Модуль 7. Создание модулей. Проверка идиоматичности кода

[Импорт собственных модулей 1](#_Toc147966359)

[Аргументы командной строки. Работа с модулем sys. 7](#_Toc147966360)

[Работа с I/O 13](#_Toc147966361)

[Работа с прикладными модулями 13](#_Toc147966362)

[Модуль random стандартной библиотеки 13](#_Toc147966363)

[Модуль decimal 13](#_Toc147966364)

[pip – пакетный менеджер Python 15](#_Toc147966365)

[Установка модулей в Python 17](#_Toc147966366)

[pylint и mypy – линтер и проверка типов 23](#_Toc147966367)

[Type Hinting и mypy 28](#_Toc147966368)

[Установка и настройка mypy 30](#_Toc147966369)

# Импорт собственных модулей

На этом занятии вы узнаете как создавать и импортировать свои собственные модули. Как мы говорили на предыдущем уроке, модули представляют собой обычный текстовый файл программы на Python. Поэтому, в текущем рабочем каталоге я создам еще один файл и назову его mymodule.py. (Щелкаем правой кнопкой по вкладке mymodule и выбираем пункт «Split and Move Right»). Запишем в этом модуле следующие строчки:

NAME = 'mymodule'

def average(\*args, accuracy=3):  
 s = sum(args) / len(args)   
 return round(s, accuracy)

В результате, в глобальном пространстве имен имеем две переменные: NAME и average. Все локальные переменные, например, x сюда уже не попадают – только глобальные определения.

Импортируем этот модуль в нашу программу ex.py:

**import** mymodule

и если теперь ниже записать:

mymodule.

А так

print(help(my\_module))

Какой результат

Улучшим читаемость сигнатуры с помощью аннотации типов

def average(\*args: tuple, accuracy=3) -> float:  
 *'''  
 Моя авторская функция  
 :param args: коллекция чисел  
 :param accuracy: точность результата  
 :return: среднее арифметическое  
 '''* s = sum(args) / len(args)  
 return round(s, accuracy)

и снова

print(average.\_\_doc\_\_)

то, как раз, увидим наши определения – переменную NAME и имя функции floor. То есть, они были импортированы в нашу программу и доступны через пространство имен mymodule:

**print**(mymodule.averagr(,4,4))

Какой тип объекта my\_module?

print(type(my\_module))

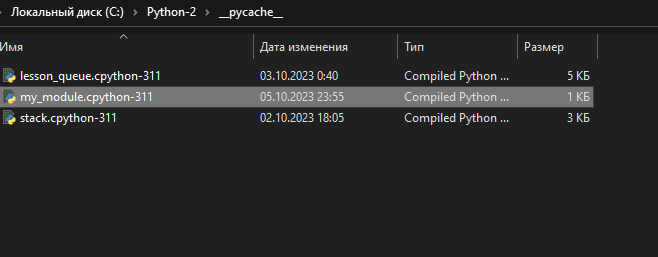
Результат:

<class 'module'>

А где питон ищет модули?

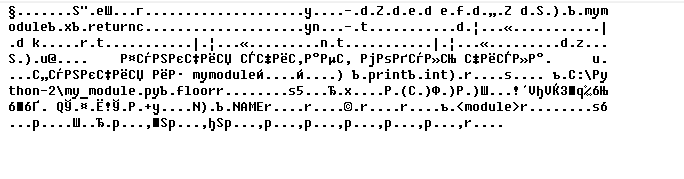
print(sys.path)

Как только в пайтоне появился модуль, он сразу предкомпилировался в байт-код.

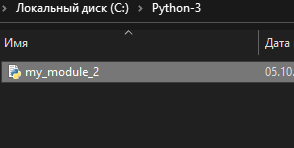


Скорость работы при обращении к нему будет как к скомпилированному exe-файлу!

Теперь на собеседовании, когда вас спросят, какой Пайтон язык, правильный ответ интерпретируемо-компилируемый!



А теперь перенесем наш модуль в папку Python-3 в корне диска С



Решаем задачу путем добавления пути 'c:\\Python-3'

sys.path.append('c:\\Python-3')  
import my\_module\_2  
  
print(help(my\_module\_2))

Или же, можем воспользоваться другой конструкцией:

**from** mymodule **import** average

и напрямую вызвать эту функцию:

**print**(average(-5,4))

Но что будет, если в файле mymodule тоже прописать импорт, скажем, библиотеки math, следующим образом:

**import** math

В этом случае, в модуле mymodule появляется глобальная переменная с именем math. Значит, при импорте уже модуля mymodule (в ex.py):

**import** mymodule

**import** pprint

мы должны увидеть это имя:

pprint.pprint(dir(mymodule))

Действительно, в конце списка оно присутствует, а это значит, к модулю math можно обратиться через модуль mymodule:

a = mymodule.math.floor(-5.6)

**print**(a)

То есть, мы здесь уже имеем иерархию модулей, сначала обращаемся к mymodule, затем к math, а потом к одной из его функций.

Конечно, чтобы не создавать таких цепочек, можно было бы импортировать библиотеку math через конструкцию:

**from** math **import** \*

(Еще раз отмечу, что так делать не рекомендуется. Здесь, я лишь в учебных целях показываю, что произойдет). В итоге получаем неявное расширение модуля mymodule за счет модуля math. И все функции из math мы теперь напрямую можем вызывать из mymodule:

a = mymodule.floor(-5.6)

b = mymodule.ceil(-5.6)

**print**(a, b)

Причем, вот эта функция floor была переопределена, поэтому вызывается не библиотечный вариант, а наш. Так вот, такой импорт ни в коем случае делать не стоит, чтобы не смешивать свои и стандартные функции в одном месте. Лучше или перечислять, те элементы, что мы собираемся использовать, например:

**from** math **import** pi, ceil

или же воспользоваться простым импортом:

**import** math

Давайте теперь детальнее разберемся, как работает импорт модулей. Первый вопрос, откуда Python «знает», где искать импортируемые модули? Порядок поиска прописан в специальной коллекции path модуля sys:

**import** sys

pprint.pprint(sys.path)

Здесь первые строчки – это каталог с исполняемым модулем ex.py и рабочий каталог, а далее, идут дополнительные пути поиска, например, для файлов стандартной библиотеки.

Модуль [pprint](https://digitology.tech/docs/python_3/library/pprint.html" \l "module-pprint" \o "pprint: Приятная печать данных.) предоставляет возможность «красиво распечатать» произвольные структуры данных Python в форме, которая может использоваться в качестве входных данных для интерпретатора. Если форматированные структуры включают объекты, которые не являются фундаментальными типами Python, представление может быть не загружаемым. Это может иметь место, если включены такие объекты, как файлы, сокеты или классы, а также многие другие объекты, которые не могут быть представлены как литералы Python.

Если мы переместим файл модуля mymodule из рабочего каталога, в какую-нибудь вложенную папку, например, с именем folder (создаем и перемещаем), то при импорте получим ошибку ModuleNotFoundError, так как к этой папке не прописан путь в коллекции path.

