**Оглавление**

[Определение Python 5](#_Toc211793820)

[Типы данных 6](#_Toc211793821)

[Мутабельность и иммутабельность 8](#_Toc211793822)

[Переменные 11](#_Toc211793823)

[Описание типов данных 12](#_Toc211793824)

[Числа 12](#_Toc211793825)

[Строки 14](#_Toc211793826)

[Перенос строки 19](#_Toc211793827)

[Списки 21](#_Toc211793828)

[Кортежи 25](#_Toc211793829)

[Множества 26](#_Toc211793830)

[Словари 30](#_Toc211793831)

[Объект 35](#_Toc211793832)

[Диапазоны 36](#_Toc211793833)

[Работа с файлами и вводом 38](#_Toc211793834)

[stdin поток ввода 38](#_Toc211793835)

[open 40](#_Toc211793836)

[close 42](#_Toc211793837)

[Чтение файла 43](#_Toc211793838)

[Менеджер контекста with 45](#_Toc211793839)

[Запись в файл 46](#_Toc211793840)

[JSON 47](#_Toc211793841)

[Операторы и функции 48](#_Toc211793842)

[Операторы сравнения 48](#_Toc211793843)

[Логические операторы 48](#_Toc211793844)

[Операторы if, elif и else 50](#_Toc211793845)

[Операции для объединения условий 50](#_Toc211793846)

[match-case 52](#_Toc211793847)

[Циклы 53](#_Toc211793848)

[for 53](#_Toc211793849)

[while 55](#_Toc211793850)

[Моржовый оператор := 56](#_Toc211793851)

[enumerate 57](#_Toc211793852)

[List comprehension Генераторы списков 58](#_Toc211793853)

[Generator Expressions Выражения-генераторы 60](#_Toc211793854)

[Dictionary comprehension Генератор словаря 61](#_Toc211793855)

[Set comprehension Генераторы множеств 62](#_Toc211793856)

[in 63](#_Toc211793857)

[Математические функции 64](#_Toc211793858)

[Ввод данных 66](#_Toc211793859)

[zip 67](#_Toc211793860)

[Распаковка с zip \* 68](#_Toc211793861)

[itertools 69](#_Toc211793862)

[Функции 76](#_Toc211793863)

[Docstring Документирование 76](#_Toc211793864)

[Параметры по умолчанию 77](#_Toc211793865)

[return 77](#_Toc211793866)

[\*args 78](#_Toc211793867)

[\*\*kwargs 79](#_Toc211793868)

[map 80](#_Toc211793869)

[filter 81](#_Toc211793870)

[Lambda-выражения 82](#_Toc211793871)

[Сортировка 83](#_Toc211793872)

[Декораторы 86](#_Toc211793873)

[Декораторы функций 86](#_Toc211793874)

[Типизация декораторов функций 87](#_Toc211793875)

[Декораторы классов 88](#_Toc211793876)

[Типизация декораторов класса 88](#_Toc211793877)

[Встроенные декораторы 89](#_Toc211793878)

[lru\_cache кеширование 89](#_Toc211793879)

[Генераторы 92](#_Toc211793880)

[Генераторы и итераторы 92](#_Toc211793881)

[next 94](#_Toc211793882)

[iter 94](#_Toc211793883)

[Отличия: Итераторы, генераторы и итерируемые объекты 95](#_Toc211793884)

[Импорты / экспорты 96](#_Toc211793885)

[Импорты 96](#_Toc211793886)

[Экспорты 98](#_Toc211793887)

[Перезагрузка модуля reload 99](#_Toc211793888)

[Пакеты и подмодули 101](#_Toc211793889)

[\_\_name\_\_ и \_\_main\_\_ 102](#_Toc211793890)

[Область видимости Scope. LEGB 104](#_Toc211793891)

[global и nonlocal 105](#_Toc211793892)

[Объектно-Ориентированное Программирование ООП 107](#_Toc211793893)

[Классы 107](#_Toc211793894)

[Атрибуты класса 108](#_Toc211793895)

[Собственные атрибуты 108](#_Toc211793896)

[Цепочка атрибутов 110](#_Toc211793897)

[Наследование 112](#_Toc211793898)

[Множественное наследование 113](#_Toc211793899)

[MRO Method Resolution Order 114](#_Toc211793900)

[Полиморфизм 116](#_Toc211793901)

[Абстрактные классы 118](#_Toc211793902)

[Магические методы классов 120](#_Toc211793903)

[Список магических методов: 122](#_Toc211793904)

[Примеры 125](#_Toc211793905)

[Даты datetime 128](#_Toc211793906)

[Объект datetime 128](#_Toc211793907)

[Создание объекта datetime 129](#_Toc211793908)

[Форматирование дат 130](#_Toc211793909)

[Промежуток времени timedelta, арифметика дат 131](#_Toc211793910)

[Часовые пояса 132](#_Toc211793911)

[Обработка ошибок 133](#_Toc211793912)

[Подходы LBYL и EAFP 133](#_Toc211793913)

[try, except, else, finally 135](#_Toc211793914)

[Несколько except 135](#_Toc211793915)

