

Лабораторная работа 2.22

Тема: Тестирование в Python [unittest]

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

Ссылка на GitHub: https://github.com/RomanGorchakov/Py3_2

Порядок выполнения работы

1. Создаём аккаунт в GitHub. Затем создаём новый общедоступный репозиторий, в котором будет использована лицензия MIT и язык программирования Python.

Create a new repository



A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Owner * Repository name *
RomanGorchakov / Test
Test is available.

Great repository names are short and memorable. Need inspiration? How about [automatic-octo-chainsaw](#) ?

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.](#)

 You are creating a public repository in your personal account.

Create repository

 **Test** Public

 Pin  Unwatch 1

 main  1 branch  0 tags

[Go to file](#) [Add file](#) [Code](#)

 **RomanGorchakov** Initial commit 87a4818 now  1 commit

 .gitignore Initial commit now


















 LICENSE Initial commit now

Help people interested in this repository understand your project by adding a README.

[Add a README](#)

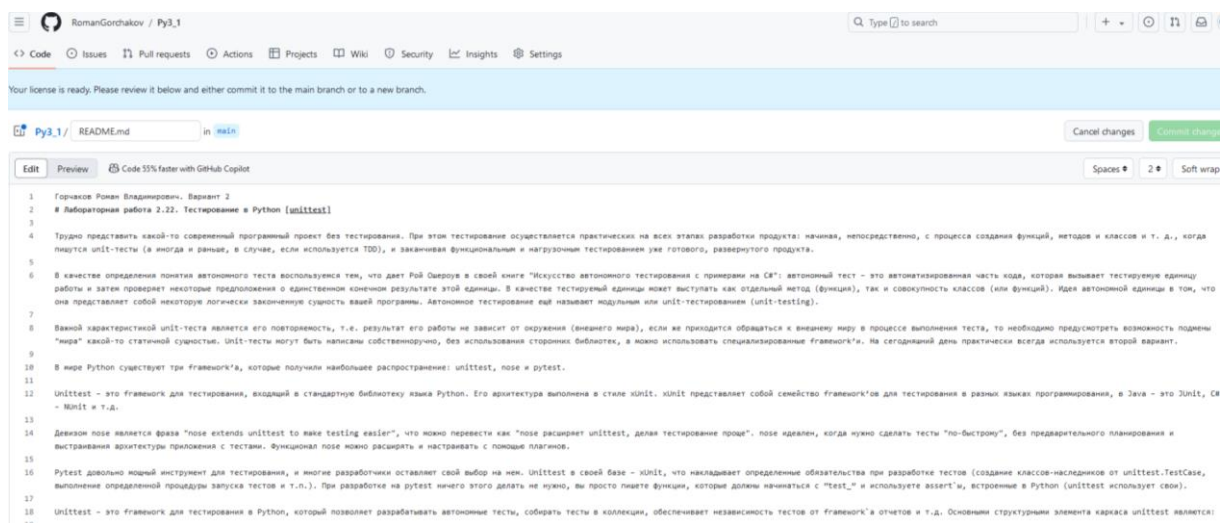
 © 2023 GitHub, Inc. [Terms](#) [Privacy](#) [Security](#) [Status](#) [Docs](#) [Contact GitHub](#) [Pricing](#) [API](#) [Training](#)

2. Теперь необходимо дополнить файл `.gitignore` с необходимыми правилами для языка программирования Python. Для этого переходим по ссылке «<https://github.com/github/gitignore>» и скачиваем оттуда файл «Python.gitignore».

