

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

Департамент цифровых, робототехнических систем и электроники института
перспективной инженерии

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

Выполнил:
Юрьев Илья Евгеньевич
курс, группа ПИЖ-б-о-22-1,
09.03.04 «Программная инженерия»,
направленность (профиль) «Разработка
и сопровождение программного
обеспечения», очная форма обучения

(подпись)

Руководитель практики:
Хацукова А.И., ассистент
департамента цифровых
робототехнических систем и
электроники

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2024 г.

Тема: тестирование в Python.

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

Ход выполнения работы:

создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и .gitignore файл для языка программирования Python:

Owner * **daxstrong** / Repository name * **OOP_2**
✔ OOP_2 is available.

Great repository names are short and memorable. Need inspiration? How about [literate-sniffle](#) ?

Description (optional)

☒ **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

☒ **Add a README file**
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore
.gitignore template: **Python**
Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license
License: **MIT License**
A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

This will set `main` as the default branch. Change the default name in your [settings](#).

You are creating a public repository in your personal account.

Create repository

Рисунок 1 – Создание репозитория

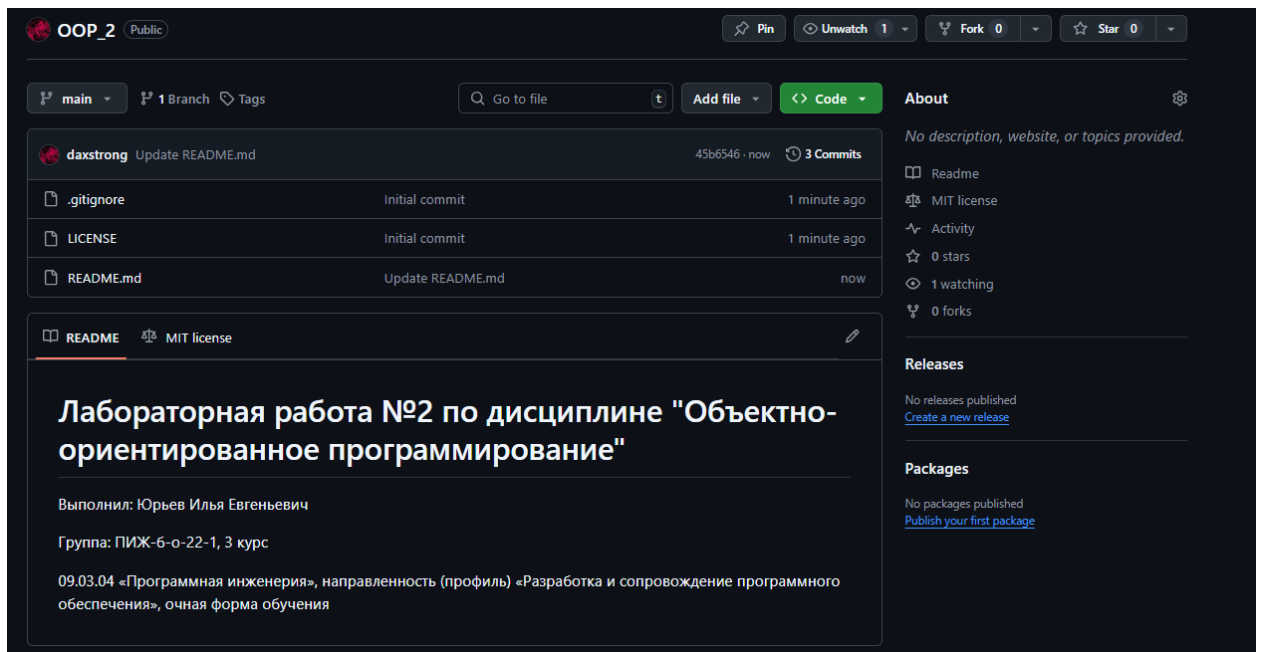


Рисунок 2 – Созданный репозиторий

```
C:\Users\Ilya>cd C:\Users\Ilya\Documents\labs\OOP

C:\Users\Ilya\Documents\labs\OOP>git clone https://github.com/daxstrong/OOP_2.git
Cloning into 'OOP_2'...
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (11/11), 5.02 KiB | 1.67 MiB/s, done.
Resolving deltas: 100% (3/3), done.

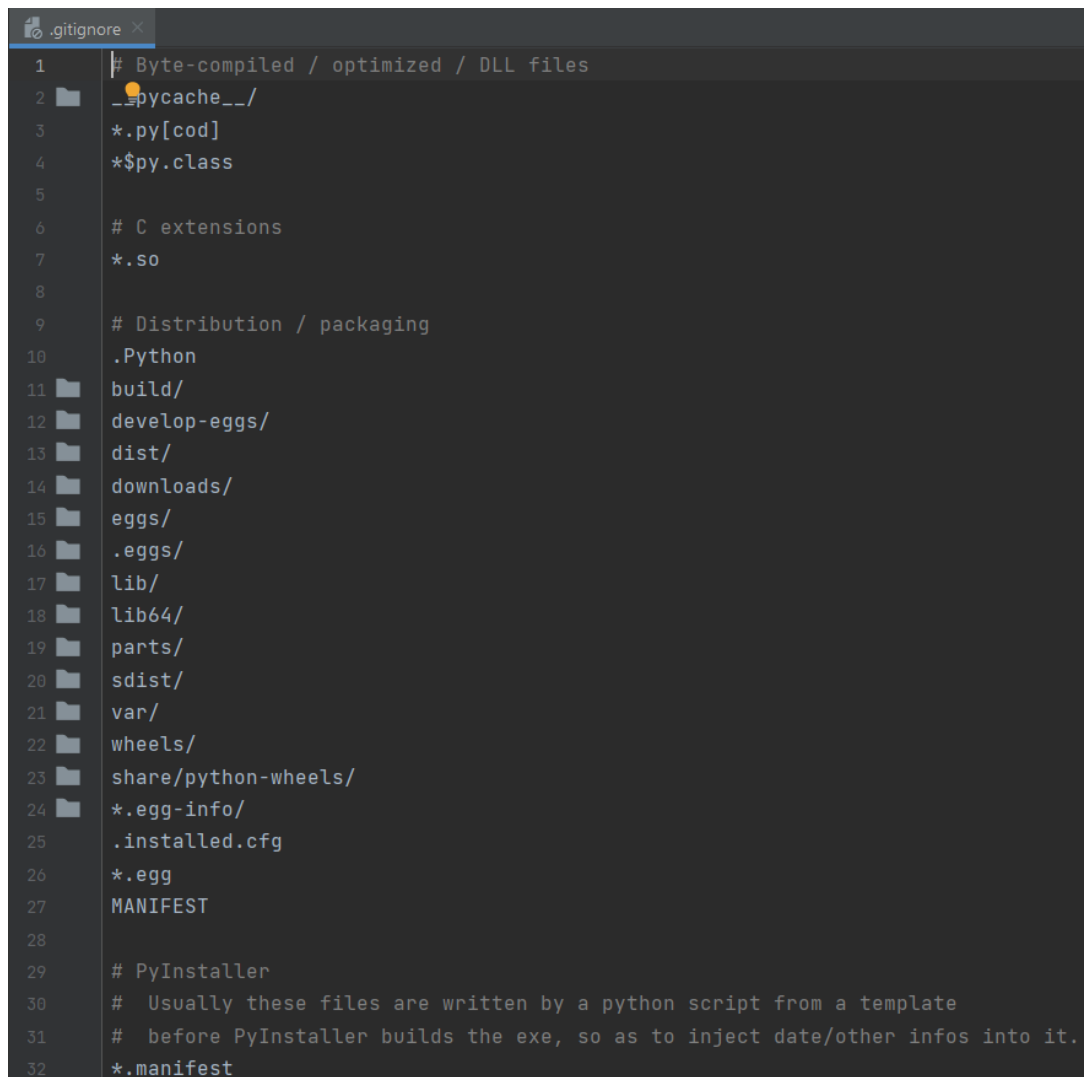
C:\Users\Ilya\Documents\labs\OOP>
```