[Тестирование 137](#_Toc211793916)

[Линтеры 137](#_Toc211793917)

[unittest 138](#_Toc211793918)

[PiPY и pip 139](#_Toc211793919)

[pip 139](#_Toc211793920)

[Виртуальное окружение 140](#_Toc211793921)

[Создание пакетов 143](#_Toc211793922)

[Conda 144](#_Toc211793923)

[poetry 146](#_Toc211793924)

[Фикс установки зависимостей в poetry 148](#_Toc211793925)

[Numpy, Pandas, math 149](#_Toc211793926)

[math 149](#_Toc211793927)

[numpy 153](#_Toc211793928)

[Массивы numpy 153](#_Toc211793929)

[pandas 157](#_Toc211793930)

[Series 157](#_Toc211793931)

[Индексация, срезы и арифметика 158](#_Toc211793932)

[Фильтрация Series 158](#_Toc211793933)

[Атрибуты Series 159](#_Toc211793934)

[DataFrame 160](#_Toc211793935)

[Индексация и срезы DataFrame 162](#_Toc211793936)

[Методы для работы с DataFrame 163](#_Toc211793937)

[Чтение и запись в файл 165](#_Toc211793938)

[matplotlib 166](#_Toc211793939)

[requests 167](#_Toc211793940)

[Фикс ошибки сертификатов 167](#_Toc211793941)

[Пример запроса 167](#_Toc211793942)

[Линтеры 168](#_Toc211793943)

[Настройка pre-commit 168](#_Toc211793944)

[Список линтеров 169](#_Toc211793945)

[pyproject.toml 170](#_Toc211793946)

[Конфигурация .pre-commit-config.yaml 171](#_Toc211793947)

[Фикс ошибки pylint UnicodeDecodeError 177](#_Toc211793948)

[Плагины VSCode 178](#_Toc211793949)

# Определение Python

**Python** – это объектно-ориентированный язык программирования со строгой динамической типизацией.

Под **«строгой»** подразумевается, что язык не производит неявные преобразования типов и не создаёт проблем при их случайном смешении. То есть нельзя совершать математические действия с разными типами данных.

Под **«динамической»** подразумевается, что типы объектов определяются в процессе исполнения программы (runtime). Поэтому типы переменных указывать не обязательно, но это хороший тон.

Динамическая типизация значит, что, допустим, объявив переменную x = 1, потом её значение можно поменять на x = [a, b, c, d]. То есть у переменной можно корректировать не только значение в рамках одного типа — допустим, единицу изменить на двойку, — но и менять тип данных с одного на другой.

**Переменные** в Python – это всего лишь указатели на объекты, они не содержат информации о типе. Переменные — это контейнеры для хранения значений, которые могут быть изменены во время выполнения программы.

# Типы данных

|  |  |  |  |
| --- | --- | --- | --- |
| **NoneType** | Экземпляр типа объекта NoneType и особая переменная, которая не имеет целевого значения – эквивалент null в других языках | Неизменяемый ❌ | None |
| **Числа** | | | |
| **int** | Целые числа (положительные и отрицательные) | Неизменяемый ❌ | 42, -7, 1000000 |
| **float** | Числа, которые могут иметь десятичную часть (с плавающей точкой) – вещественные числа | Неизменяемый ❌ | 3.14, -0.5, 1e6 |
| **complex** | Комплексные числа | Неизменяемый ❌ | 3+4j |
| **Булевы Значения** | | | |
| **bool** | Булевы значения | Неизменяемый ❌ | True, False |
| **Символы и строки** | | | |
| **str** | Строка, последовательность символов | Неизменяемый ❌ | "Hello", 'Python' |
| **Последовательности** | | | |
| **list** | Изменяемые упорядоченные коллекции элементов (списки) | Изменяемый ✅ | [1, 2, 3] |
| **tuple** | Неизменяемые упорядоченные коллекции элементов (кортежи) | Неизменяемый ❌ | (1, 2, 3) |
| **Отображения** | | | |
| **dict** | Словарь - ассоциативный массив, пары «ключ-значение», где каждый ключ является уникальным (аналог объектов в JS) | Изменяемый ✅ | {'name': 'Alice', 'age': 30} |
| **Множества** | | | |
| **set** | Неупорядоченная и неиндексированная коллекция уникальных элементов | Изменяемый ✅ | {1, 2, 3} |
| **frozenset** | Функция, которая возвращает неизменяемый объект frozenset, инициализированный элементами из заданного итерируемого объекта | Неизменяемый ❌ | frozenset({1, 2, 3}) |
| **Байты** | | | |
| **bytes** | Бинарные данные – неизменяемая последовательность байтов | Неизменяемый ❌ | b"hello" |
| **bytearray** | Изменяемая последовательность байтов | Изменяемый ✅ | bytearray(b"hello") |
| **memoryview** | Представление буфера | – | memoryview(b’abc’) |

## Мутабельность и иммутабельность

**Переменные** в Python – это всего лишь указатели на объекты, они не содержат информации о типе. Переменные — это контейнеры для хранения значений, которые могут быть изменены во время выполнения программы.