	Phalcon.gitignore	Remove trailing asterisks in Phalcon rules	10 years ago
	PlayFramework.gitignore	Added /project/project to PlayFramework.gitignore	7 years ago
	Plone.gitignore	Covered by global vim template	10 years ago
	Prestashop.gitignore	Update for Prestashop 1.7 (#3261)	3 years ago
	Processing.gitignore	Ignore transpiled .java and .class files (#3016)	4 years ago
	PureScript.gitignore	Update PureScript adding .spago (#3278)	3 years ago
	Python.gitignore	Update Python.gitignore	last year
	Qooxdoo.gitignore	Add gitignore for qooxdoo apps	13 years ago
	Qt.gitignore	Remove trailing whitespace	2 years ago
	R.gitignore	Merge pull request #3792 from jl5000/patch-1	2 years ago
	README.md	Merge pull request #3854 from AnilSeervi/patch-1	2 years ago
	ROS.gitignore	Added ignore for files created by catkin_make_isolated	6 years ago
	Racket.gitignore	Update Racket.gitignore	2 years ago
	Rails.gitignore	Ignore Rails .env according recommendations	2 years ago
	Raku.gitignore	Changes the name of Perl 6 to Raku (#3312)	3 years ago
	RhodesRhomobile.gitignore	Add Rhodes mobile application framework gitignore	13 years ago
	Ruby.gitignore	Ruby: ignore RuboCop remote inherited config files (#3197)	4 years ago

1	# Byte-compiled / optimized / DLL files	1	# Byte-compiled / optimized / DLL files
2	__pycache__	2	__pycache__
3	*.py[cod]	3	*.py[cod]
4	*\$py.class	4	*\$py.class
5		5	
6	# C extensions	6	# C extensions
7	*.so	7	*.so
8		8	
9	# Distribution / packaging	9	# Distribution / packaging
10	.Python	10	.Python
11	build/	11	build/
12	develop-eggs/	12	develop-eggs/
13	dist/	13	dist/
14	downloads/	14	downloads/
15	eggs/	15	eggs/
16	.eggs/	16	.eggs/
17	lib/	17	lib/
18	lib64/	18	lib64/
19	parts/	19	parts/
20	sdist/	20	sdist/
21	var/	21	var/
22	wheels/	22	wheels/
23	share/python-wheels/	23	share/python-wheels/
24	*.egg-info/	24	*.egg-info/
25	.installed.cfg	25	.installed.cfg
26	*.egg	26	*.egg
27	MANIFEST	27	MANIFEST
28		28	

3. Теперь создаём файл «README.md», где вносим информацию о своей группе и ФИО. Сохраняем набранный текст через кнопку «Commit changes».



4. В окне «Codespace» выбираем опцию «Create codespace on main». Откроется терминал, куда мы введём команду «git clone», чтобы клонировать свой