Поэтому, нам дополнительно нужно указать каталог folder, в котором находится модуль mymodule, следующим образом:

**import** folder.mymodule

Конечно, в список path мы можем добавить путь для поиска модулей:

sys.path.append(r"C:**\P**ythonProjects**\e**xample**\f**older")

и тогда при импорте по-прежнему достаточно будет прописывать только mymodule. Но это делается крайне редко, обычно добавляют имена подкаталогов через точку.

Давайте теперь поменяем местами эти два файла (ex.py и mymodule.py). В этом случае для импорта достаточно будет также прописать:

**import** mymodule

так как пути поиска теперь будут включать и рабочий каталог, где находится mymodule и каталог с исполняемым файлом.

Вернем файл ex.py в рабочий каталог. И заметим, что в момент выполнения импорта модуль преобразуется интерпретатором языка Python сначала в байт-код, а затем, запускается на исполнение. Это важный момент. Смотрите, если в mymodule прописать строчку:

**print**(NAME)

и запустить файл ex.py на исполнение, то в консоли увидим результат срабатывания функции print(). Причем она сработала в момент импорта модуля до вызова функции pprint(). Это означает, что если в mymodule будет записана программа:

**for** i **in** range(5):

**print**(NAME)

то она будет выполнена. Так что с этим следует быть аккуратным.

Значит, получается, что подключаемые модули должны исключительно содержать определения переменных, функций и других объектов языка Python, но не их вызовы? Не обязательно! В каждом модуле есть специальная переменная \_\_name\_\_, которая принимает имя модуля, если они запускается при импорте:

**print**(\_\_name\_\_)

и значение "\_\_main\_\_", если он запускается как самостоятельная программа (показываем).

И это свойство часто используют для контроля исполнения модуля. Если в нем нужно прописать программу, выполняемую только при непосредственном запуске модуля, то ее следует поместить в следующее условие:

**if** \_\_name\_\_ == "\_\_main\_\_": # поменять main

**print**("самостоятельный запуск")

**else**:

**print**("запуски при импорте")

Такую проверку часто можно увидеть в питоновских программах и вы теперь знаете, для чего она нужна.

Т.е. если эта программа запущена как модуль, этот блок кода (**if** \_\_name\_\_ ) не выполняется.

Надеюсь, из этого занятия вам стало понятно, как создавать свои собственные модули, как их импортировать и как все это работает в деталях. Жду всех вас на следующем занятии!

Ключевые моменты

* Выражения import производят поиск по списку путей в sys.path.
* sys.path всегда включает в себя путь скрипта, запущенного из командной строки, и не зависит от текущей рабочей директории.
* Импортирование пакета по сути равноценно импортированию  \_\_init\_\_.py этого пакета.

Основные определения

* Модуль: любой файл \*.py. Имя модуля — имя этого файла.
* Встроенный модуль: «модуль», который был написан на Си, скомпилирован и встроен в интерпретатор Python, и потому не имеет файла \*.py.
* Пакет: любая папка, которая содержит файл \_\_init\_\_.py. Имя пакета — имя папки.
  + С версии Python 3.3 любая папка (даже без \_\_init\_\_.py) считается пакетом.
* Объект: в Python почти всё является объектом — функции, классы, переменные и т. д.

Пример структуры директорий

test/ # Корневая папка

packA/ # Пакет packA

subA/ # Подпакет subA

\_\_init\_\_.py

sa1.py

sa2.py

\_\_init\_\_.py

a1.py

a2.py

packB/ # Пакет packB (неявный пакет пространства имён)

b1.py

b2.py

math.py

random.py

other.py

start.py

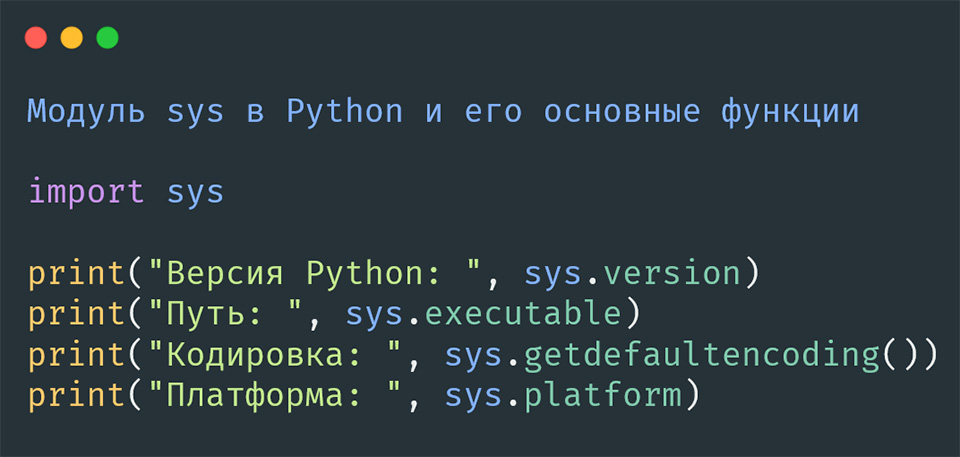
Обратите внимание, что в корневой папке test/ нет файла \_\_init\_\_.py.

Что делает import

При импорте модуля Python выполняет весь код в нём. При импорте пакета Python выполняет код в файле пакета \_\_init\_\_.py, если такой имеется. Все объекты, определённые в модуле или \_\_init\_\_.py, становятся доступны импортирующему.

# Аргументы командной строки. Работа с модулем sys.

**Sys** — это встроенный **модуль** Python, который предоставляет доступ к системным функциям и переменным. Этот **модуль** может быть использован для получения информации о версии Python, управления аргументами командной строки, работы с путями к файлам, принудительного завершения программы и многого другого.



print(sys.version)  
print(sys.executable)  
print(sys.getdefaultencoding())  
print(sys.platform)

Модуль sys предоставляет системе особые параметры и функции. В данном разделе мы рассмотрим следующее:

* sys.argv
* sys.executable
* sys.exit
* sys.modules
* sys.path
* sys.platform
* sys.stdin/stdout/stderr

**sys.argv**

Значение **sys.argv** – это список аргументов командной строки, которые причастны к скрипту Python. Первый аргумент, **argv[0]**, имеет аналогичное скрипту Python наименование.

Как вы, наверное, знаете, все программы можно условно разделить на консольные и использующие графический интерфейс (сервисы под Windows и демоны под Linux не будем брать в расчет).

Параметры запуска, задаваемые через командную строку, чаще всего используют консольные программы, хотя программы с графическим интерфейсом тоже не брезгуют этой возможностью.

Наверняка в жизни каждого программиста была ситуация, когда приходилось разбирать параметры командной строки, как правило, это не самая интересная часть программы, но без нее не обойтись.

В зависимости от платформы, на которой вы работаете, первый аргумент может содержать полный путь к скрипту или к названию файла. Для дополнительных деталей обратитесь к документации. А тем временем, попробуем поработать с парочкой примеров, чтобы познакомиться с этим инструментом:

Python

|  |
| --- |
| import sys  print(sys.argv)  # Ответ: [''] |

Если вы запустите это в интерпретаторе, вы получите **список с пустой строкой**. Давайте создадим файл под названием **sysargv.py**, со следующим содержимым:

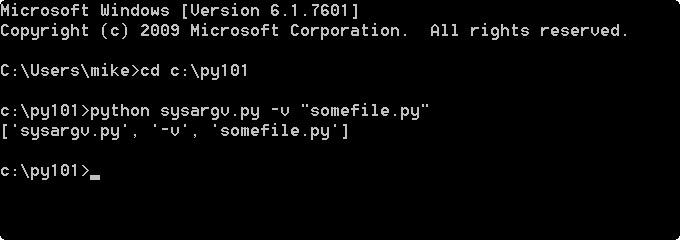
Python

|  |
| --- |
| import sys  print(sys.argv) |

Теперь **запустите код в IDLE**. Вы увидите список с одним элементом, который содержит путь к вашему скрипту.