Переменные содержат ссылки (эти **ссылки хранятся в стеке**) на области памяти в куче **heap**, в которых уже хранятся непосредственно значения

Например,

*def* test\_heap\_function() -> None:

    a = 100

    b = a

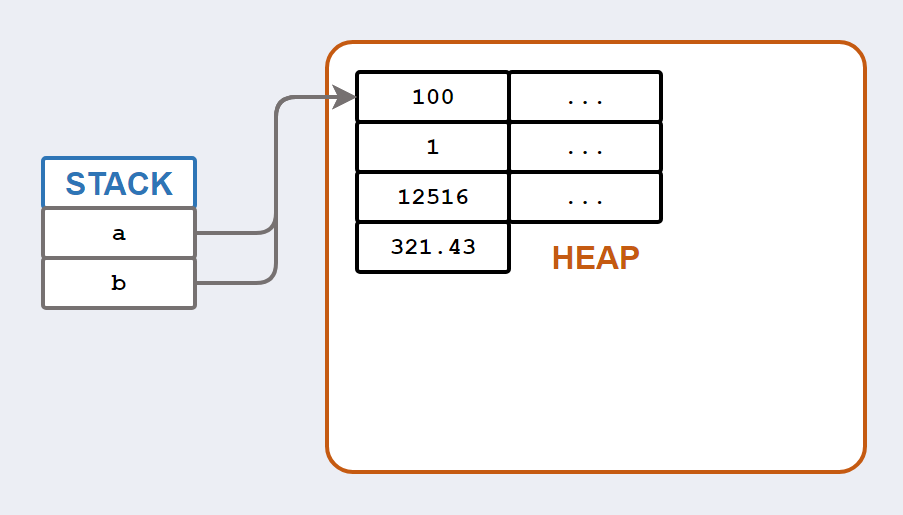
    print(*f*"a: id({id(a)})")

    print(*f*"b: id({id(b)})")

test\_heap\_function()

**ВАЖНО!**

Встроенная функция **id()** возвращает целое число , которое уникально для данного объекта в момент его существования. В реализации CPython (стандартная версия Python) это значение — **адрес объекта в памяти (в head)**. Переменная в Python хранит не сам объект, а ссылку на него, и эта ссылка указывает на область памяти, которую можно получить через **id()**



Если мы попытаемся присвоить переменной, хранящей, например, число, новое значение, то в куче будет выделен новый участок памяти под это новое число, а в область памяти в стеке, на которую указывает ссылка будет помещена новая ссылка:

*def* test\_heap\_function() -> None:

    a = 100

    b = a

    print(*f*"a: id({id(a)})")

    print(*f*"b: id({id(b)})")

    a += 1

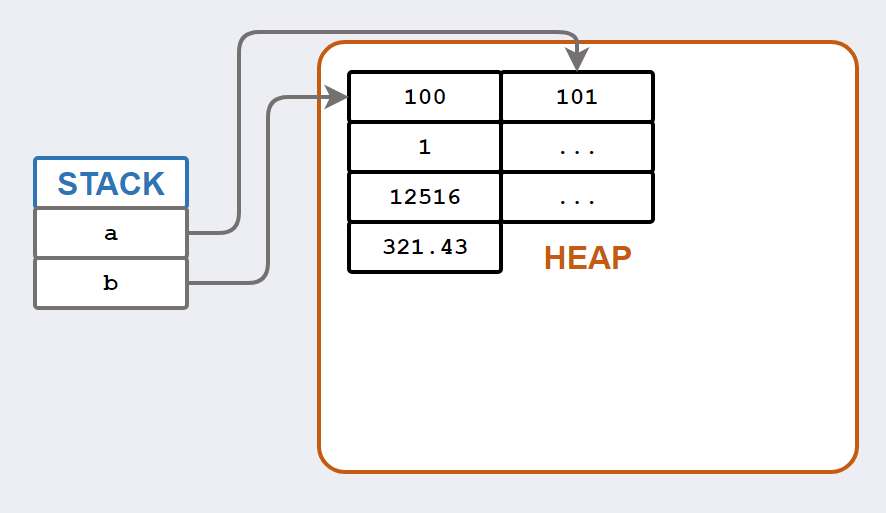
    print(*f*"a: id({id(a)})")

test\_heap\_function()