репозиторий. После этого организуем репозиторий в соответствии с моделью ветвления Git-flow. Для этого введём в терминал команды: «git checkout -b develop» для создания ветки разработки; «git branch feature_branch» для создания ветки функций; «git branch release/1.0.0» для создания ветки релиза; «git checkout main» и «git branch hotfix» для создания веток hotfix. Устанавливаем библиотеки isort, black и flake8 и создаём файлы .pre-commit-config.yaml и environment.yml.

```
ПРОБЛЕМЫ    ВЫХОДНЫЕ ДАННЫЕ    КОНСОЛЬ ОТЛАДКИ    ТЕРМИНАЛ    ПОРТЫ    КОММЕНТАРИИ

● @RomanGorchakov → /workspaces/Py3_2 (main) $ git checkout -b develop
  Switched to a new branch 'develop'
● @RomanGorchakov → /workspaces/Py3_2 (develop) $ git branch feature_branch
● @RomanGorchakov → /workspaces/Py3_2 (develop) $ git branch release/1.0.0
● @RomanGorchakov → /workspaces/Py3_2 (develop) $ git checkout main
  Switched to branch 'main'
  Your branch is up to date with 'origin/main'.
● @RomanGorchakov → /workspaces/Py3_2 (main) $ git branch hotfix
● @RomanGorchakov → /workspaces/Py3_2 (main) $ git checkout develop
  Switched to branch 'develop'
○ @RomanGorchakov → /workspaces/Py3_2 (develop) $ █

● @RomanGorchakov → /workspaces/Py3_2 (develop) $ pip install black
  Collecting black
    Downloading black-24.10.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.manylinux_2_28_x86_64.whl.metadata (79 kB)
  Collecting click>=8.0.0 (from black)
    Downloading click-8.1.7-py3-none-any.whl.metadata (3.0 kB)
  Collecting mypy_extensions>=0.4.3 (from black)
    Downloading mypy_extensions-1.0.0-py3-none-any.whl.metadata (1.1 kB)
  Requirement already satisfied: packaging>=22.0 in /home/codespace/.local/lib/python3.12/site-packages (from black) (24.1)
  Collecting pathspec>=0.9.0 (from black)
    Downloading pathspec-0.12.1-py3-none-any.whl.metadata (21 kB)
  Requirement already satisfied: platformdirs>=2 in /home/codespace/.local/lib/python3.12/site-packages (from black) (4.3.6)
  Downloading black-24.10.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.manylinux_2_28_x86_64.whl (1.8 MB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 20.1 MB/s eta 0:00:00
  Downloading click-8.1.7-py3-none-any.whl (97 kB)
  Downloading mypy_extensions-1.0.0-py3-none-any.whl (4.7 kB)
  Downloading pathspec-0.12.1-py3-none-any.whl (31 kB)
  Installing collected packages: pathspec, mypy_extensions, click, black
  Successfully installed black-24.10.0 click-8.1.7 mypy_extensions-1.0.0 pathspec-0.12.1
● @RomanGorchakov → /workspaces/Py3_2 (develop) $ pip install flake8
  Collecting flake8
    Downloading flake8-7.1.1-py2.py3-none-any.whl.metadata (3.8 kB)
  Collecting mccabe<0.8.0,>=0.7.0 (from flake8)
    Downloading mccabe-0.7.0-py2.py3-none-any.whl.metadata (5.0 kB)
  Collecting pycodestyle<2.13.0,>=2.12.0 (from flake8)
    Downloading pycodestyle-2.12.1-py2.py3-none-any.whl.metadata (4.5 kB)
  Collecting pyflakes<3.3.0,>=3.2.0 (from flake8)
    Downloading pyflakes-3.2.0-py2.py3-none-any.whl.metadata (3.5 kB)
  Downloading flake8-7.1.1-py2.py3-none-any.whl (57 kB)
  Downloading mccabe-0.7.0-py2.py3-none-any.whl (7.3 kB)
  Downloading pycodestyle-2.12.1-py2.py3-none-any.whl (31 kB)
  Downloading pyflakes-3.2.0-py2.py3-none-any.whl (62 kB)
  Installing collected packages: pyflakes, pycodestyle, mccabe, flake8
  Successfully installed flake8-7.1.1 mccabe-0.7.0 pycodestyle-2.12.1 pyflakes-3.2.0
● @RomanGorchakov → /workspaces/Py3_2 (develop) $ pre-commit sample-config > .pre-commit-config.yaml
● @RomanGorchakov → /workspaces/Py3_2 (develop) $ conda env export > environment.yml
○ @RomanGorchakov → /workspaces/Py3_2 (develop) $ █
```

5. Создаём файл «individual.py», в котором нужно добавить тесты с использованием модуля unittest, проверяющие операции по работе с базой данных. Написать программу, выполняющую следующие действия: ввод с клавиатуры данных в список, состоящий из словарей заданной структуры; записи должны быть упорядочены по возрастанию номера рейса; вывод на экран номеров рейсов и типов самолетов, вылетающих в пункт назначения, название которого совпало с названием, введенным с клавиатуры; если таких рейсов нет, выдать на дисплей соответствующее сообщение. Необходимо также проследить за тем, чтобы файлы генерируемый этой программой не попадали в репозиторий лабораторной работы.

```
Командная строка
D:\3 курс\Объектно-ориентированное программирование\2\Индивидуальное задание>python individual.py
...
Ran 3 tests in 0.761s
OK
D:\3 курс\Объектно-ориентированное программирование\2\Индивидуальное задание>
```