Попробуем передать скрипту несколько аргументов.

Откройте окно терминала \ консоли и при помощи команды **cd** измените каталоги на тот, в котором находится скрипт. После этого, запустите что-то наподобие этого:



sys.argv

Обратите внимание на то, что будет выведено на экран:

Python

|  |  |
| --- | --- |
| 1 | ['sysargv.py', '-v', 'somefile.py'] |

print(**"Привет"**, sys.argv[1])

Первый аргумент – это название файла нашего скрипта. Следующие два аргумента в списке – это те, что мы передали нашему скрипту в командной строке.

l1 = len(sys.argv)  
print(**"Количество параметров:"**, l1 - 1)  
print(**"Путь запуска программы"**, **"\""** + sys.argv[0] + **"\""**)  
**if** l1 > 1:  
 print(**"Параметры командной строки:"**)  
 **for** i **in** range(1, l1):  
 print(sys.argv[i])

Существует Python-библиотека argparse, которая упрощает жизнь по сравнению с ручным разбором параметров. Позволяет создавать именованные и позиционные параметры, различные варианты их использования, в том числе в виде флагов, а также научились создавать команды и оформлять справку.

**sys.executable**

Значение **sys.executable** – это полный путь к **интерпретатору Python**. Это очень полезно, когда вы используете чей-то компьютер, и вам нужно узнать, где установлен Python. В некоторых системах, данная команда не сработает, и **выдаст пустую строку** с надписью None. Посмотрим, как пользоваться этой функцией:

Python

|  |
| --- |
| import sys    print(sys.executable)  # Вывод: 'C:\\Python27\\pythonw.exe' |

**sys.exit**

Данная функция позволяет разработчику **выйти из Python**. Функция **exit** принимает необязательный аргумент, обычно целое число, которое дает **статус выхода**.

Ноль считается как успешное завершение. Обязательно проверьте, имеет ли ваша операционная система какие-либо особые значения для своих статусов выхода, чтобы вы могли следить за ними в своем собственном приложении.

Обратите внимание на то, что когда вы вызываете **exit**, это вызовет [исключение SystemExit](https://python-scripts.com/try-except-finally), которое позволяет функциям очистки работать в конечных пунктах блоков [try / except](https://python-scripts.com/try-except-finally). Давайте взглянем на то, как вызывается данная функция:

Python

|  |
| --- |
| import sys  sys.exit(0)    Traceback (most recent call last):     File "<pyshell#5>", line 1, in <module>      sys.exit(0)  SystemExit: 0  print(**'start'**) time.sleep(3) print(**'end'**) exit(3) # каким кодом завершится прога |

Запустив данный код в IDLE, вы увидите возникшую **ошибку SystemExit**. Давайте создадим несколько скриптов для теста. Для начала вам нужно создать основной скрипт, программу, которая будет вызывать другой скрипт Python. Давайте назовем его “**call\_exit.py**”. Скрипт должен содержать следующее:

|  |
| --- |
| **import** subprocess  code = subprocess.call([**"python.exe"**, **"main.py"**]) print(**'main.py вернул -'**, code)  exit(4) |

Получается, мы заодно научились **вызывать разные скрипты Python** изнутри самого Python!

sys.path

Значение функции **path** модуля sys – это список строк, которые указывают путь поиска для модулей. Как правило, данная функция указывает Python, в каких локациях смотреть, когда он пытается [импортировать модуль](https://python-scripts.com/import-modules-python). В соответствии с документацией Python, **sys.path** инициализируется из переменной окружения **PYTHONPATH**, плюс зависимое от [установки](https://python-scripts.com/install-python-windows) значение, указанное по умолчанию. Давайте взглянем на пример:

|  |
| --- |
| import sys  print(sys.path) |

Результат:

Python

|  |
| --- |
| ['',  'C:\\Python27\\Lib\\idlelib',  'C:\\Python27\\lib\\site-packages\\setuptools-0.9.5-py2.7.egg',  'C:\\Python27\\lib\\site-packages\\pip-1.3.1-py2.7.egg',  'C:\\Python27\\lib\\site-packages\\sphinx-1.2b3-py2.7.egg',  'C:\\Python27\\lib\\site-packages\\docutils-0.11-py2.7.egg',  'C:\\Python27\\lib\\site-packages\\pygments-1.6-py2.7.egg',  'C:\\Windows\\system32\\python27.zip',  'C:\\Python27\\DLLs',  'C:\\Python27\\lib',  'C:\\Python27\\lib\\plat-win',  'C:\\Python27\\lib\\lib-tk',  'C:\\Python27',  'C:\\Python27\\lib\\site-packages',  'C:\\Python27\\lib\\site-packages\\PIL',  'C:\\Python27\\lib\\site-packages\\wx-2.9.4-msw'] |

Данная функция может быть весьма полезной во время отладки причины, по которой **модуль не импортируется**. Вы также можете изменить путь. Так как данная функция является путем, мы можем добавлять или удалять путь из неё. Давайте взглянем на то, **как добавлять путь**:

Python

|  |
| --- |
| sys.path.append |

sys.path.append(**"/path/to/my/module"**)  
print(sys.path)

#Удаление пути я оставлю как задание для учеников  
sys.path.remove(**"/path/to/my/module"**)  
print(sys.path)

sys.platform

Значение **sys.platform** – идентификатор платформы. Вы можете использовать **sys.platform** чтобы добавлять модули к **sys.path**, [импортировать разные модули](https://python-scripts.com/import-modules-python), в зависимости от платформы, или запускать разные части кода.

Давайте взглянем:

Python

|  |
| --- |
| import sys  print(sys.platform) # ответ: 'win32' |

В данном примере мы видим, что **Python работает в Windows**. Вот пример того, как мы можем воспользоваться данной информацией:

**import** sys  
  
os = sys.platform  
**if** os == **"win32"**:  
 print (**"Работаем с реестром Windows"**)

code = subprocess.call([**"mspaint"**])  
 *# import \_winreg***elif** os.startswith(**'linux'**):  
 *# Выполняем Linux комманду* **import** subprocess  
 subprocess.Popen([**"ls -l"**])

# Работа с I/O

Модуль sys содержит стандартные потоки ввода, вывода и ошибок:

sys.stdin — стандартный поток ввода

sys.stdout — стандартный поток вывода

sys.stderr — стандартный поток ошибок

Пример использования:

import sys

sys.stdout.write("Привет, мир!\n")

sys.stderr.write("Это сообщение об ошибке\n")

Размер объекта в памяти

С помощью функции getsizeof() модуля sys можно получить размер объекта в памяти:

import sys

a = [1, 2, 3, 4]

print("Размер списка a в памяти:", sys.getsizeof(a), "байт")

Существует много других значений и методов в [модуле sys](https://python-scripts.com/module-sys).