*# a: id(140088361431424)*

*# b: id(140088361431424)*

*# a: id(140088361431456)*

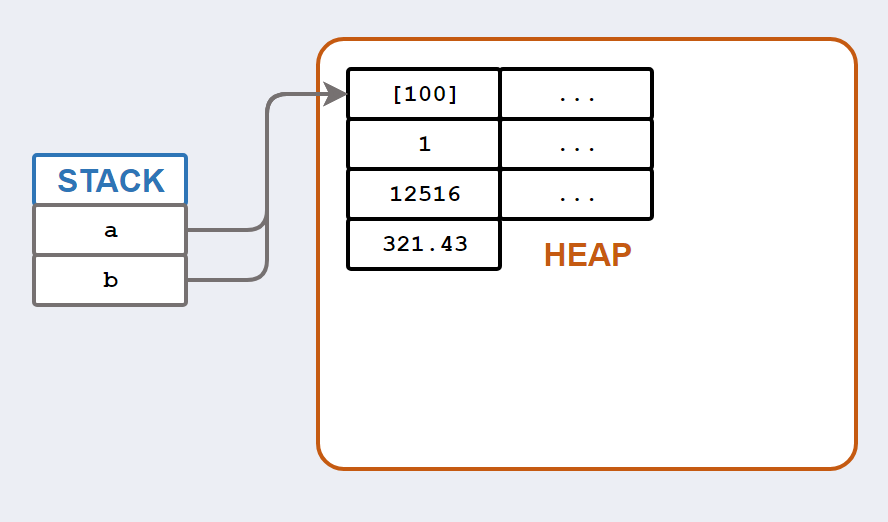


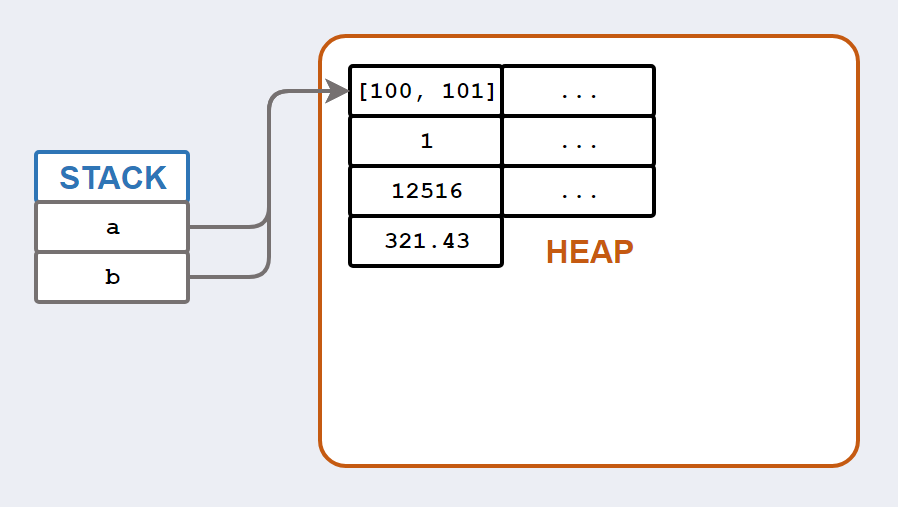
При работе с изменяемыми типами данных происходит изменение имеющегося объекта (списка), например, в него будет добавлена ссылка на новый объект (101)

list\_a = [100]

list\_b = list\_a

list\_a.append(101)

 =====>



# Переменные

1. Не могут начинаться с цифры
2. Без пробелов. Если нужен пробел, то используют \_
3. Нельзя использовать спецсимволы (в том числе -, $, @, % и пр.)

# Описание типов данных

## Числа

Число может быть как положительным, так и отрицательным. В отличие от некоторых других языков, где переменная обычно не бывает длиннее 64 бит, в Python нет ограничения по размеру, есть возможность записать число любой длины. Правда, стоит учесть, что сложение очень длинных чисел может занимать много времени.

* **int** — целые числа

Представляют целые числа (положительные, отрицательные и ноль).

Нет ограничения на размер целого числа (ограничено только объёмом памяти).

* **float** — числа с плавающей запятой

Используются для представления вещественных (дробных) чисел.

* **complex** — комплексные числа
  + Имеют вид a + bj, где a — действительная часть, b — мнимая
  + **j** или **J** обозначает мнимую единицу (√-1)

z = 3 + 4*j*

w = 2*j*

* + Можно получить действительную и мнимую части через атрибуты .real и .imag:

z = 3 + 4*j*

z.real  *# 3.0*

z.imag  *# 4.0*

* Экспоненциальная запись для типа float (это **НЕ** отдельный тип данных)

*# 2 умножить на 10 в степени 3*

d = 2e3

print(d) *# 2000.0*

print(*type*(d)) *# <class 'float'>*

**Кэширование (интернирование)**

Python запоминает числа **от -5 до 256**, которые чаще всего используются в языке. Интернирование нужно для того, чтобы ускорить работу кода и каждый раз не выполнять сложение чисел. То есть используемое в разных частях программы число будет хранится в одной и той же области памяти

x = 10

y = 10

print(id(x) == id(y))  *# True*

Но для больших чисел

x = 1000

y = 1000

print(id(x) == id(y))  *# False*

Для строк тоже работает кэширование

s1 = "hello"

s2 = "hello"

print(id(s1) == id(s2))  *# True*

Но если строка создается динамически, то нет:

s1 = "hello"

s2 = "".join(["h", "e", "l", "l", "o"])

print(id(s1) == id(s2))  *# False*

## Строки

Строка – это набор символов, заключённых в кавычки. Одинарные и двойные кавычки записывают данные в одну строку, а тройные — в несколько. Такой тип данных в Python может содержать любые объекты, например, буквы, цифры или другие данные.

str1 = 'Hello python'

str2 = "Hello python"

str3 = """Hello

python"""

Строки — неизменяемый тип данных, но с ним можно проделывать некоторые манипуляции. Например, есть возможность математически сложить две строки или выполнить оператор +=. Однако в недрах Python при этом создаётся новая строка, то есть первоначальные данные не меняются, а создаются новые. Предложить складывать строки через оператор += при создании цикла неэффективно: на каждой итерации цикла будет создаваться новая строка в новой области памяти. Лучше сделать это через функцию **join**: размер новой строки известен заранее и под нее выделяется нужный объем памяти.