6. Выполняем коммит файлов в репозиторий Git в ветку разработки, сливаем её с веткой main и отправляем изменения на сервер GitHub.

```
@RomanGorchakov → /workspaces/Py21 (develop) $ git add .
@RomanGorchakov → /workspaces/Py21 (develop) $ git commit -m "SQL databases"
[develop cc32d83] SQL databases
5 files changed, 617 insertions(+)
create mode 100644 .pre-commit-config.yaml
create mode 100644 environment.yml
create mode 100644 "\320\230\320\275\320\264\320\270\320\262\320\270\320\264\321\203\320\260\320\273\321\214\320\275\320\276\320\265 \320\267\320\260\320\264\320\260\320\275\320\270\320\265\individual.py"
create mode 100644 "\320\236\321\202\321\207\321\221\321\202\320\233\320\2402.19_\320\223\320\276\321\200\321\207\320\260\320\272\320\276\320\262\320\240\320\222.pdf"
create mode 100644 "\320\237\321\200\320\270\320\274\320\265\321\200\example.py"
@RomanGorchakov → /workspaces/Py21 (develop) $ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
@RomanGorchakov → /workspaces/Py21 (main) $ git merge develop
Updating 41dd812..cc32d83
Fast-forward
 .pre-commit-config.yaml
 environment.yml
 .../individual.py
 .../\320\223\320\2402.19_\320\223\320\276\321\200\321\207\320\260\320\272\320\276\320\262\320\240\320\222.pdf"
 "\320\237\321\200\320\270\320\265\321\200\example.py"
5 files changed, 617 insertions(+)
create mode 100644 .pre-commit-config.yaml
create mode 100644 environment.yml
create mode 100644 "\320\230\320\275\320\264\320\270\320\262\320\270\320\264\321\203\320\260\320\273\321\214\320\275\320\276\320\265 \320\267\320\260\320\264\320\260\320\275\320\270\320\265\individual.py"
create mode 100644 "\320\236\321\202\321\207\321\221\321\202\320\233\320\2402.19_\320\223\320\276\321\200\321\207\320\260\320\272\320\276\320\262\320\240\320\222.pdf"
create mode 100644 "\320\237\321\200\320\270\320\274\320\265\321\200\example.py"
@RomanGorchakov → /workspaces/Py21 (main) $ git push -u
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 2 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (10/10), 554.90 KiB | 19.82 MiB/s, done.
Total 10 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/RomanGorchakov/Py21
 41dd812..cc32d83 main -> main
branch 'main' set up to track 'origin/main'.
@RomanGorchakov → /workspaces/Py21 (main) $
```

main
1 Branch
0 Tags

Add file
Code

RomanGorchakov SQL databases		cc32d83 · 1 minute ago	5 Commits
Индивидуальное задание	SQL databases	1 minute ago	
Отчёт	SQL databases	1 minute ago	
Пример	SQL databases	1 minute ago	
.gitignore	Create .gitignore	3 days ago	
.pre-commit-config.yaml	SQL databases	1 minute ago	
LICENSE	Create LICENSE	3 days ago	
README.md	Update README.md	14 hours ago	
environment.yml	SQL databases	1 minute ago	

README
MIT license

Горчаков Роман Владимирович. Вариант 4

Лабораторная работа 2.21. Взаимодействие с базами данных SQLite3 с помощью языка программирования Python.

Сами по себе СУБД редко используются для работы с базами данных. В том смысле, что в реальных проектах связи БД + СУБД бывает недостаточно. Обычно с СУБД работают через какой-либо язык программирования. Это позволяет более гибко принимать запросы, обрабатывать ответы перед передачей их куда-либо далее. Ведь у императивного, а не декларативного как SQL, языка программирования средств для работы с данными больше, да и логика богаче.

Чтобы использовать SQLite3 в Python, прежде всего, вам нужно будет импортировать модуль sqlite3, а затем создать объект соединения, который соединит нас с базой данных и позволит нам выполнять операторы SQL. Объект соединения создается с помощью функции connect(). Будет создан новый файл под названием

Контрольные вопросы

1. Для чего используется автономное тестирование?

Автономный тест — автоматизированная часть кода, которая вызывает тестируемую единицу работы и затем проверяет некоторые предположения о единственном конечном результате этой единицы. В качестве тестируемой единицы может выступать как отдельный метод (функция), так и совокупность классов (или функций). Идея автономной единицы в том, что она представляет собой некоторую логически законченную сущность вашей программы.

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

В мире Python существуют три фреймворка, которые получили наибольшее распространение: unittest, nose и pytest.