Теперь вы знаете, как выйти из программы Python, как получить информацию о платформе, работать с аргументами, переданными командной строке, и многому другому.

# Работа с прикладными модулями

## Модуль decimal

number = 0.1 + 0.1 + 0.1

print(number) # 0.30000000000000004

Проблему может решить использование функции round(), которая округлит число. Однако есть и другой способ, который заключается в использовании встроенного модуля decimal.

Ключевым компонентом для работы с числами в этом модуле является класс Decimal. Для его применения нам надо создать его объект с помощью конструктора. В конструктор передается строковое значение, которое представляет число:

from decimal import Decimal

number = Decimal("0.1")

После этого объект Decimal можно использовать в арифметических операциях:

from decimal import Decimal

number = Decimal("0.1")

number = number + number + number

print(number) # 0.3

Округление чисел

Объекты Decimal имеют метод quantize(), который позволяет округлять числа. В этот метод в качестве первого аргумента передается также объект Decimal, который указывает формат округления числа:

number = Decimal("0.444")

number = number.quantize(Decimal("1.00"))

print(number) # 0.44

number = Decimal("0.555678")

print(number.quantize(Decimal("1.00"))) # 0.56

number = Decimal("0.999")

print(number.quantize(Decimal("1.00"))) # 1.00

Используемая строка "1.00" указывает, что округление будет идти до двух знаков в дробной части.

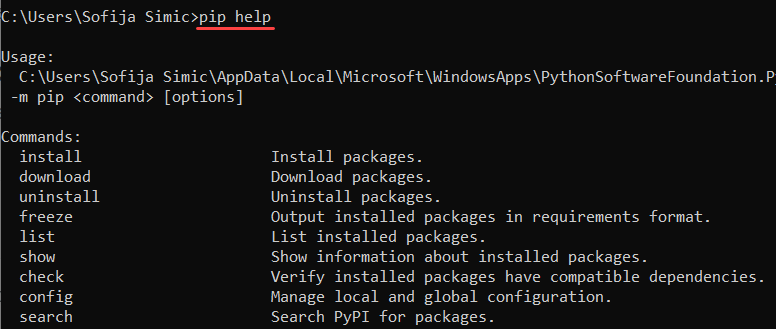
# pip – пакетный менеджер Python

Внешний модуль представляет собой такую же программу на python, которая уже написана до вас другим разработчиком и имеет свою функциональность. Чтобы не писать ту же самую программу можно просто установить сторонний модуль.

Ставятся сторонние модули через pip.

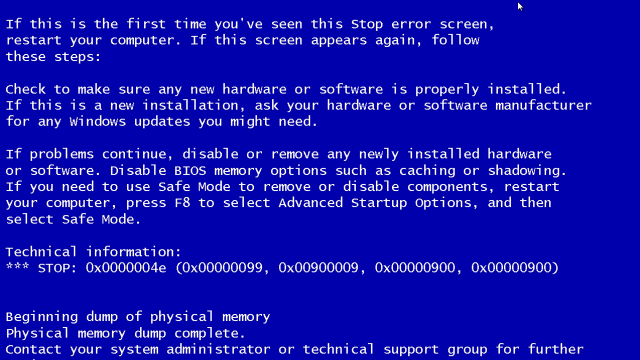
*pip - менеджер пакетов для Python. Это означает, что это инструмент, который позволяет вам устанавливать и управлять дополнительными библиотеками и зависимостями, которые не распространяются как часть стандартной библиотеки.*

В последних версиях python pip устанавливается автоматически вместе с интерпретатором python. Проверить его работоспособность можно в командной строке windows или в термина linux/mac. Наберите там pip или pip3:

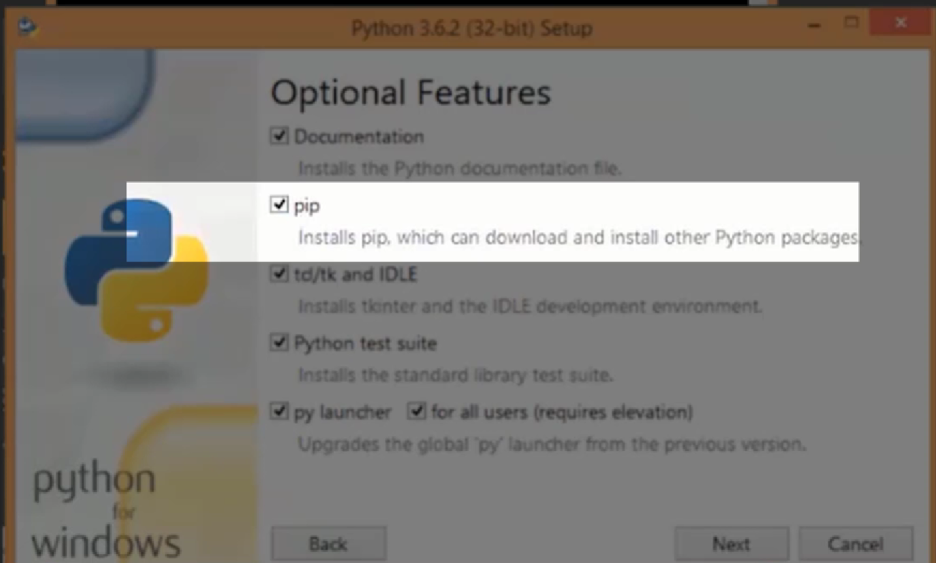


Если у вас возникла ошибка

pip: command not found

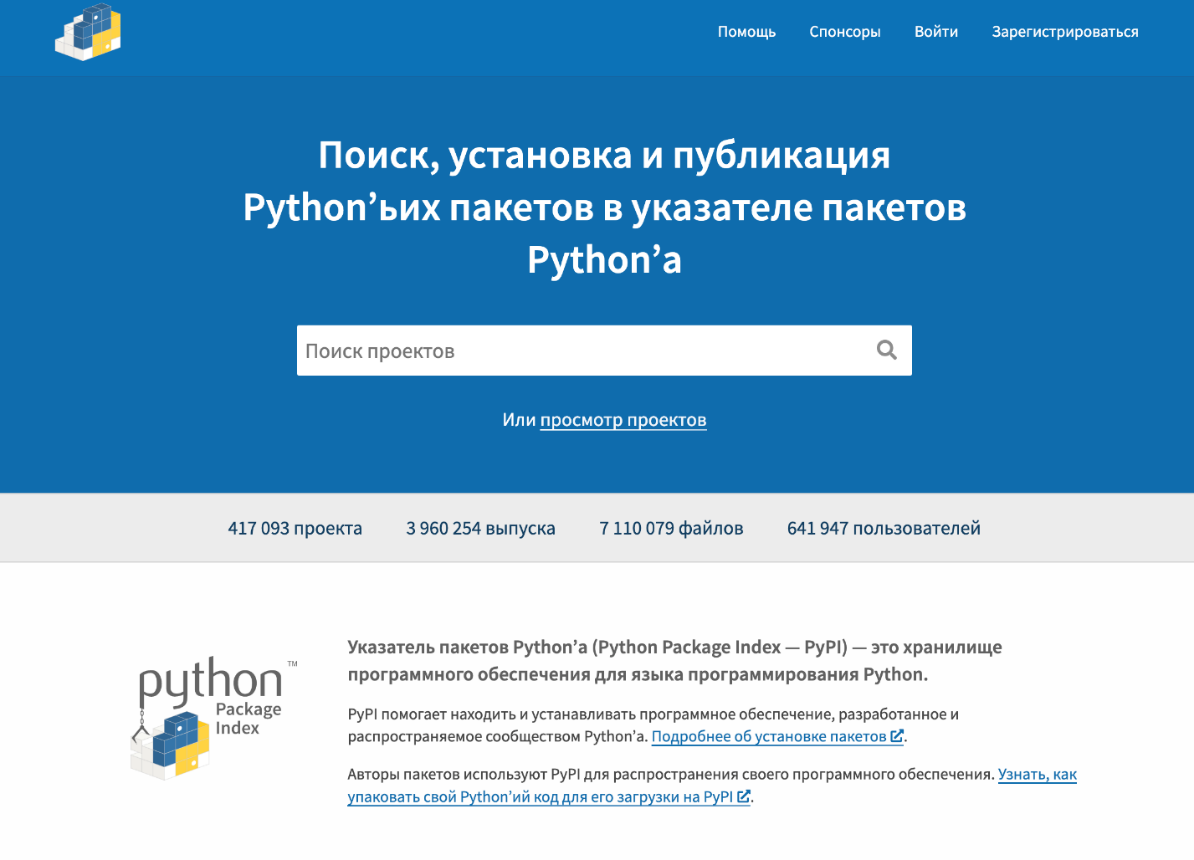


вы можете его установить pip [отдельно](https://phoenixnap.com/kb/install-pip-windows) , либо переустановить интерпретатор python, нажав при установке на эту галочку:



Откуда pip берет пакеты?

Существует специальный репозиторий(хранилище) пакетов [pypi.org](https://pypi.org/). Именно отсюда pip будет по умолчанию скачивать пакеты.

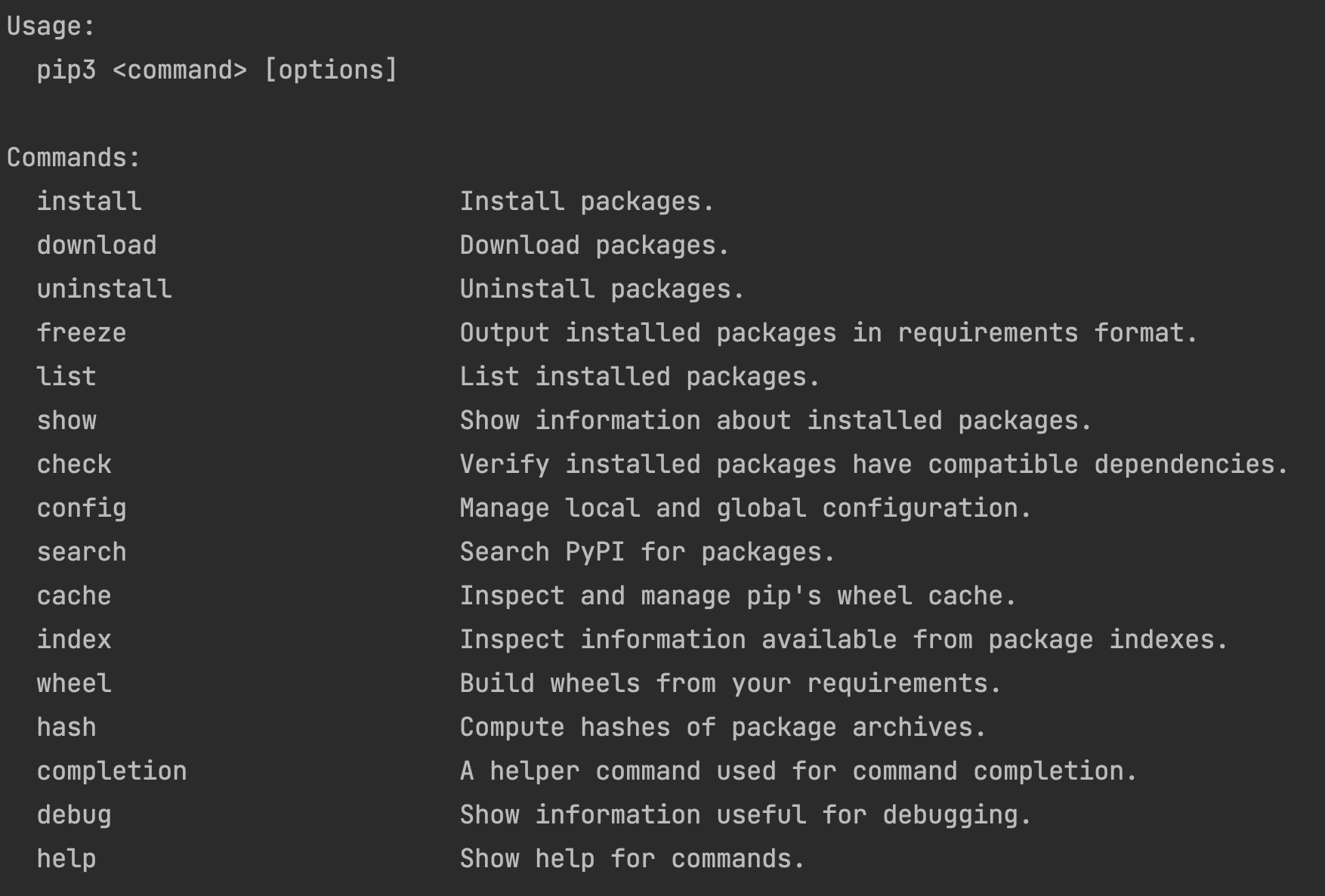


В репозитории существует поиск, в котором вы можете поискать по ключевым словам интересующие вас пакеты.

# Установка модулей в Python

Команды pip

Когда вы наберете в командной строке pip, то должна выйти подсказка по этой инструкции, в которой перечисляется следующий перечень команд:



Среди них нам наиболее интересны:

* команда install, позволяющая установить модуль;
* команда uninstall, позволяющая удалить установленный модуль;
* команда list, позволяющая вывести список установленных модулей в вашем интерпретаторе;
* команда freeze, позволяющая вывести установленные пакеты в формате файла requirements;

Способы установки сторонних модулей на python

Способ 1. При помощи командной строки.

Для этого:

* в windows открываем панель «Пуск», набираем cmd и открываем командную строку.
* в linux/max открываем «Терминал»

Для установки модуля необходимо пользоваться следующим шаблоном:

pip install <имя\_модуля>

Пример, для установки пакета flask необходимо набрать:

pip install flask

Если при установке не указывать версию пакета, то по умолчанию будет устанавливаться самая последняя версия.