**Индексация и срезы**

Как и в списках, как элементу строки можно обратиться по индексу или использовать срез строки:

Третий параметр в срезе – шаг среза

mystring = "abcdefghijk"

print(mystring[0:3])  *# abc*

*# Вся строка*

print(mystring[:])  *# abcdefghijk*

*# Вся строка*

print(mystring[::])  *# abcdefghijk*

print(mystring[::2])  *# acegik*

print(mystring[::3])  *# adgj*

print(mystring[3:7:2])  *# df*

*# Инверсия строки*

print(mystring[::-1])  *# kjihgfedcba*

**Свойства строк:**

1. Иммутабельность

name = "Sam"

*# Получим ошибку TypeError: 'str' object does not support item assignment*

*# name[0] = "P"*

last\_letters = name[1:]

name1 = "P" + last\_letters

print(name1)  *# Pam*

1. Конкатенация

x = "Hello wolrd"

x = x + "! It's a beatiful world!"

print(x)  *# Hello wolrd! It's a beatiful world!*

1. Умножение строк

x = "abc"

x3 = x \* 3

print(x3)  *# abcabcabc*

**Методы строк**

|  |  |  |
| --- | --- | --- |
| **upper** | Приводит все символы к верхнему регистру. НЕмутирующий |  |
| **lower** | Приводит все символы к нижнему регистру. НЕмутирующий |  |
| **capitalize** | Делает заглавной первую букву. НЕмутирующий |  |
| **title** | Возвращает строку, в которой каждое отдельное слово начинается с буквы в верхнем регистре, а остальные буквы идут в нижнем |  |
| **split** | Разделяет строку на отдельные части. По умолчанию использует в качестве разделителя пробел. Передать пустую строку в качестве разделителя нельзя.  НЕмутирующий | x = "Hello world"  print(x.split())  *# ['Hello', 'world']*  print(x.split(""))  *# ValueError: empty separator*  print(x.split("l"))  *# ['He', '', 'o wor', 'd']*  print(x)  *# Hello world* |
| **count(substr)** | Возвращает количество неперекрывающихся вхождений подстроки substr. | string = "AAAAA"  print(string.count("AA"))  *# 3* |
| **find(substr)** | Возвращает индекс первого вхождения подстроки substr.  Если подстрока не найдена, то *возвращает -1* |  |
| **index(sub)** | Возвращает индекс первого вхождения подстроки sub.  *Вызывает исключение ValueError*, если подстрока не найдена. |  |
| **isalnum** | Возвращает True, если все символы строки являются буквами и цифрами и в строке есть хотя бы один символ. Иначе возвращает False |  |
| **isalpha** | Возвращает True, если все символы строки являются буквами и в строке есть хотя бы один символ. Иначе возвращает False |  |
| **isdigit** | Возвращает True, если все символы строки являются цифрами и в строке есть хотя бы один символ. Иначе возвращает False |  |
| **islower** | Возвращает True, если все буквенные символы строки написаны строчными буквами. Иначе возвращает False |  |
| **isupper** | Возвращает True, если все буквы в строке большие и в строке есть хотя бы одна буква. Иначе возвращает False |  |
| **join(str\_col)** | Возвращает строку, полученную конкатенацией (сложением) строк — элементов коллекции str\_col (обозначение коллекции с элементами типа данных «строка»). Разделителем является строка, для которой вызван метод и цифрами и в строке есть хотя бы один символ. Иначе возвращает False | my\_list = ["a", "b", "c"]  my\_str = "abc"  print("-".join(my\_list))  *# a-b-c*  print("-".join(my\_str))  *# a-b-c* |
| **ljust(width, fillchar)** | Возвращает строку длиной width с выравниванием по левому краю. Строка дополняется справа символами fillchar до требуемой длины. По умолчанию значение fillchar — пробел | my\_str = "text"  print(my\_str.ljust(10, "="))  *# text======* |
| **rjust (width, fillchar)** | Аналогичен ljust, но выравнивание по правому краю | my\_str = "text"  print(my\_str.rjust(10, "="))  *# ======text* |
| **strip(chars)** | Возвращает строку, у которой в начале и в конце удалены символы, встречающиеся в строке chars. Если значение chars не задано, то пробельные символы удаляются |  |
| **lstrip(chars)** | Возвращает строку, у которой в начале удалены символы, встречающиеся в строке *chars*. Если значение *chars* не задано, то пробельные символы удаляются | my\_str = "BCCAstring"  print(my\_str.lstrip("ABC"))  *# string* |
| **rstrip(chars)** | Аналогичен lstrip:  Возвращает строку, у которой в конце удалены символы, встречающиеся в строке chars. Если значение chars не задано, то пробельные символы удаляются | my\_str = "stringBfooCCA"  print(my\_str.rstrip("ABC"))  *# stringBfoo* |
| **split(sep)** | Возвращает список строк по разделителю sep. По умолчанию sep — любое количество пробельных символов |  |
| **startswith(prefix)** | Возвращает True, если строка начинается на подстроку prefix, иначе возвращает False. prefix может быть кортежем проверяемых префиксов строки. |  |
| **endswith(suffix)** | Возвращает True, если строка оканчивается на подстроку suffix |  |
| **zfill(width)** | Возвращает строку, дополненную слева символами «0» до длины width | my\_str = "123"  print(my\_str.zfill(7))  *# 0000123* |