3. Какие существуют основные структурные единицы модуля unittest?

Основными структурными элементами каркаса unittest являются:

1) 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 необходимых серверов и т.п.;

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

3) Test suite – коллекция тестов, которая может в себя включать как отдельные test case'ы, так и целые коллекции. Коллекции используются с целью объединения тестов для совместного запуска;

4) Test runner – компонент, который 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. Вся работа по написанию тестов заключается в разработке отдельных тестов в рамках test case'ов, сборе их в модули и запуске, если нужно объединить несколько test case'ов, для их совместного запуска, они помещаются в test suite'ы, которые помимо test case'ов могут содержать другие test suite'ы.

4. Какие существуют способы запуска тестов unittest?

Запуск тестов можно сделать как из командной строки, так и с помощью графического интерфейса пользователя (GUI). CLI позволяет запускать тесты из целого модуля, класса или даже обращаться к конкретному тесту. Запуск всех

тестов в модуле: `python -m unittest <имя модуля>`. Если осуществить запуск без указания модуля с тестами, будет запущен Test Discovery.

5. Каково назначение класса TestCase?

Основным строительным элементом при написании тестов с использованием unittest является 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?

К методам класса TestCase, выполняющимся при запуске и завершении работы тестов, относятся:

1) `setUp()`. Метод вызывается перед запуском теста. Как правило, используется для подготовки окружения для теста;

2) `tearDown()`. Метод вызывается после завершения работы теста. Используется для "приборки" за тестом;

3) `setUpClass()`. Метод действует на уровне класса, т.е. выполняется перед запуском тестов класса. При этом синтаксис требует наличие декоратора `@classmethod`;

4) `tearDownClass()`. Запускается после выполнения всех методов класса, требует наличия декоратора `@classmethod`;

5) `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?

К методам класса TestCase, используемым для проверки условий и генерации ошибок, относятся:

1) `assertEqual(a, b)` - `a == b`

2) `assertNotEqual(a, b)` - `a != b`

3) `assertTrue(x)` - `bool(x) is True`

4) `assertFalse(x)` - `bool(x) is False`

5) `assertIs(a, b)` - `a is b`

6) `assertIsNot(a, b)` - `a is not b`

7) `assertIsNone(x)` - `x is None`

8) `assertIsNotNone(x)` - `x is not None`

9) `assertIn(a, b)` - `a in b`

10) `assertNotIn(a, b)` - `a not in b`

11) `assertIsInstance(a, b)` - `isinstance(a, b)`

12) `assertNotIsInstance(a, b)` - `not isinstance(a, b)`

13) `assertRaises(exc, fun, *args, **kws)` - Функция `fun(*args, **kws)` вызывает исключение `exc`

14) `assertRaisesRegex(exc, r, fun, *args, **kws)` - Функция `fun(*args, **kws)` вызывает исключение `exc`, сообщение которого совпадает с регулярным выражением `r`

15) `assertWarns(warn, fun, *args, **kws)` - Функция `fun(*args, **kws)` выдает сообщение `warn`

16) `assertWarnsRegex(warn, r, fun, *args, **kws)` - Функция `fun(*args, **kws)` выдает сообщение `warn` и оно совпадает с регулярным выражением `r`

17) `assertAlmostEqual(a, b)` - `round(a-b, 7) == 0`

18) `assertNotAlmostEqual(a, b)` - `round(a-b, 7) != 0`

19) `assertGreater(a, b)` - `a > b`

20) `assertGreaterEqual(a, b)` - `a >= b`

21) `assertLess(a, b)` - `a < b`

22) `assertLessEqual(a, b)` - `a <= b`

23) `assertRegex(s, r)` - `r.search(s)`

24) `assertNotRegex(s, r)` - `not r.search(s)`

25) `assertCountEqual(a, b)` - `a` и `b` имеют одинаковые элементы (порядок неважен)

26) `assertMultiLineEqual(a, b)` - строки (strings)

27) `assertSequenceEqual(a, b)` - последовательности (sequences)

28) `assertListEqual(a, b)` - списки (lists)

29) `assertTupleEqual(a, b)` - кортежи (tuples)

30) `assertSetEqual(a, b)` - множества или неизменяемые множества (frozensets)

31) `assertDictEqual(a, b)` - словари (dicts)

32) `fail(msg=None)` - ошибка в тесте.