При установке можно указывать версию по следующему принципу:

pip install <имя\_модуля>==версия

На примере как это делается:

pip install requests==2.10

При этом, если у вас был модуль, а вы установили его другую версию, то первая версия будет удалена, а лишь потом установится указанная версия. То есть необходимо запомнить, что

*Одновременно у одной версии интерпретатора python не может быть установлен один и тот же пакет в двух разных версиях. При установке пакета, если уже был установлен данный пакет другой версии, сперва удалится прежняя версия, и только потом установится новая*

Для проверки установленных модулей необходимо ввести:

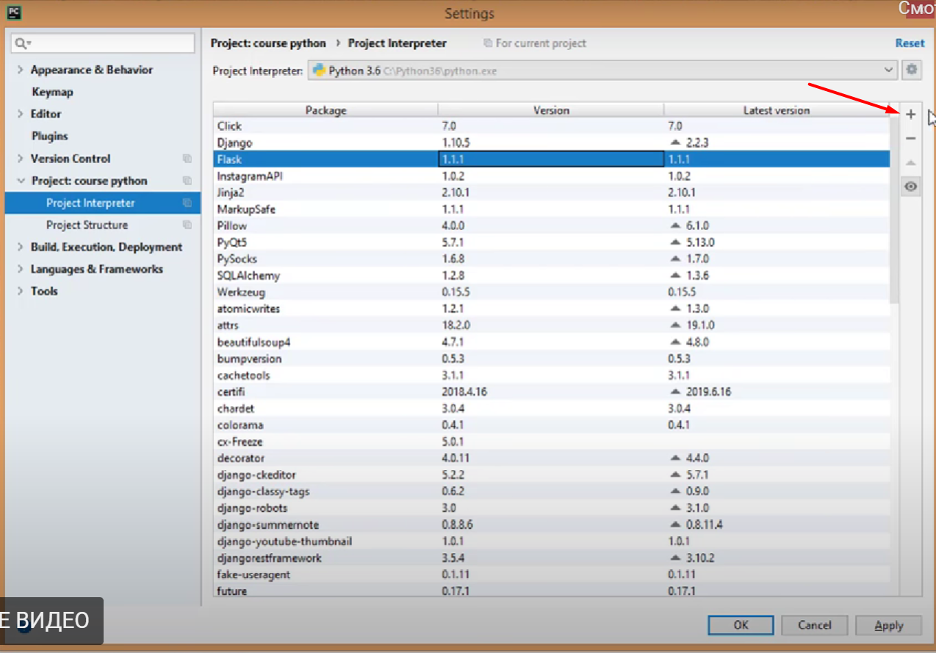
pip list

Для тех, кто до этого не работал с командной строкой, подобный способ установки пакетов выглядит сложным и затруднительным, поэтому можете воспользоваться альтернативным способом через pyСharm.

Способ 2. Через Pycharm.

В pyСharm открываете вкладку «File» –> «Settings» –> «Project interpreter»

Здесь высветится весь список установленных пакетов.



Для добавления нового пакета надо нажать на «плюсик», как на рисунке выше. После чего открывается список всех возможных модулей. Данный список достаточно большой, поэтому лучше воспользоваться поиском, после нахождения нужно нажать на модуль и нажать на кнопку Install Package. Процесс установки начинается, после его завершения указанный модуль появится в списке наших модулей. После чего нажимаем Apply, потом ok. Теперь можно пользоваться этим модулем.

Также в самом pycharm можно легко открыть командную строку, это делается нажатием на кнопку Terminal.

Файл requirements.txt

В одном проекте может быть задействовано десятки сторонних пакетов устанавливать по одному пакету было очень неудобно. Плюс версии самих пакетов постоянно меняются и ваша программа, которая идеально работала с пакетом django==2.0, будет выпадать с ошибкой при работе с django==3.0

Поэтому принято среди разработчиков python создавать спецификацию зависимостей и версий, которые используются для разработки и тестирования своего приложения. Такая спецификация создается в файле requirements.txt

requirements.txt позволяют вам точно указать, какие пакеты и версии должны быть установлены. Вот пример такого файла:

# requirements.txt

openpyxl==3.0.7

pandas==1.3.3

ujson==4.1.0

uvicorn==0.15.0

uvloop==0.16.0

fastapi==0.68.1

Здесь перечисляются пакеты с их версиями

Если указать без версии

# requirements.txt

openpyxl

pandas

ujson

uvicorn

uvloop

fastapi

то будут установлены самые последнии версии пакетов.

*Лучше всегда указывать используемую версию пакета в файле requirements.txt*

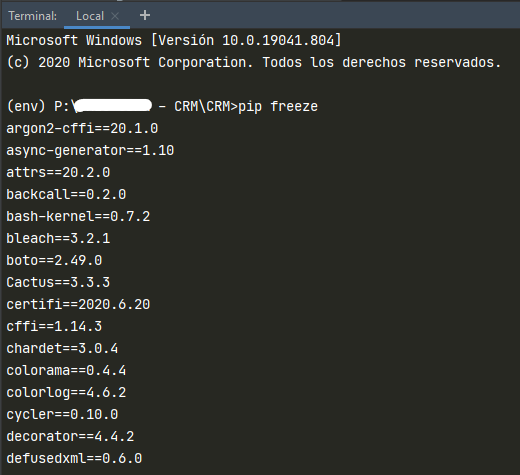
Создание файла requirements.txt

Вы можете создать файл requirements.txt вручную, это обычный текстовый документ. И затем заполнить его названиями необходимых пакетов и версий

Или вы можете создать файл requirements.txt на основании уже установленных пакетов на вашем интерпретаторе. Для это нужно вспомнить команду

pip freeze

она выведет все установленные пакеты в формате, подходящем для



Остается перенаправить это вывод в файл requirements.txt при помощи следующей команды

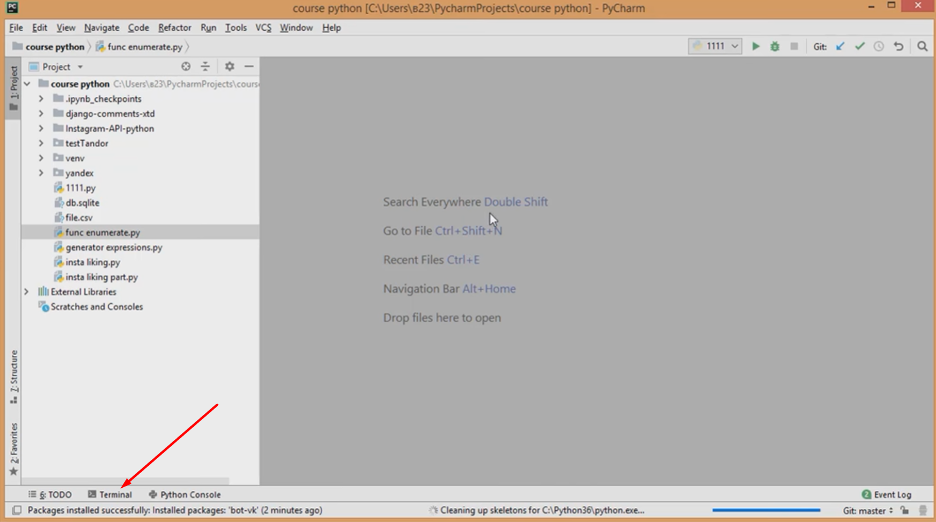
pip freeze > requirements.txt

Установка пакетов через файл requirements.txt

Файл requirements.txt позволяет установить нескольких пакетов сразу. Для это вам нужно в командной строке набрать следующую команду

pip install -r requirements.txt

При запуске этой команды ваш текущий путь в командной строке должен находиться в той же директории, где находиться файл requirements.txt



После её открытия можно так же воспользоваться пакетным менеджером pip.