**Форматирование строк**

**Интерполяция –** вставка переменных в строку

|  |  |
| --- | --- |
| **format** | print("У меня есть {} {} яблоко".format("вкусное", "спелое", "красное"))  *# У меня есть вкусное спелое яблоко*  Можно указать позицию:  print("У меня есть {1} {0} яблоко".format("вкусное", "спелое", "красное"))  *# У меня есть спелое вкусное яблоко*  Или указать название переменной:  print("У меня есть {v} {s} яблоко".format(*v*="вкусное", *s*="спелое"))  *# У меня есть вкусное спелое яблоко* |
| **format**  Форматирование чисел | {value:width.precision f}  *width* – ширина, которое должно занимать число (обычно используют 1)  *precision* – количество символов после запятой  result = 100 / 777  print(result)  *# 0.1287001287001287*  print("Результат: {r*:1.3f*}".format(*r*=result))  *# Результат: 0.129*  print("Результат: {r*:10.3f*}".format(*r*=result))  *# Результат:      0.129* |
| **Строковые литералы**  **(f-строки)** | name = "Ivan"  print(*f*"Имя {name}")  *# Имя Ivan* |

### Перенос строки

1. Сложение строк с помощью +

string\_part1 = "This is the first part, "

string\_part2 = "and this is the second part."

full\_string = string\_part1 + string\_part2

print(full\_string)  *# This is the first part, and this is the second part.*

1. Использованием обратного слэша \

Важно:

Автоформаттеры могут автоматически переформатировать такой код и удалять слэш

string\_continued = "This is a long string that we want to " \

                   "split across multiple lines."

print(string\_continued)

1. Несколько строк (*multiline - v*) с помощью тройных кавычек ''' или """

multi\_line\_string = """This is a string that spans

multiple lines. You can write freely

and it will keep the line breaks."""

print(multi\_line\_string)

*# This is a string that spans*

*# multiple lines. You can write freely*

*# and it will keep the line breaks.*

1. Объединение с помощью *join*

strings = [

    "This is the first line.",

    "This is the second line.",

    "This is the third line.",

]

result = "\n".join(strings)  *# Используем перенос строк '\n'*

print(result)

*# This is the first line.*

*# This is the second line.*

*# This is the third line.*

1. Круглые скобки ()

long\_string = (

    "This is a very long string that I would like to "

    "continue on the next line."

)

print(long\_string)

*# This is a very long string that I would like to continue on the next line.*

1. Форматированные строки (f-строки) с использованием скобок

letter\_a = 5

letter\_b = 6

product\_ab = letter\_a \* letter\_b

message = (

*f*"when {letter\_a} is multiplied by {letter\_b}, "

*f*"the result is {product\_ab}"

)

print(message)  *# when 5 is multiplied by 6, the result is 30*

## Списки

В целом работать со списками в Python удобно благодаря тому, что он хранит внутри них не сами элементы, а ссылки на них, а сама ссылка занимает 8 байт. При добавлении новых элементов заполняется выделенная под них память, а когда свободное место заканчивается, его увеличивают с запасом. Это позволяет существенно ускорить расширение списка. Это называется **динамическое расширение** или **reallocation**.

Также существуют **срезы списков** – способ доступа к подмножеству объектов списка. Срез возвращает новый список, содержащий элементы исходного списка в заданном диапазоне индексов, например, slice\_list = my\_list[1:3]

*list* = [1, 'Hello', True, [4,5,6], 3.3]

sliced\_list = list[1:3]