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

К методам класса `TestCase`, позволяющим собирать информацию о самом тесте, относятся:

1) `countTestCases()`. Возвращает количество тестов в объекте класса-наследника от `TestCase`;

2) `d()`. Возвращает строковый идентификатор теста. Как правило, это полное имя метода, включающее имя модуля и имя класса;

3) `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.

10. Каково назначение класса `TestResult`?

Класс `TestResult` используется для сбора информации о результатах прохождения тестов. Объекты класса `TextTestRunner` используются для запуска тестов. Среди параметров, которые передаются конструктору класса, можно выделить `verbosity`, по умолчанию он равен 1, если создать объект с `verbosity=2`, то будем получать расширенную информацию о результатах прохождения тестов. Для запуска тестов используется метод `run()`, которому в качестве аргумента передается класс-наследник от `TestCase` или группа (`TestSuite`).

11. Для чего может понадобиться пропуск отдельных тестов?

Пропуск отдельных тестов может понадобиться в разных ситуациях, например:

- 1) Устаревшие или неправильные тесты. Их можно маркировать, добавив сообщение о неисправности кода.
- 2) Платформозависимые тесты. Например, ряд тестов могут выполняться только под операционной системой MS Windows.
- 3) Тесты, зависящие от версии программы. Например, если в новой версии уже не поддерживается часть методов.
- 4) Тесты, для которых недоступен ресурс, необходимый для настройки. Это полезно, когда ресурс, который нужно настроить, недоступен.

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

Для безусловного пропуска теста воспользуемся декоратором `@unittest.skip(reason)`, который пишется перед тестом.

Для условного пропуска тестов применяются следующие декораторы:

- 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.

Для пропуска классов используется декоратор `@unittest.skip(reason)`, который записывается перед объявлением класса.

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

Некоторые средства для поддержки тестов в PyCharm:

1. Работа с разными средами тестирования. По умолчанию используется `unittest`, но PyCharm поддерживает и другие среды, например `pytest`, `nose`, `doctest`, `tox` и `trialc`. Чтобы включить `pytest` для проекта, нужно открыть диалоговое окно настроек, перейти в раздел Tools, выбрать Python Integrated Tools и выбрать `pytest` в поле «Запуск теста по умолчанию».

2. Создание тестового класса непосредственно из исходного кода вместе с необходимыми методами тестирования. Можно переключать тестовые классы и исходный код с помощью ярлыка, запускать несколько тестов, просматривать статистику для каждого теста и экспортировать результаты тестов в файл.

3. Покрывание кода тестами. Позволяет анализировать код и понимать, какие области кода покрыты тестами, а какие требуют большего тестирования.

Обобщённый алгоритм проведения тестирования с помощью PyCharm:

Настройка тестовой среды. В окне «Пакеты» нужно найти и установить `pytest`. При выборе интерпретатора Python PyCharm автоматически обнаружит установленный тестовый исполнитель, по умолчанию используется `unittest`. Чтобы настроить другой, нужно открыть настройки проекта, выбрать «Инструменты», затем «Интегрированные инструменты Python» и нужный тестовый исполнитель из списка «По умолчанию».

Создание тестов. Нужно открыть файл, нажать правой кнопкой мыши на имя класса, выбрать «Перейти» и «Тест». Во всплывающем окне предложить создать новый тест.

Запуск тестов. Для запуска всего файла нужно нажать правой кнопкой мыши и выбрать «Запустить тесты Python». Также можно выбрать текущий файл во выпадающем списке конфигураций запуска. В открывшемся окне запуска будет указано, сколько тестов прошло или не прошло.

Отладка неудачных тестов. Нужно выбрать неудачный тест в левой панели окна запуска, а в правой – номер строки, где произошла ошибка. По этому номеру в редакторе установить точку останова. Затем запустить сессию отладки: для этого нужно на фоне редактора нажать правой кнопкой мыши на метод и выбрать «Отладка» из контекстного меню.