

# Программирование на Python



Базовый курс. Дополнительные материалы

# Python. Импорт стандартных модулей

Импорт модулей:

- Простой импорт: **import** <модуль>
- Использование псевдонима: **import** <модуль> **as** <псевдоним>
- Выборочный импорт: **from** <модуль> **import** <объект>

Поддерживается, но **не рекомендуется** импортирование всех объектов модуля: **from** <модуль> **import \***

Согласно PEP8:

- Каждый импорт помещается на отдельной строке
- Все импорты помещаются в начало файла в порядке, вначале выполняется импорт стандартных модулей, затем сторонних, затем модулей проекта

В процессе выполнения программы есть риск (возможность) объявить объект с таким же названием, как импортируемый объект.

```
import datetime  
  
print(datetime.datetime.now())
```

✓ 0.0s

2023-09-05 22:21:00.855918

```
import datetime as dt  
  
print(dt.datetime.now())
```

✓ 0.0s

2023-09-05 22:21:19.637830

```
from datetime import timedelta as tdelta, datetime  
  
print(datetime.now())  
print(tdelta.days)
```

✓ 0.0s

3-09-05 22:28:47.714270  
member 'days' of 'datetime.timedelta' objects>

```
print(dt.datetime.now())
```

```
dt = 'строка'  
print(dt)
```

```
print(dt.datetime.now())
```

⊗ 0.4s

2023-09-05 22:40:03.035655  
строка

```
-----  
AttributeError  
Cell In[12], line 6  
      3 dt = 'строка'  
      4 print(dt)  
----> 6 print(dt.datetime.now())
```

Traceback (most recent call last):

AttributeError: 'str' object has no attribute 'datetime'

# Python. Импорт пользовательских модулей

Пользовательский модуль может представлять собой файл (.py).

Импорт пройдет без проблем, если файл находится в известных интерпретатору директориях для импорта. Список этих директорий:

- **sys.path** (для работы нужно вначале выполнить `import sys`)

Можно добавить директорию с модулем в список **sys.path**.

Если файл модуля находится в поддиректории проекта:

- **import** <поддиректория>.<название модуля>

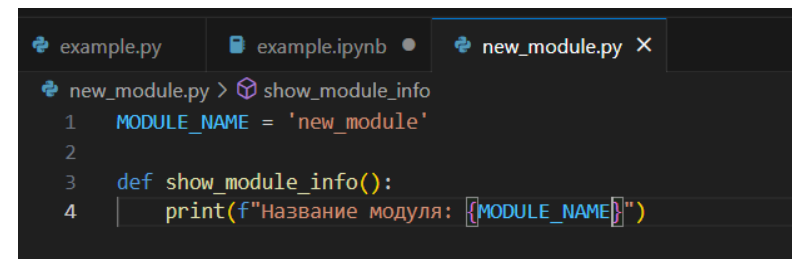
Если файл модуля находится в родительской директории проекта:

- **import** ..<название модуля>

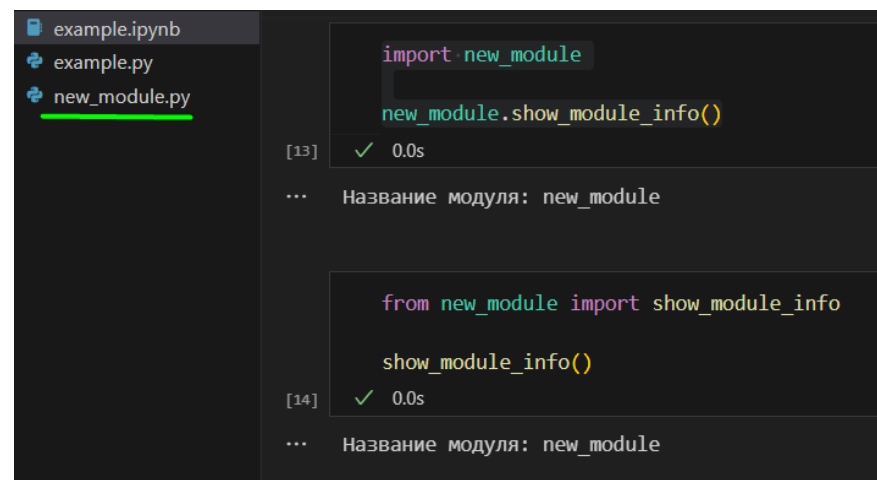
В файле импортируемого модуля для кода, который не должен выполняться при импорте необходимо добавлять проверку имени исполняемого модуля:

- **if \_\_name\_\_ == "\_\_main\_\_":** <исполняемый код>

Импорт модуля в проекте производится только один раз, если нужно импортировать модуль повторно следует использовать функцию **reload** библиотеки **importlib**.



```
example.py  example.ipynb  new_module.py X
new_module.py > show_module_info
1  MODULE_NAME = 'new_module'
2
3  def show_module_info():
4      print(f"Название модуля: {MODULE_NAME}")
```



```
example.ipynb
example.py
new_module.py
import new_module
new_module.show_module_info()
[13] ✓ 0.0s
... Название модуля: new_module

from new_module import show_module_info
show_module_info()
[14] ✓ 0.0s
... Название модуля: new_module
```

# Python. Импорт внешних модулей

- Проверка установленных внешних модулей: `$ pip list` или `$ pip freeze`
- Установка нового внешнего модуля: `$ pip install <имя модуля>==<версия модуля>`
- Установка внешних модулей обычно производится из репозитория **pypi.org**
- Сохранение списка внешних модулей в файл: `$ pip freeze > <имя файла>`
- Пакетная установка внешних модулей из файла: `$ pip install -r <имя файла>`
- Обновление текущей версии модуля: `$ pip install <имя модуля> --upgrade`
- Удаление модуля: `$ pip uninstall <имя модуля>`

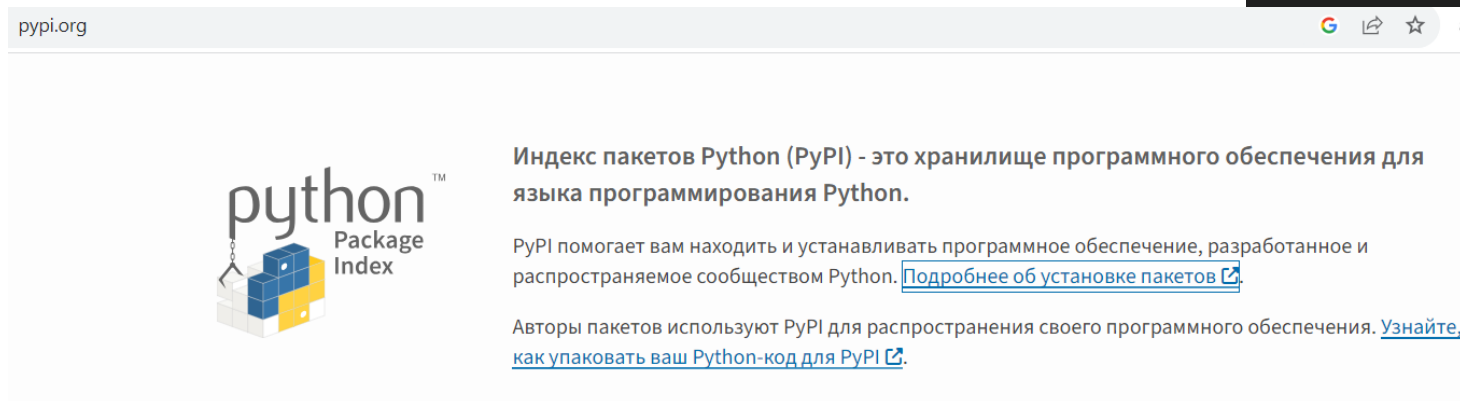
Для исполнения системных команд **из среды Jupyter Notebook** необходимо **добавлять к команде «!»**: `!pip install`

```
!pip list
✓ 2.5s
```

Package	Version
anyio	3.7.0
argon2-cffi	21.3.0
argon2-cffi-bindings	21.2.0
arrow	1.2.3
asttokens	2.2.1
async-lru	2.0.2
attrs	23.1.0
Babel	2.12.1

```
!pip install plotly==5.16.1
✓ 1m 55.2s
```

Collecting plotly==5.16.1  
Downloading plotly-5.16.1-py2.py3-none-any.whl (15.6 MB)  
0.0/15.6 MB ? eta -:  
0.0/15.6 MB ? eta -:



```
!pip freeze > requirements.txt
✓ 1.3s
```

```
!pip freeze
✓ 1.2s
```

```
anyio==3.7.0
argon2-cffi==21.3.0
argon2-cffi-bindings==21.2.0
arrow==1.2.3
asttokens==2.2.1
```