print(sliced\_list) *# ['Hello', True]*

Можно

**Методы для работы со списками**

|  |  |
| --- | --- |
| **len()**  Длина списка | my\_list = [1, 2, 3]  len(my\_list) #3 |
| **\***  Умножение списка | *list* = [0, 1]  list\_x3 = *list* \* 3  print(*list*)  *# [0, 1]*  print(list\_x3)  *# [0, 1, 0, 1, 0, 1]* |
| **Срезы** | my\_list = ["one", "two", "three", "four", "five"]  print(my\_list[1:4])  *# ['two', 'three', 'four']*  print(my\_list[1:5:2])  *# ['two', 'four']*  print(my\_list[::-1])  *# ['five', 'four', 'three', 'two', 'one']* |
| **Конкатенация**  **list1 + list2** | my\_list = ["one", "two", "three"]  new\_list = ["four", "five"]  print(my\_list + new\_list)  *# ['one', 'two', 'three', 'four', 'five']* |
| **list.extend(other\_list)**  Расширяет список list элементами из списка other\_list | list1 = [1, 2, 3]  list2 = [4, 5]  list1.extend(list2)  print(list1)  *# [1, 2, 3, 4, 5]* |
| **append**  Добавление элемента в конец | my\_list = ["one", "two", "three"]  my\_list.append("four")  print(my\_list)  *# ['one', 'two', 'three', 'four']* |
| **pop(index)**  Извлечение элемента  По умолчанию удаляет элементы с конца списка  Можно передать индекс удаляемого элемента | my\_list = ["one", "two", "three"]  popped\_item = my\_list.pop()  print(popped\_item)  *# three*  print(my\_list)  *# ['one', 'two']*  my\_list = ["one", "two", "three"]  popped\_item = my\_list.pop(1)  print(popped\_item)  *# two*  print(my\_list)  *# ['one', 'three']* |
| **del**  Удаляет элемент по индексу. Работает со *срезами* | my\_list = [0, 1, 2, 3, 3, 4, 5, 6]  del my\_list[3]  print(my\_list)  *# [0, 1, 2, 3, 4, 5, 6]*  del my\_list[-1]  print(my\_list)  *# [0, 1, 2, 3, 4, 5, 6]*  del my\_list[::2]  print(my\_list)  *# [1, 3, 5]*  del my\_list[::1]  print(my\_list)  *# []* |
| **insert(index, x)**  Вставляет элемент x в список по индексу index | my\_list = [1, 3, 4]  my\_list.insert(1, 2)  print(my\_list)  *# [1, 2, 3, 4]* |
| **remove(x)**  Удаляет первый элемент со значением x |  |
| **clear**  удаляет все элементы списка |  |
| **copy**  Возвращает копию списка |  |
| **index(x)**  Возвращает индекс первого найденного элемента x. Вызывается исключение ValueError, если элемент не найден |  |
| **count(x)**  Возвращает количество элементов x |  |
| **sort**  Сортировка  *Мутирующий* метод  *Ничего не возвращает*  Можно использовать функцию *sorted()*, которая НЕ мутирует список | my\_list = ["a", "e", "x", "b", "c"]  my\_list.sort()  print(my\_list)  *# ['a', 'b', 'c', 'e', 'x']* |
| **reverse**  Работает аналогично срезу [::-1], но является мутирующим | my\_list = [1, 2, 3, 4, 5]  my\_list.reverse()  print(my\_list)  *# [5, 4, 3, 2, 1]*  my\_list = [1, 2, 3, 4, 5]  print(my\_list[::-1])  *# [5, 4, 3, 2, 1]*  print(my\_list)  *# [1, 2, 3, 4, 5]* |
| **x in list**  Проверят наличие элемента x в списке list | print(3 in [1, 2, 3])  *# True* |
| **x not in list**  Проверят отсутствие элемента x в списке list | print(13 not in [1, 2, 3])  *# True* |

## Кортежи

**Кортежи** — **неизменяемые списки**. Их выбирают для создания статического набора простых данных, который нет необходимости изменять в дальнейшем. Этот тип данных в Python удобен тем, что в нём можно быстро найти элемент по его номеру, потому что они идут по порядку. При этом в произвольное место кортежа нельзя присвоить какое-то значение, как в других видах списков

*tuple* = (4, 1, 2, 3, 13)

print(tuple[4]) *# 13*

Типы элементов в кортеже можно смешивать

items = (1, "Hello", True)

**Методы для работы с кортежами**

|  |  |
| --- | --- |
| **Count**  Сколько раз элемент встречается в кортеже | t = ("a", "a", "a", "b")  print(t.count("a"))  *# 3* |
| **Index**  Индекс первого вхождения элемента в кортеж | t = ("a", "a", "a", "b")  print(t.index("b"))  *# 3* |

## Множества

**Множества** — это неупорядоченные коллекции уникальных элементов. Они создаются с помощью фигурных скобок или функции **set()**. Множества полезны для выполнения операций над коллекциями данных, таких *как объединение, пересечение и разность и др.*

this\_set = {"B", "A", "C"}

При этом в Python есть отдельный вид множества frozenset — это неизменяемый тип данных. В отличие от обычного множества, значения frozenset записываются в двойных фигурных скобках.

frozen\_set = *frozenset*({"B", "A", "C"})

**Особенности множеств:**

* Множество (изменяемое и неизменяемое) можно создать из любого итерируемого объекта

*# Из списка*

set1 = *set*([1, 2, 3, 2]) *# {1, 2, 3}*

*# Из кортежа*

set2 = *set*((1, 2, 3)) *# {1, 2, 3}*

*# Из строки*

set3 = *set*('hello') *# {'e', 'h', 'o', 'l'}*

*# Из словаря*

frozen\_set1 = *frozenset*({"a": 1, "b": 2}) *# frozenset({'b', 'a'})*

*# Из другого set*

frozen\_set2 = *frozenset*({1, 2, 3}) *# frozenset({1, 2, 3})*

*# Из генератора*

frozen\_set3 = *frozenset*(x\*\*2 for x in range(5)) *# frozenset({0, 1, 4, 9, 16})*

* Во множестве можно хранить только неизменяемые объекты (то есть словари, списки, другие множества (кроме frozenset) не могут быть элементами множества).

