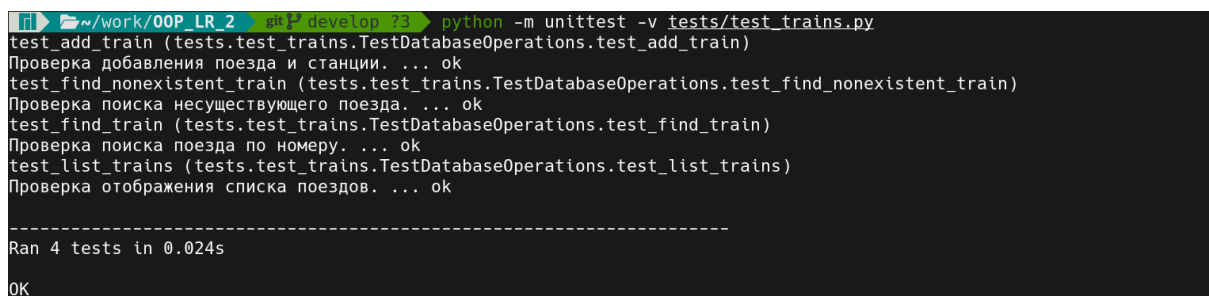
```
~/work/OOP_LR_2 git develop +6 ?2 git commit -m 'Added examples from LR'
check yaml.....(no files to check)Skipped
fix end of files.....Passed
trim trailing whitespace.....Passed
ruff.....Passed
ruff-format.....Passed
[develop 7bfe814] Added examples from LR
6 files changed, 171 insertions(+)
create mode 100644 examples/calc.py
create mode 100644 examples/calc_tests.py
create mode 100644 examples/test_calcs.py
create mode 100644 examples/test_runner.py
create mode 100644 examples/update_uteest_calc.py
create mode 100644 examples/uteest_calc.py
```

7. Выполните индивидуального задание. Приведите в отчет скриншот работы программы решения индивидуального задания.

Для индивидуального задания лабораторной работы 2.21 добавьте тесты с использованием модуля unittest, проверяющие операции по работе с базой данных.

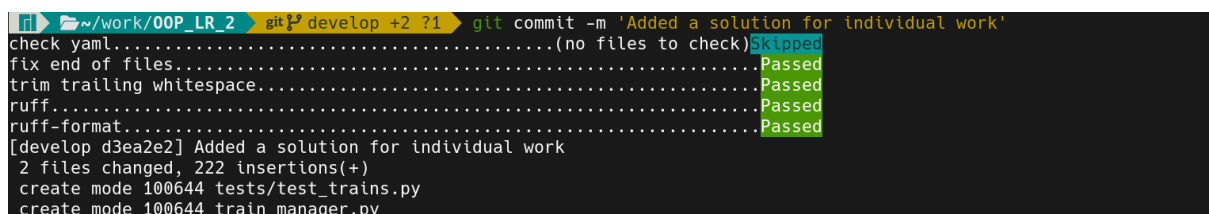


```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 # Для индивидуального задания лабораторной работы добавьте тесты с использованием
5 # модуля unittest, проверяющие операции по работе с базой данных.
6
7
8 import os
9 import sys
10 import unittest
11
12 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
13
14 from train_manager import add_train, connect_db, find_train, list_trains
15
16
17 class TestDatabaseOperations(unittest.TestCase):
18     def setUp(self):
19         """Создание временной базы данных в памяти перед каждым тестом."""
20         self.conn = connect_db("test_trains")
21
22     def tearDown(self):
23         """Закрытие соединения с базой данных после каждого теста."""
24         self.conn.close()
25         os.remove("data/test_trains.db")
26
27     def test_add_train(self):
```



```
~/work/OOP_LR_2 git develop ?3 python -m unittest -v tests/test_trains.py
test_add_train (tests.test_trains.TestDatabaseOperations.test_add_train)
Проверка добавления поезда и станции. ... ok
test_find_nonexistent_train (tests.test_trains.TestDatabaseOperations.test_find_nonexistent_train)
Проверка поиска несуществующего поезда. ... ok
test_find_train (tests.test_trains.TestDatabaseOperations.test_find_train)
Проверка поиска поезда по номеру. ... ok
test_list_trains (tests.test_trains.TestDatabaseOperations.test_list_trains)
Проверка отображения списка поездов. ... ok
-----
Ran 4 tests in 0.024s
OK
```

8. Зафиксируйте сделанные изменения в репозитории.



```
~/work/OOP_LR_2 git develop +2 ?1 git commit -m 'Added a solution for individual work'
check yaml.....(no files to check)Skipped
fix end of files.....Passed
trim trailing whitespace.....Passed
ruff.....Passed
ruff-format.....Passed
[develop d3ea2e2] Added a solution for individual work
2 files changed, 222 insertions(+)
create mode 100644 tests/test_trains.py
create mode 100644 train_manager.py
```

9. Выполните слияние ветки для разработки с веткой main.

```

~/work/OOP_LR_2 main ?1 git merge develop
Обновление 4a6ede9..d3ea2e2
Fast-forward
 .pre-commit-config.yaml      | 20 ++++++
 examples/calc.py              | 14 ++++++
 examples/calc_tests.py        | 41 ++++++
 examples/test_calcs.py        | 35 ++++++
 examples/test_runner.py        | 9 ++++++
 examples/update_ute_test_calc.py | 52 ++++++
 examples/ute_test_calc.py      | 20 ++++++
 requirements.txt               | 1 +
 tests/test_trains.py          | 75 ++++++
 train_manager.py              | 147 ++++++
10 files changed, 414 insertions(+)
create mode 100644 .pre-commit-config.yaml
create mode 100644 examples/calc.py
create mode 100644 examples/calc_tests.py
create mode 100644 examples/test_calcs.py
create mode 100644 examples/test_runner.py
create mode 100644 examples/update_ute_test_calc.py
create mode 100644 examples/ute_test_calc.py
create mode 100644 requirements.txt
create mode 100644 tests/test_trains.py
create mode 100644 train_manager.py