# Python. Задание 1.

1. Создайте в IDE новый проект `python_basic`
2. Создайте в корневом каталоге проекта файл `test.py`
3. Импортируйте одну из стандартных библиотек с использованием псевдонима, вызовите любую функцию из этой библиотеки. Кому лень искать, можно импортировать, например, модуль `os` и вызвать функцию `os.listdir()`
4. Импортируйте только предыдущую функцию из пакета, убедитесь в ее работе
5. Переопределите название предыдущей функции, например, числом и вызовите ее снова
6. Проверьте установленные внешние пакеты при помощи команды `freeze` или `show`
7. Найдите среди пакетов модуль `pandas` (`| grep pandas`). Или другой, еще не установленный
8. Загуглите текущие версии этого пакета и установите не самую новую версию (`install`)
9. Обновите текущую версию пакета до последней
10. Импортируйте установленный пакет и вызовите из него любую функцию

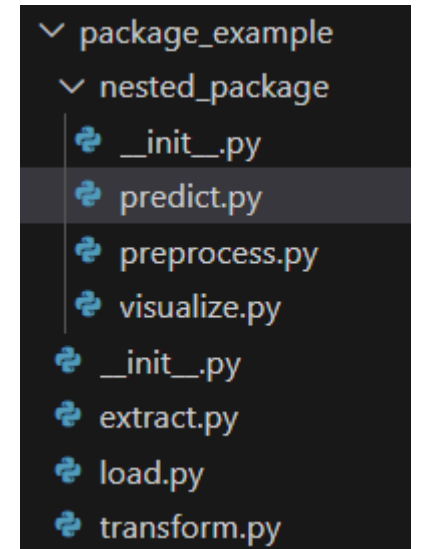
# Python. Задание 2.

1. Создайте в корневом каталоге проекта файл `new_module.py`
2. Добавьте в файл несколько переменных, функций и цикл `for`, например, с вызовом `print`
3. Импортируйте модуль `new_module` в `test.py`
4. Вызовите функцию из `new_module` и выведите в консоль значение одной из объявленных там переменных
5. Переместите файл в каталог `new_dir` в директории проекта (каталог нужно предварительно создать), перезапустите интерпретатор Python и снова выполните вызов функции и вывод переменной
6. После того, как вы убедитесь в работе импорта, можно вернуть `new_module` в корневой каталог и удалить каталог `new_dir`
7. Ограничьте в модуле `new_module` выполнение цикла `for` (из п.2), чтобы он не выполнялся при импорте
8. Перезапустите интерпретатор и убедитесь в корректной работе импорта: импорт функции, вывод переменной в консоль и отсутствие выполнения цикла

# Python. Пакет (package)

Пакет представляет собой структурированный каталог с набором модулей, которые решают различные задачи в рамках работы пакета.

- Пакеты импортируются точно так же, как и модули, через **import** <название пакета>
- В каталоге пакета должен находиться файл-инициализатор **\_\_init\_\_.py**, в котором задаются настройки импорта, в т.ч. модули, которые должны импортироваться при импорте пакета.
- Для импорта модулей пакета (или отдельных функций модуля) при импорте самого пакета следует импортировать необходимые модули (отдельные функции) в инициализаторе **\_\_init\_\_.py**. Для импорта лучше использовать не абсолютный (**import** <имя пакета>.<имя модуля>), а относительный импорт (**import** .<имя модуля>).
- Для импорта нескольких модулей пакета в инициализаторе можно использовать конструкцию **from . import** <модуль1>, <модуль2> и т.д.
- Для ограничения импорта объектов модуля внутри файла модуля указывается специальная переменная **\_\_all\_\_** - список, в котором перечисляются разрешенные к импорту модули.
- Если для модулей указаны переменные **\_\_all\_\_**, то в инициализаторе пакета удобно для импорта пользоваться конструкцией **from .<имя модуля> import \***
- Можно использовать вложенные пакеты, которые создаются по такому же принципу, как основной пакет (каталог с **\_\_init\_\_.py** и набором модулей), и импортируются в основной пакет.



```
package_example > __init__.py > ...
1  from . import extract, transform, load
2  PACKAGE_NAME = 'Example package'
3  print('loading package', PACKAGE_NAME)
```

```
package_example > load.py > load_data2
1  __all__ = ['load_data']
2
3  def load_data():
4      print("load data to source")
5
6  def load_data2():
7      print("Служебная функция")
```