Рисунок 3 – Клонирование репозитория

```
C:\Users\Ilya\Documents\labs\OOP\OOP_2>git checkout -b develop
Switched to a new branch 'develop'

C:\Users\Ilya\Documents\labs\OOP\OOP_2>
```

Рисунок 4 – Создание ветки разработки, в которой будут вноситься изменения до окончательного релиза проекта



The image shows a code editor window with a file named `.gitignore`. The file contains a list of patterns to ignore, organized into sections with comments. The patterns include byte-compiled files, C extensions, distribution/packaging files, and PyInstaller-related files.

```
1 # Byte-compiled / optimized / DLL files
2 __pycache__/
3 *.py[cod]
4 *$py.class
5
6 # C extensions
7 *.so
8
9 # Distribution / packaging
10 .Python
11 build/
12 develop-eggs/
13 dist/
14 downloads/
15 eggs/
16 .eggs/
17 lib/
18 lib64/
19 parts/
20 sdist/
21 var/
22 wheels/
23 share/python-wheels/
24 *.egg-info/
25 .installed.cfg
26 *.egg
27 MANIFEST
28
29 # PyInstaller
30 # Usually these files are written by a python script from a template
31 # before PyInstaller builds the exe, so as to inject date/other infos into it.
32 *.manifest
```

Рисунок 5 – Часть `.gitignore`, созданного GitHub

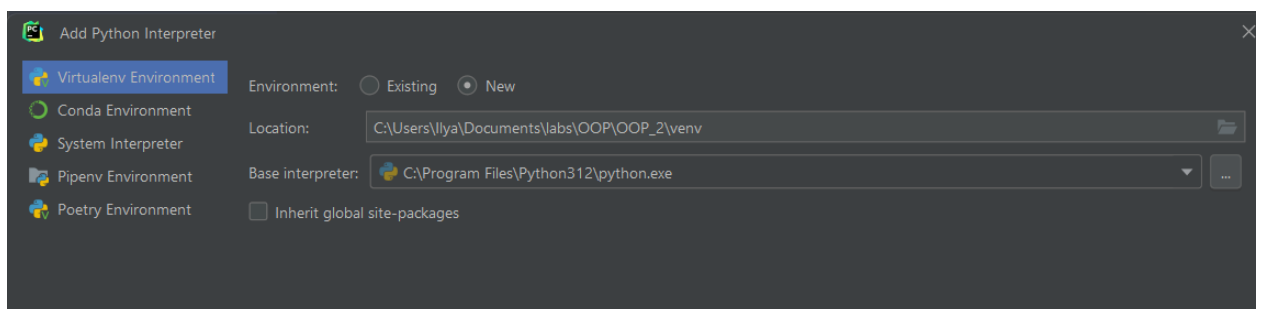


Рисунок 6 – Создание виртуального окружения

Try the redesigned packaging support in Python Packages tool window. [Go to tool window](#) ×

Package	Version	Latest version
pip	23.2.1	⬆ 24.3.1
setuptools	68.2.0	⬆ 75.3.0
wheel	0.41.2	⬆ 0.44.0

Рисунок 7 – Созданное окружение

доработка примеров из лабораторной работы:

Создадим простой модуль `calc.py`, содержащий функции для базовых арифметических операций. Затем пишем отдельный файл `test_calc.py` для проверки корректности работы этих функций, реализуя тестирование без использования какого-либо фреймворка.