```

10. Отправьте сделанные изменения на сервер Github.

```

~/work/OOP_LR_2 main i3 ?1 git push
Перечисление объектов: 19, готово.
Подсчет объектов: 100% (19/19), готово.
При сжатии изменений используется до 12 потоков
Сжатие объектов: 100% (16/16), готово.
Запись объектов: 100% (18/18), 5.04 КиБ | 5.04 МиБ/с, готово.
Total 18 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (3/3), done.
To https://github.com/m4g0med0v/OOP_LR_2.git
 4a6ede9..d3ea2e2 main -> main

```

Контрольные вопросы

1. Для чего используется автономное тестирование?

Автономное тестирование, также называемое модульным или unit-тестированием, используется для проверки работы отдельных функций или компонентов программы в изоляции от остальных частей системы. Оно позволяет разработчикам выявлять ошибки на раннем этапе разработки, не полагаясь на взаимодействие с другими модулями или внешними ресурсами. Это особенно важно для обеспечения стабильности и надежности кода.

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

Среди фреймворков Python, которые получили наибольшее распространение для решения задач автономного тестирования, можно

выделить три основных: unittest, nose и pytest. Фреймворк unittest является частью стандартной библиотеки Python и реализует архитектуру xUnit, что делает его знакомым для разработчиков, работающих с другими языками, такими как Java или C#. nose расширяет возможности unittest, делая тестирование более простым и удобным, тогда как pytest предлагает мощные инструменты и более естественный для Python синтаксис, позволяющий писать тесты без необходимости создания классов.

3. Какие существуют основные структурные единицы модуля unittest?

Основными структурными единицами модуля unittest являются такие элементы, как TestCase, TestSuite, TestRunner, TestResult и TestLoader. Класс TestCase используется для создания и выполнения отдельных тестов, тогда как TestSuite позволяет группировать несколько тестов в одну коллекцию для совместного запуска. TestRunner отвечает за выполнение тестов и отображение результатов, а TestResult собирает информацию о выполнении тестов, включая количество успешных и неудачных тестов. TestLoader используется для автоматической загрузки тестов из модулей или классов.

4. Какие существуют способы запуска тестов unittest?

Существуют различные способы запуска тестов в unittest. Тесты можно запускать из командной строки, используя команду `python -m unittest`, либо можно воспользоваться встроенной функцией `unittest.main()` для запуска тестов из самого скрипта. Кроме того, многие интегрированные среды разработки (IDE), такие как PyCharm, позволяют запускать тесты через графический интерфейс, а также поддерживают автоматическое обнаружение тестов.

5. Каково назначение класса TestCase?

Класс `TestCase` служит для создания тестов и проверки работы отдельных методов или классов. Каждый метод в классе `TestCase`, имя которого начинается с `test`, будет считаться отдельным тестом. Этот класс предоставляет разработчикам методы для проверки условий и управления жизненным циклом тестов с помощью методов `setUp()` и `tearDown()`, которые выполняются перед и после каждого теста соответственно.

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

При запуске тестов в классе `TestCase` вызываются несколько методов, таких как `setUp()` и `tearDown()`, которые помогают подготовить среду тестирования и освободить ресурсы после выполнения тестов. Эти методы обеспечивают чистоту и изолированность каждого теста, что способствует более надежным результатам.

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

Методы класса `TestCase` используются для проверки условий и генерации ошибок. Например, `assertEqual`, `assertTrue` и `assertRaises` позволяют проверять, что результат выполнения функции соответствует ожидаемым значениям, и вызывать ошибки, если тест не проходит. Эти методы помогают разработчикам удостовериться в корректности работы кода.

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

Для сбора информации о тесте класс `TestCase` предоставляет такие методы, как `countTestCases()`, который возвращает количество тестов, и

`id()`, который возвращает строковый идентификатор теста. Метод `shortDescription()` позволяет получить краткое описание теста, которое соответствует первой строке документации метода.

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

Класс `TestSuite` используется для объединения тестов в группы, которые могут включать как отдельные тесты, так и коллекции тестов. Это позволяет удобно управлять и запускать связанные тесты вместе. Тесты могут быть добавлены в `TestSuite` с помощью методов `addTest()` или `addTests()`, а также могут загружаться с помощью `TestLoader`, что упрощает процесс группирования тестов.

10. Каково назначение класса `TestResult`?

Класс `TestResult` предназначен для сбора информации о результатах выполнения тестов. Он хранит данные о количестве успешных тестов, ошибках и пропущенных тестах, что позволяет разработчикам анализировать результаты и выявлять проблемы в тестируемом коде.

11. Для чего может понадобиться пропуск отдельных тестов?

Пропуск отдельных тестов может понадобиться, если тест временно не актуален, зависит от внешних факторов или тестируемый функционал больше не поддерживается.

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

Для безусловного пропуска тестов используется декоратор `@unittest.skip(reason)`, который указывается перед методом теста:


```
@unittest.skip("Тест временно пропущен")
def test_example(self):
    ...
```

Условный пропуск выполняется с помощью декораторов `@unittest.skipIf(condition, reason)` и `@unittest.skipUnless(condition, reason)`:

```
@unittest.skipIf(condition=True, reason="Пропуск из-за условия")
def test_example(self):
    ...
```

Для пропуска целого класса тестов используется тот же декоратор перед объявлением класса:

```
@unittest.skip("Пропуск всех тестов в классе")
class MyTestClass(unittest.TestCase):
    ...
```