# Python. Задание 3.

1. Создайте в корневом каталоге проекта пакет `package_example`. Если в IDE нет функции создания пакета, вручную создайте каталог с именем пакета и файл `__init__.py` внутри него.
2. Внутри `__init__.py` вызовите функцию `print` для проверки работы импорта пакета и импортируйте этот пакет в `test.py`
3. Добавьте в один из модулей еще несколько функций и определите переменную `__all__` в которой укажите функции, которые должны импортироваться из модуля, и импортируйте все функции этого модуля в `__init__.py` при помощи `*`.
4. Проверьте импорт функций из пакета `package_example` в `test.py`
5. Добавьте вложенный пакет в пакет `package_example` и создайте там несколько модулей.
6. Импортируйте эти модули в основной пакет и проверьте их работу аналогично проверке работы основного модуля.
7. Проверьте импорт модуля из основного пакета во вложенном пакете (`from ..<модуль основного пакета>`  
`import <имя функции>`)



# Python. Виртуальное окружение

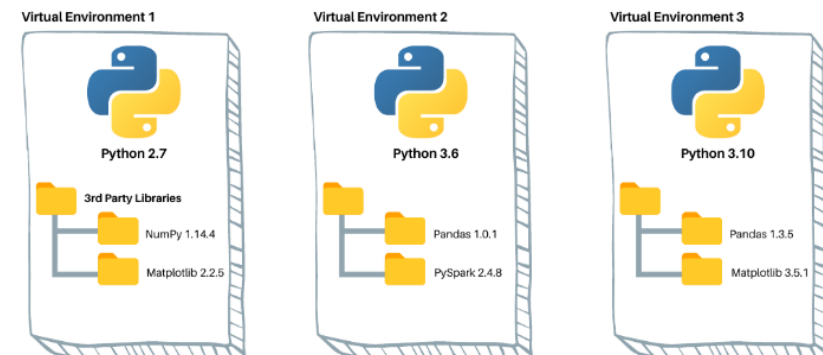
Основное назначение виртуального окружения (Virtual Environment) – создание изолированной конфигурации программного обеспечения с определенным зафиксированным набором библиотек определенных версий.

Для создания виртуального окружения существуют специальные утилиты, например:

- **venv** – встроен в Python, управление через CLI, не быстрый, нельзя поставить версию Python выше установленной
- **Virtualenv** – схожая с venv функциональность, сторонняя утилита
- **conda** – удобный, функциональный, есть GUI, но есть проблемы с производительностью
- **poetry** – функциональный, не простой в настройке

По сути утилиты для создания виртуального окружения создают отдельные директории для каждого виртуального окружения, в которых хранятся все конфигурации и непосредственно интерпретатор Python.

Для создания аналогичной конфигурации на другой машине достаточно установить модули из requirements.txt, который предварительно был выгружен из текущей конфигурации.



# Python. Утилита venv

Процесс настройки виртуального окружения зависит от того, какую утилиту вы используете.

Мы рассмотрим настройки при помощи встроенной в Python утилиты **venv**.

Процесс работы с **venv** достаточно прост:

- **Создание** нового виртуального окружения: **python -m venv** *<имя виртуального окружения>*. При этом создается каталог с именем виртуального окружения.

- **Активация** виртуального окружения:

**source bin/activate** для Linux или **.\<имя виртуального окружения>\Scripts\activate** для Windows

- Установка необходимых пакетов
- Работа с интерпретатором
- Деактивация виртуального окружения: **deactivate**