Листинг 1 – Модуль `calc.py`

```
def add(a, b):
    return a + b

def sub(a, b):
    return a - b

def mul(a, b):
    return a * b

def div(a, b):
    return a / b
```

Листинг 3 – Содержимое `test_calc.py`

```
import calc

def test_add():
    if calc.add(1, 2) == 3:
        print("Test add(a, b) is OK")
    else:
        print("Test add(a, b) is Fail")

def test_sub():
    if calc.sub(4, 2) == 2:
        print("Test sub(a, b) is OK")
    else:
        print("Test sub(a, b) is Fail")

def test_mul():
```

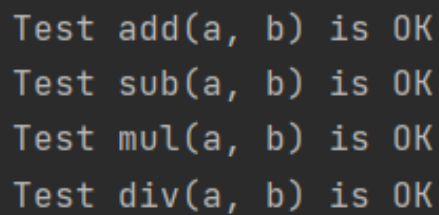
```

if calc.mul(2, 5) == 10:
    print("Test mul(a, b) is OK")
else:
    print("Test mul(a, b) is Fail")

def test_div():
    if calc.div(8, 4) == 2:
        print("Test div(a, b) is OK")
    else:
        print("Test div(a, b) is Fail")

test_add()
test_sub()
test_mul()
test_div()

```



```

Test add(a, b) is OK
Test sub(a, b) is OK
Test mul(a, b) is OK
Test div(a, b) is OK

```

Рисунок 8 – Пример вывода test_calc.py

Создадим тестовый файл `utest_calc.py`, используя `unittest` для проверки функций модуля `calc.py`. Такой подход позволяет структурировать тесты в классе и выводить более информативные результаты, что особенно полезно для масштабного тестирования.

Листинг 4 – Содержимое `utest_calc.py`

```

import unittest
import calc

class CalcTest(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)

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

```

```
(venv) PS C:\Users\Ilya\Documents\labs\00P\00P_2> python -m unittest utest_calc.py
....
-----
Ran 4 tests in 0.000s

OK
(venv) PS C:\Users\Ilya\Documents\labs\00P\00P_2>
```

Рисунок 9 – Запуск файла utest_calc.py

```
(venv) PS C:\Users\Ilya\Documents\labs\00P\00P_2> python -m unittest -v utest_calc.py
test_add (utest_calc.CalcTest.test_add) ... ok
test_div (utest_calc.CalcTest.test_div) ... ok
test_mul (utest_calc.CalcTest.test_mul) ... ok
test_sub (utest_calc.CalcTest.test_sub) ... ok

-----
Ran 4 tests in 0.000s

OK
(venv) PS C:\Users\Ilya\Documents\labs\00P\00P_2>
```

Рисунок 10 – Запуск файла utest_calc.py с запросом расширенной информации

Расширим тестовый файл utest_calc.py, чтобы продемонстрировать использование методов setUpClass, tearDownClass, setUp, и tearDown для выполнения подготовительных и завершающих операций перед и после тестов:

Листинг 5 – Расширенный файл utest_calc.py

```
import unittest
import calc

class CalcTest(unittest.TestCase):
    """Calc tests"""

    @classmethod
    def setUpClass(cls):
        """Set up for class"""
        print("setUpClass")
        print("=====")

    @classmethod
    def tearDownClass(cls):
        """Tear down for class"""
        print("=====")
```

```

        print("tearDownClass")

    def setUp(self):
        """Set up for test"""
        print("Set up for [" + self.shortDescription() + "]")

    def tearDown(self):
        """Tear down for test"""
        print("Tear down for [" + self.shortDescription() + "]")
        print("")

    def test_add(self):
        """Add operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        """Sub operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        """Mul operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        """Div operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.div(8, 4), 2)

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

```

```

(venv) PS C:\Users\Ilya\Documents\labs\OOP\OOP_2> python -m unittest -v utest_calc.py
setUpClass
=====
test_add (utest_calc.CalcTest.test_add)
Add operation test ... Set up for [Add operation test]
id: utest_calc.CalcTest.test_add
Tear down for [Sub operation test]

ok
=====
tearDownClass

-----
Ran 4 tests in 0.002s

OK
(venv) PS C:\Users\Ilya\Documents\labs\OOP\OOP_2>

```

Рисунок 11 – Запуск файла utest_calc.py

Создадим модуль `test_runner.py`, который с помощью `TestSuite` объединит тесты из двух классов `CalcBasicTests` и `CalcExTests` для тестирования функций модуля `calc`:

Листинг 6 – Измененный модуль `calc.py`

```
def add(a, b):
    return a + b

def sub(a, b):
    return a - b

def mul(a, b):
    return a * b

def div(a, b):
    return a / b

def sqrt(a):
    return a ** 0.5

def pow(a, b):
    return a ** b
```

Листинг 7 – Модуль `calc_tests.py`

```
import unittest
import calc

class CalcBasicTests(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)

class CalcExTests(unittest.TestCase):
    def test_sqrt(self):
        self.assertEqual(calc.sqrt(4), 2)

    def test_pow(self):
        self.assertEqual(calc.pow(3, 3), 27)
```

Листинг 8 – Модуль `test_runner.py`

```
import unittest
import calc_tests

calcTestSuite = unittest.TestSuite()
calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcBasicTests))
calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcExTests))
```

```
print("count of tests: " + str(calcTestSuite.countTestCases()) + "\n")
runner = unittest.TextTestRunner(verbosity=2)
runner.run(calcTestSuite)
```

```
count of tests: 6
```

```
test_add (calc_tests.CalcBasicTests.test_add) ... ok
test_div (calc_tests.CalcBasicTests.test_div) ... ok
test_mul (calc_tests.CalcBasicTests.test_mul) ... ok
test_sub (calc_tests.CalcBasicTests.test_sub) ... ok
test_pow (calc_tests.CalcExTests.test_pow) ... ok
test_sqrt (calc_tests.CalcExTests.test_sqrt) ... ok
```

```
-----
Ran 6 tests in 0.000s
```

```
OK
```

Рисунок 12 – Запуск файла test_runner.py

Создадим тестовый модуль, в котором пропустим выполнение всех тестов из класса CalcExTests, используя декоратор `@unittest.skip`.

Листинг 9 –Модуль calc_tests.py

```
import unittest
import calc

class CalcBasicTests(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)

@unittest.skip("Skip CalcExTests")
class CalcExTests(unittest.TestCase):
    def test_sqrt(self):
        self.assertEqual(calc.sqrt(4), 2)

    def test_pow(self):
        self.assertEqual(calc.pow(3, 3), 27)
```

```

Ran 6 tests in 0.003s

OK (skipped=2)

Skipped: Skip CalcExTests

Skipped: Skip CalcExTests

Process finished with exit code 0

```

Рисунок 13 – Запуск файла calc_tests.py

ольем ветки develop и main/master и отправим изменения на удаленный репозиторий:

Ссылка: https://github.com/daxstrong/OOP_2.git

```

(venv) PS C:\Users\Ilya\Documents\labs\OOP\OOP_2> git log --oneline
93ebc8d (HEAD -> develop) final changes
45b6546 (origin/main, origin/HEAD, main) Update README.md
3782ce7 Update README.md
e95a6e7 Initial commit

```

Рисунок 14 – История коммитов

```

(venv) PS C:\Users\Ilya\Documents\labs\OOP\OOP_2> git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
(venv) PS C:\Users\Ilya\Documents\labs\OOP\OOP_2> git merge develop
Updating 45b6546..93ebc8d
Fast-forward
 .idea/.gitignore          | 8 +++
 .idea/OOP_2.iml           | 10 ++++
 .idea/inspectionProfiles/profiles_settings.xml | 6 +++
 .idea/misc.xml            | 4 ++
 .idea/modules.xml         | 8 ++++
 .idea/vcs.xml             | 6 +++
 calc.py                   | 22 ++++++++
 calc_tests.py             | 25 ++++++++
 test_calc.py              | 35 ++++++++
 test_runner.py            | 10 ++++
 utest_calc.py             | 51 ++++++++
11 files changed, 185 insertions(+)
create mode 100644 .idea/.gitignore
create mode 100644 .idea/OOP_2.iml
create mode 100644 .idea/inspectionProfiles/profiles_settings.xml
create mode 100644 .idea/misc.xml
create mode 100644 .idea/modules.xml
create mode 100644 .idea/vcs.xml
create mode 100644 calc.py
create mode 100644 calc_tests.py
create mode 100644 test_calc.py
create mode 100644 test_runner.py
create mode 100644 utest_calc.py
(venv) PS C:\Users\Ilya\Documents\labs\OOP\OOP_2>