Задание по pip и requirements

Задание 1. Установите пакет, который называется openpyxl

В комментарии делитесь результатами при помощи команды

pip list

Задание 2. Установите пакет, который называется requests версии 2.18.0

В комментарии делитесь результатами при помощи команды

Задание 3. Создайте файл requirements.txt при помощи команды

pip freeze > requirements.txt

В комментарии делитесь результатами при помощи команды

Задание 4. Ваша задача создайте файл requirements.txt со следующим содержимым

openpyxl==3.0.7

ujson==4.1.0

uvicorn==0.15.0

uvloop==0.16.0

jinja2==3.0.2

pdfkit-async==0.7.2

aiosmtplib==1.1.6

И произвести установку всех этих модулей через команду

pip install -r requirements.txt

В комментарии делитесь результатами при помощи команды

# pylint и mypy – линтер и проверка типов

В сообществе Python, как и в любой другой группе людей, существует некое коллективное знание. Множество людей прошлось по всем возможным граблям и получило опыт через набитые шишки. Затем через какое-то время, благодаря выступлениям на конференциях, официальным заявлениям, документам, статьям в блогах, код-ревью и личному общению, это знание стало коллективным. Теперь мы просто называем его “хорошими практиками”.

К таким хорошим практикам можно отнести, например, следующие.

Форматировать код по PEP8 — если этого не делать, то другим людям будет намного сложнее понимать ваш код; в плохо оформленном коде сложнее увидеть суть, потому что мозг постоянно отвлекается на не несущие смысловой нагрузки особенности оформления.

Не допускать объявленных, но неиспользуемых переменных/функций/импортов — опять же, это усложняет восприятие кода; читателю потребуется потратить время на то, чтобы осознать, что вот на эту сущность обращать внимания не нужно.

Писать короткие функции — слишком сложные функции с большим количеством ветвлений и циклов тяжело понимать.

Не использовать изменяемый объект в качестве значения аргумента функции по умолчанию — иначе в результате можно получить очень неожиданные эффекты.

Соблюдать (и даже просто помнить) все хорошие практики — не самая простая задача. Зачастую люди плохо справляются с тем, чтобы отсчитывать пробелы и контролировать переменные, и вообще склонны допускать ошибки по невнимательности. Таковы люди, ничего не поделаешь. Машины, наоборот, прекрасно справляются с такими хорошо определёнными задачами, поэтому появились инструменты, которые контролируют следование хорошим практикам.

В компилируемых языках ещё на этапе компиляции программист может получить по щщам первый полезный фидбэк о написанном коде. Компилятор проверит, что код валиден и может быть скомпилирован, а также может выдать предупреждения и рекомендации, как сделать код лучше или читаемее. Т.к. Python является интерпретируемым языком, где этап компиляции как таковой отсутствует, линтеры особенно полезны. На самом деле, это очень важно и круто — узнать, что твой код как минимум является валидным Python-кодом, даже не запуская его.

Существует множество линтеров для языка Python, каждый со своими особенностями и настройками. Вот некоторые из наиболее популярных линтеров Python:

1. **pylint** – это один из наиболее популярных линтеров Python, который проверяет синтаксис, стиль кода, потенциальные ошибки и многое другое. Pylint имеет множество опций настройки и гибкую конфигурацию.
2. **flake8** – это линтер, который объединяет несколько инструментов, включая pylint, pycodestyle и McCabe. Он проверяет синтаксис, стиль кода и комплексность кода. Flake8 очень прост в использовании и настраивается через конфигурационный файл.
3. **black** – это линтер, который автоматически форматирует код в соответствии с PEP 8. Он использует жесткие правила форматирования, которые не всегда соответствуют личным предпочтениям разработчика. Тем не менее, black является очень популярным инструментом для автоматического форматирования кода.
4. **mypy** – это линтер, который проверяет типы данных в вашем коде и обнаруживает потенциальные ошибки типизации. Mypy может быть очень полезен при разработке крупных проектов, где типизация может быть сложной.

pylint — это ещё один популярный линтер для Python. Этот линтер значительно умнее и продвинутее flake8. В pylint из коробки заложено очень много правил и рекомендаций, и по умолчанию они все включены, так что он достаточно строгий и придирчивый. Чтобы интегрировать его в существующий большой проект придётся потратить некоторое время, чтобы выбрать те правила, которые для вас важны. Так же как и flake8, pylint поддерживает плагины для расширения базовой функциональности, но насколько я вижу, экосистема плагинов у pylint значительно беднее.

Также при каждом запуске pylint выводит оценку качества кода по десятибалльной шкале, а также следит, как эта оценка меняется с течением времени. Достичь десятки очень сложно, но это благородная цель, к которой нужно стремиться.

Установка pylint принципиально ничем не отличается от установки flake8. Выполнить внутри виртуального окружения проекта:

$ pip install pylint

Для pipenv:

$ pipenv install --dev pylint

Для poetry:

$ poetry add --dev pylint

Использование - pylint можно натравить на определённый файл:

$ pylint file.py

С директориями у pylint дела обстоят чуть сложнее. Все директории он обрабатывает как питоновские модули, поэтому если в директории нет хотя бы пустого файла \_\_init\_\_.py, то работать с ней pylint не сможет. Имейте это ввиду.

Давайте попросим pylint прокомментировать файл с плохими практиками из предыдущего примера:

$ pylint bad\_code.py

\*\*\*\*\*\*\*\*\*\*\*\*\* Module bad\_code

bad\_code.py:4:25: C0326: No space allowed before bracket

def CalculateSquareRoot (Number ):

^ (bad-whitespace)

bad\_code.py:4:33: C0326: No space allowed before bracket

def CalculateSquareRoot (Number ):

^ (bad-whitespace)

bad\_code.py:5:0: W0312: Found indentation with tabs instead of spaces (mixed-indentation)

bad\_code.py:5:21: C0326: No space allowed before bracket

return sqrt(Number )

^ (bad-whitespace)

bad\_code.py:13:19: C0326: Exactly one space required around assignment

your\_number=float(input('Enter your number: '))

^ (bad-whitespace)

bad\_code.py:14:66: C0303: Trailing whitespace (trailing-whitespace)

bad\_code.py:15:13: C0303: Trailing whitespace (trailing-whitespace)

bad\_code.py:1:0: W0622: Redefining built-in 'pow' (redefined-builtin)

bad\_code.py:1:0: C0114: Missing module docstring (missing-module-docstring)

bad\_code.py:1:0: W0401: Wildcard import math (wildcard-import)

bad\_code.py:4:0: C0103: Function name "CalculateSquareRoot" doesn't conform to snake\_case naming style (invalid-name)

bad\_code.py:4:0: C0103: Argument name "Number" doesn't conform to snake\_case naming style (invalid-name)

bad\_code.py:4:0: C0116: Missing function or method docstring (missing-function-docstring)

bad\_code.py:7:0: W0102: Dangerous default value [] as argument (dangerous-default-value)

bad\_code.py:7:0: C0103: Argument name "l" doesn't conform to snake\_case naming style (invalid-name)

bad\_code.py:7:0: C0116: Missing function or method docstring (missing-function-docstring)

bad\_code.py:16:4: W0702: No exception type(s) specified (bare-except)

bad\_code.py:1:0: W0614: Unused import acos from wildcard import (unused-wildcard-import)

bad\_code.py:1:0: W0614: Unused import acosh from wildcard import (unused-wildcard-import)

bad\_code.py:1:0: W0614: Unused import asin from wildcard import (unused-wildcard-import)

bad\_code.py:1:0: W0614: Unused import asinh from wildcard import (unused-wildcard-import)

...

bad\_code.py:2:0: W0611: Unused import itertools (unused-import)

-------------------------------------

Your code has been rated at -41.43/10

Я немного сократил вывод. Как видите, даже без плагинов pylint нашёл все ожидаемые ошибки, и даже больше — например, он даже предлагает написать документацию.