# Python. Утилита venv

```
PS C:\!ML\ITcloud\01_PythonBasic\04_Модульное_программирование> python -m venv new_env2
PS C:\!ML\ITcloud\01_PythonBasic\04_Модульное_программирование> .\new_env2\Scripts\activate
(new_env2) PS C:\!ML\ITcloud\01_PythonBasic\04_Модульное_программирование> python
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
>>> import requests
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'requests'
(new_env2) PS C:\!ML\ITcloud\01_PythonBasic\04_Модульное_программирование> pip install requests
Collecting requests
  Using cached requests-2.31.0-py3-none-any.whl (62 kB)
Collecting urllib3<3,>=1.21.1
  Downloading urllib3-2.0.4-py3-none-any.whl (123 kB)
    |████████████████████| 123 kB 652 kB/s
Collecting certifi>=2017.4.17
  Downloading certifi-2023.7.22-py3-none-any.whl (158 kB)
    |████████████████████| 158 kB 2.2 MB/s
Collecting charset-normalizer<4,>=2
  Downloading charset-normalizer-3.2.0-cp310-cp310-win_amd64.whl (96 kB)
    |████████████████████| 96 kB 2.1 MB/s
Collecting idna<4,>=2.5
  Using cached idna-3.4-py3-none-any.whl (61 kB)
Installing collected packages: urllib3, idna, charset-normalizer, certifi, requests
Successfully installed certifi-2023.7.22 charset-normalizer-3.2.0 idna-3.4 requests-2.31.0 urllib3-2.0.4
WARNING: You are using pip version 21.2.3; however, version 23.2.1 is available.
You should consider upgrading via the 'C:\!ML\ITcloud\01_PythonBasic\04_Модульное_программирование\new_env2\Scripts\python.exe -m pip install --upgrade pip' command.
(new_env2) PS C:\!ML\ITcloud\01_PythonBasic\04_Модульное_программирование> python
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>> exit()
(new_env2) PS C:\!ML\ITcloud\01_PythonBasic\04_Модульное_программирование> deactivate
PS C:\!ML\ITcloud\01_PythonBasic\04_Модульное_программирование> █
```

# Python. Задание 4.

1. Создайте внутри проекта новое виртуальное окружение при помощи команды [venv](#).
2. Активируйте его и установите несколько пакетов, которые в нем еще не установлены.
3. Попробуйте импортировать установленные только что пакеты в интерпретаторе Python.
4. Выгрузите список установленных пакетов в файл [requirements.txt](#)
5. Деактивируйте виртуальное окружение
6. Создайте новое виртуальное окружение с другим именем
7. Активируйте его и установите все пакеты из ранее созданного файла [requirements.txt](#)
8. Проверьте работу функций (п.3)
9. Деактивируйте виртуальное окружение
10. Удалите созданные окружения (директории)

# Python. Задание 5.

Задание выполняется после выполнения заданий по ООП.

1. Выгрузите текущий список установленных пакетов в файл `requirements.txt`
2. Создайте новое виртуальное окружение и установите туда все пакеты из сохраненного файла.
3. Создайте новый файл `*.py` и импортируйте в него из файла, созданного в рамках заданий по ООП, классы `Person`, `Employee`, `Engineer`, `Manager`.
4. Создайте несколько объектов классов `Engineer` и `Manager` и убедитесь в их корректной работе.

# Python. Тестирование

Назначение:

- **Проверка работоспособности кода**
- **Проверка исполнения контрактов кода** (после добавления новых функций старые могут перестать работать)
- **Помогают понять, что делает код**

Тесты бывают:

- **Юнит-тесты** – тестирование отдельных компонентов системы
- **Интеграционные** – тестирование связей компонентов системы или системы в целом

И еще множество различных разновидностей:

- **Нагрузочное** тестирование
- **Регрессионное** тестирование (на старую функциональность)
- **Приемочное** тестирование
- **Стресс-тестирование**
- **Модульное и функциональное** тестирование

Пакеты для автоматизации тестирования:

- **Pytest** – удобный, с возможностью параметризации, хорошо задокументированный,
- **Unittest** – встроен в Python, не очень удобный
- **Doctest** – тестирование документации
- ....

# Python. Тестирование. Pytest

## Преимущества pytest:

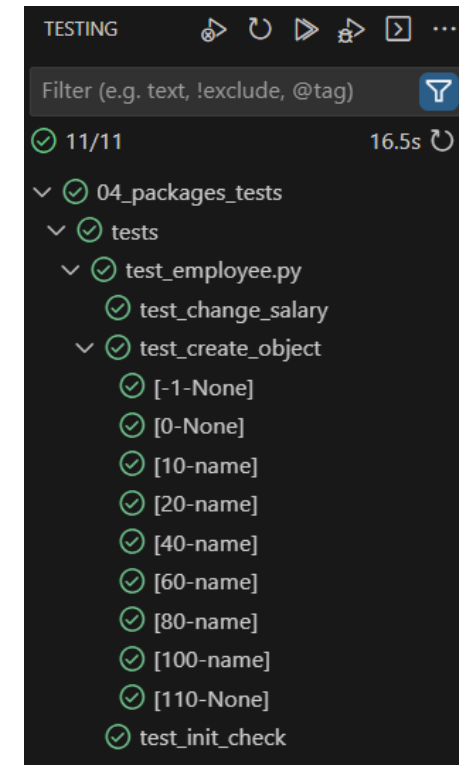
- Фильтрация тестов
- Параметризация тестов
- Совместимость с unittest
- Параллельное тестирование