```

Рисунок 15 – Слияние веток main и develop

```
(venv) PS C:\Users\Ilya\Documents\labs\OOP\OOP_2> git push origin main
Enumerating objects: 16, done.
Counting objects: 100% (16/16), done.
Delta compression using up to 12 threads
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 2.71 KiB | 555.00 KiB/s, done.
Total 15 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/daxstrong/OOP\_2.git
 45b6546..93ebc8d  main -> main
(venv) PS C:\Users\Ilya\Documents\labs\OOP\OOP_2> █
```

Рисунок 16 – Отправка изменений на удаленный репозиторий

Вывод: приобрели полезные навыки, связанные с работой с классами и их экземплярами – объектами. Решили задачи при помощи новых знаний о возможностях ООП в Python. При написании программ использовался язык программирования Python версии 3.10.

Ответы на контрольные вопросы:

для чего используется автономное тестирование?

Автономное тестирование используется для проверки функциональности кода в изолированной среде, чтобы обеспечить его корректность без зависимости от других частей системы. Это помогает выявить ошибки на ранних этапах разработки и повышает надёжность программы.

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

Наиболее распространённые фреймворки для автономного тестирования в Python — это unittest, pytest и nose.

Какие существуют основные структурные единицы модуля unittest?

Основные структурные единицы unittest включают классы TestCase для определения тестов, TestSuite для их группировки, TestLoader для загрузки тестов, TestRunner для их запуска, и TestResult для хранения результатов тестирования.

Какие существуют способы запуска тестов unittest?

Тесты unittest можно запускать через командную строку, используя

каково назначение класса TestCase?

Класс TestCase используется для создания отдельных тестов. Каждый метод в классе TestCase, начинающийся с test, представляет отдельный тест, проверяющий определённое поведение программы.

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

Метод setUp() выполняется перед каждым тестом для подготовки окружения, а метод tearDown() — после каждого теста для очистки или завершения настроек.

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

Для проверки условий и генерации ошибок используются методы `assert`, которые проверяют различные условия и вызывают ошибку, если условия не выполнены.

Какие методы класса `TestCase` позволяют собирать информацию о самом тесте?

Методы `shortDescription()` и `id()` позволяют собирать информацию о тесте, такую как его описание и уникальный идентификатор.

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

Класс `TestSuite` используется для объединения нескольких тестов в одну группу, которую можно запускать вместе. Тесты загружаются в `TestSuite` с помощью методов `addTest()` и `addTests()`, а также с помощью `TestLoader`.

Каково назначение класса `TestResult`?

Класс `TestResult` хранит результаты выполнения тестов, фиксирует успешные и проваленные тесты, а также сохраняет информацию о возникших ошибках и пропущенных тестах.

Для чего может понадобиться пропуск отдельных тестов?

Пропуск отдельных тестов может понадобиться, если тесты временно не актуальны, требуют особой настройки или зависят от функций, находящихся в разработке.

Как выполняется безусловный и условный пропуск тестов? Как выполнить пропуск класса тестов?

Безусловный пропуск тестов осуществляется с помощью декоратора пропуска всего класса используется декоратор `@unittest.skip` перед объявлением класса.

Средства поддержки тестов `unittest` в PyCharm. Обобщённый алгоритм тестирования с помощью PyCharm.

PyCharm предоставляет встроенные средства для запуска и отладки тестов unittest. Обобщённый алгоритм таков: сначала написать тесты в отдельном модуле и открыть этот модуль в PyCharm. Затем запустить тесты с помощью кнопки запуска или через контекстное меню. Результаты тестирования будут отображены в специальной панели, где можно увидеть список успешных и проваленных тестов, сообщения об ошибках и трассировку ошибок.