По каждой ошибке можно запросить более подробную справку, используя название правила из конца строки с ошибкой или код:

$ pylint --help-msg=missing-docstring

$ pylint --help-msg=R0902

Вот какие ошибки pylint находит для файла, который с точки зрения flake8 не содержит никаких ошибок:

$ pylint not\_so\_bad\_code.py

\*\*\*\*\*\*\*\*\*\*\*\*\* Module not\_so\_bad\_code

not\_so\_bad\_code.py:1:0: C0114: Missing module docstring (missing-module-docstring)

not\_so\_bad\_code.py:4:0: C0116: Missing function or method docstring (missing-function-docstring)

not\_so\_bad\_code.py:8:0: C0103: Argument name "l" doesn't conform to snake\_case naming style (invalid-name)

not\_so\_bad\_code.py:8:0: C0116: Missing function or method docstring (missing-function-docstring)

not\_so\_bad\_code.py:20:11: W0703: Catching too general exception Exception (broad-except)

-----------------------------------

Your code has been rated at 6.67/10

А вот так в pylint можно игнорировать отдельную ошибку на строке прямо в файлах с кодом:

def append\_item(item, l=None): # pylint: disable=C0103

...

Ещё pylint умеет игнорировать ошибки в блоках кода:

def test():

# Disable all the no-member violations in this function

# pylint: disable=no-member

...

И для файлов целиком. Вот так можно отключить все ошибки из категорий Warning, Convention и Refactor:

# pylint: disable=W,C,R

А можно не проверять файл вообще:

# pylint: skip-file

Подробнее о правилах управления сообщениями [смотрите в документации](http://pylint.pycqa.org/en/latest/user_guide/message-control.html). Для более сложной настройки правил, придётся по-настоящему сконфигурировать pylint.

Конфигурация. pylint настраивается через файл .pylintrc в корне проекта. Чтобы создать дефолтный файл конфигурации, нужно выполнить следующую команду:

$ pylint --generate-rcfile > .pylintrc

Созданный файл содержит все поддерживаемые pylint опции с довольно подробными комментариями, так что углубляться я не буду.

# Type Hinting и mypy

**Type hinting**– это возможность указать типы аргументов функций, переменных и возвращаемых значений в Python коде. Это дополнительная информация для интерпретатора, которая позволяет проводить статический анализ кода и выявлять возможные ошибки на этапе компиляции или запуска программы.

Type hinting был введен в Python 3.5 в качестве экспериментальной возможности и стал стандартным в Python 3.6 и выше. С помощью type hinting можно указать типы аргументов функций и возвращаемых значений, а также типы переменных и атрибутов классов.

**Преимущества использования type hinting включают:**

* **Улучшение читаемости и понимания кода**: указание типов позволяет лучше понимать, какие данные используются в программе, и какие типы данных ожидаются на входе и выходе функций.
* **Улучшение надежности и безопасности**: статическая проверка типов может выявлять ошибки на этапе компиляции или запуска программы, что помогает предотвратить многие ошибки и баги в программном коде.
* **Улучшение автоматической документации**: type hinting позволяет автоматически генерировать документацию по коду, которая может быть использована другими разработчиками.

Type hinting в Python осуществляется с помощью аннотаций типов. Например, вот как можно указать тип аргументов и возвращаемого значения функции с использованием type hinting:

def add\_numbers(a: int, b: int) -> int:

return a + b

Здесь мы указываем, что функция **add\_numbers** принимает два аргумента типа **int** и возвращает значение типа **int**.

* Использование Union для указания нескольких возможных типов:

from typing import Union

def to\_str(value: Union[int, float]) -> str:

return str(value)

В этом примере мы указываем, что функция **to\_str** может принимать как целое число (**int**), так и число с плавающей точкой (**float**) и возвращает строку.

* Использование List и Tuple для указания типов элементов в списке или кортеже:

from typing import List, Tuple

def process\_data(data: List[Tuple[str, int]]) -> None:

for name, age in data:

print(f"{name} is {age} years old")

В этом примере мы указываем, что функция **process\_data** принимает список кортежей, где первый элемент - это строка (**str**), а второй - это целое число (**int**).

* Использование Callable для указания типа функции в качестве аргумента:

from typing import Callable

def apply(func: Callable[[int, int], int], x: int, y: int) -> int:

return func(x, y)

В этом примере мы указываем, что функция **apply** принимает другую функцию (**func**) в качестве аргумента, которая принимает два целых числа и возвращает целое число. Возвращаемое значение функции **apply** также является целым числом.

**Mypy** – это статический анализатор типов для языка Python. Он позволяет избежать многих ошибок, связанных с неправильными типами данных.

# **Установка и настройка mypy**

Для установки mypy можно использовать менеджер пакетов pip:

poetry add mypy

Для настройки mypy в файле **setup.cfg** нужно добавить секцию **[mypy]** со следующими параметрами:

* **python\_version** - версия Python, для которой настраивается mypy;
* **ignore\_missing\_imports** - если установлено в **True**, mypy не будет ругаться на отсутствие импорта модулей, которые используются только в типах;
* **strict** - уровень строгости проверки типов.

Пример файла **setup.cfg** с настройками mypy:

[mypy]

python\_version = 3.9

ignore\_missing\_imports = True

strict = True

После установки мы можем запустить mypy для конкретного файла с помощью команды:

mypy <имя\_файла>

Мы также можем запустить mypy для всех файлов в определенной директории с помощью команды:

mypy <имя\_директории>

В качестве примера, давайте создадим файл **example.py** со следующим содержимым:

def greet(name: str) -> str:

return "Hello, " + name

greet(1)

Здесь мы определяем функцию **greet**, которая принимает строку **name** и возвращает строку. Мы указали тип аргумента **name** как **str** и тип возвращаемого значения как **str**.

Однако, мы вызываем функцию **greet** с аргументом типа **int**. Это приведет к ошибке во время выполнения.

Давайте запустим mypy для файла **example.py** с помощью команды:

mypy example.py

Это выведет следующую ошибку:

example.py:4: error: Argument 1 to "greet" has incompatible type "int"; expected "str"

Mypy обнаружил ошибку типизации в нашем коде, указав на строку 4, где мы передаем аргумент неправильного типа.