## Основные моменты:

- Установка пакета через `$ pip install pytest`
- Для выполнения тестов требуется создать файл с описанием тестов в виде функций
- По умолчанию тестовые файлы и тестовые функции внутри них должны начинаться или заканчиваться словом «test»
- Выполнение тестов в файле: `$ pytest <имя файла>.py`
- Выполнение конкретного теста в файле: `$ pytest <имя файла>.py::<имя теста>`
- Проверка соответствия ожидаемого результата полученному через ключевое слово `assert`
- В pytest есть метки (`@pytest.mark.`), которые отвечают за настройки выполнения конкретного теста и добавляются как декоратор к нужному тесту: можно пропустить тест (`@pytest.mark.skip`), определить тест заранее как падающий (`@pytest.mark.xfail`), параметризовать тест (`@pytest.mark.parametrize`) и т.п.
- Также возможно использование для параметризации фикстур
- Больше информации здесь: <https://pytest-docs.ru.readthedocs.io/ru/latest/example/>

```
@pytest.mark.parametrize("age, result", [
    (-1, None), (0, None),
    (10, 'name'), (20, 'name'), (40, 'name'),
    (60, 'name'), (80, 'name'), (100, 'name'),
    (110, None)])
def test_create_object(age, result):
    assert str(mm.Manager('name', 'surname', 'manager', age)) == 'name surname' if result else 'None'
    assert str(mm.Engineer('name', 'surname', 'manager', age)) == 'name surname' if result else 'None'
```



# Python. Задание 6.

Напишите и выполните тесты для классов `Engineer` и `Manager`, в тестах должно присутствовать:

1. Создание объектов этих классов с различными значениями возраста (`@pytest.mark.parametrize`)
2. Использование для пропуска теста декоратора `@pytest.mark.skip`
3. Использование для тестов, которые не должны проходить декоратора `@pytest.mark.xfail` с указанием типа исключения

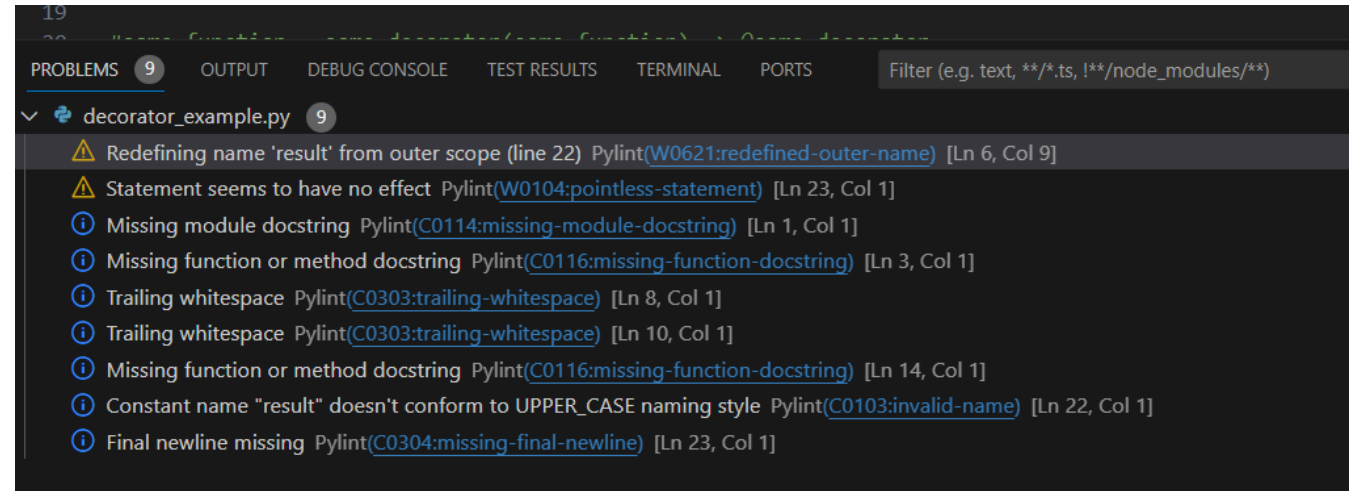


# Python. Линтеры

**Линтер** (от lint – программа для статического анализа кода на C, 1979) – утилита для статического анализа кода.

Популярные линтеры для Python:

- **flake8**
- **pylint**
- **PyFlakes**
- **Ruff**
- **mypy**
- ...



Линтеры чаще всего только указывают на ошибки, но есть некоторые линтеры, которые позволяют автоматически исправлять ошибки.

В интегрированной среде также возможно автоматически форматировать код в соответствии с указанными правилами (например, PEP8).

Для автоматического форматирования кода в IDE VSCode и PeCharm можно воспользоваться комбинацией [Shift-Alt-F](#).

Для установки линтера чаще всего следует его установить через `pip install` (например, [pip install pylint](#)).

Для корректной работы линтеров при объявлении метода следует указывать т.н. аннотации: типы аргументов и возвращаемый тип данных. В случае с несколькими типами они перечисляются через «|», также есть встроенная библиотека [typing](#), которая позволяет делать более детальные аннотации. Аннотации не влияют на работу интерпретатора Python, но помогают встроенным инструментам статического анализа кода делать верные выводы.

```
def __init__(self, name: str, surname: str, age: int) -> None:
```

# GIT



**GIT** – распределённая система управления версиями программного кода (OpenSource).

**github.com** – самый крупный веб-сервис для хранения и совместной разработки проектов (~340 млн репозиторий, 2022г).

**GitLab** – веб-инструмент для обслуживания жизненного цикла разработки и поддержки приложений, в котором реализована технология GIT и другие необходимые механизмы (например, CI/CD).

Кроме возможности совместной работы GIT обеспечивает просмотр изменений в коде и возможность их откатки.

Основные понятия:

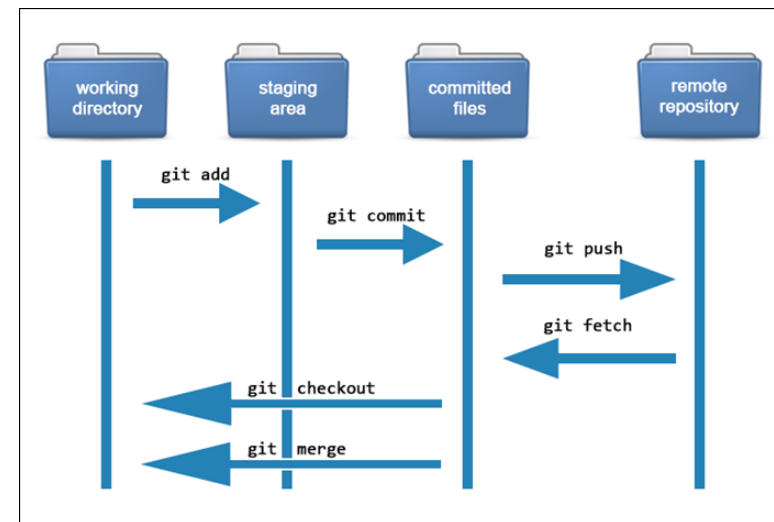
- **Репозиторий (repository)** – совокупность файлов, для которых отслеживается состояние и история их изменений.
- **Коммит (commit)** – сохраненное состояние (версия) репозитория.
- **Ветка (branch)** – последовательность коммитов, в репозитории может быть несколько.

Краткое описание работы:

Вы работаете над проектом локально в рабочей директории (**working directory**), для чего подгружаете последнее состояние репозитория командой **git pull** или полностью копируете репозиторий командой **git clone**. После внесения изменений в файлы вы добавляете измененные файлы в область подготовленных изменений (**staging area**). **Staging area** позволяет формировать **commit** из подготовленных файлов.

Собственно, после того, как у вас есть нужный набор подготовленных файлов вы их коммитите (**git commit**) и отправляете в репозиторий (**git push**).

<https://proglib.io/p/system-git>



<https://phoenixnap.com/kb/how-git-works>