Множество (set) реализовано на основе **хэш-таблицы**. Оно использует хэш элементов для быстрого поиска и проверки уникальности. Если объект может измениться — его хэш тоже может измениться , и тогда структура данных перестанет работать корректно.

**Методы множеств**

|  |  |
| --- | --- |
| **add(element)** | Добавить элемент во множество |
| **remove(element)** | Удалить элемент множества. Возвращает исключение *KeyError*, если элемент не принадлежит множеству |
| **discard(element)** | Удалить элемент, если он принадлежит множеству |
| **pop()** | Вернуть и удалить произвольный элемент множества |
| **clear()** | Очистить множество, удалив все его элементы |

**Операции над множествами**

|  |  |
| --- | --- |
| **|** или **union()** | Объединяет два множества — результат содержит все элементы из обоих. Повторы исключаются.  s\_1 = {1, 2, 3}  s\_2 = {3, 4, 5}  s\_union = s\_1 | s\_2  *# s\_union = s\_1.union(s\_2)*  print(s\_union)  *# {1, 2, 3, 4, 5}* |
| **&** или **intersection()** | Возвращает только те элементы, которые есть в обоих множествах  s\_1 = {1, 2, 3}  s\_2 = {3, 4, 5}  s\_intersection = s\_1 & s\_2  *# s\_intersection = s\_1.intersection(s\_2)*  print(s\_intersection)  *# {3}* |
| **-** или **difference()** | Возвращает элементы из первого множества, которых нет во втором  s\_1 = {1, 2, 3}  s\_2 = {3, 4, 5}  s\_dif = s\_1 - s\_2  *# s\_dif = s\_1.difference(s\_2)*  print(s\_dif)  *# {1, 2}* |
| **^** или **symmetric\_difference()** | Возвращает элементы, которые есть только в одном из двух множеств (в первом или втором, но не в обоих)  s\_1 = {1, 2, 3}  s\_2 = {3, 4, 5}  s\_sym\_dif = s\_1 ^ s\_2  *# s\_sym\_dif = s\_1.symmetric\_difference(s\_2)*  print(s\_sym\_dif)  *# {1, 2, 4, 5}* |
| **clear()** | Очистить множество, удалив все его элементы |

**Сравнение множеств**

|  |  |
| --- | --- |
| **==** | Проверяет, содержат ли два множества одни и те же элементы, независимо от порядка.  s\_1 = {1, 2, 3}  s\_2 = {3, 1, 2}  print(s\_1 == s\_2)  *# True* |
| **>= или <=** | Проверяет, входят ли все элементы одного множества в другое.  s\_1 = {1, 2, 3}  s\_2 = {1, 2, 3, 4}  print(s\_1 <= s\_2)  *# True* |

## Словари

Словари — это неупорядоченные коллекции пар "ключ-значение". Они создаются с помощью фигурных скобок и позволяют быстро находить значения по ключам. Словари являются одним из наиболее мощных и гибких типов данных в Python, так как они позволяют хранить и манипулировать данными в виде ассоциативных массивов.

Ключи в словаре должны быть **хэшируемыми** (чаще всего, это неизменяемые типы). Это означает, что объект должен иметь неизменный хэш и поддерживать операцию сравнения **==**.

d = {

    1: "один",

    3.14: "пи"

}

user\_info = {

    "name": "Alice",

    "city": "Moscow"

}

*# Кортежи - только если кортеж содержит иммутабельные (хэшируемые) занчения*

*# {(1, [2, 3]): "плохой ключ"}  # ❌ TypeError: unhashable type: 'list'*

coordinates = {

    (10, 20): "точка A",

    (30, 40): "точка B"

}

groups = {

*frozenset*(["apple", "orange"]): "фрукты",

*frozenset*(["carrot", "potato"]): "овощи"

}

*# В качестве ключей могут использоваться байтовые строки*

data = {

*b*"header": "start",

*b*"footer": "end"

}

*# Переменные, которые будут использоваться как ключи*

key1 = "name"

key2 = 42

key3 = ("x", "y")  *# кортеж — хэшируемый тип*

*# Используем их в словаре*

data = {

    key1: "Alice",

    key2: "Answer to the Ultimate Question of Life, the Universe, and Everything",

    key3: [10, 20]

}

**Важно!**

Чтобы проверить, можно ли использовать объект как ключ, используйте функцию

**hash()**

hash((1, 2))            *# ✅ число*

hash("hello")            *# ✅ число*

hash([1, 2])            *# ❌ TypeError*

hash({"a": 1})          *# ❌ TypeError*

hash(*frozenset*([1, 2]))  *# ✅*

**Сортировка**

Начиная с версии 3.7 в реализации CPython, обычные словари сохраняют порядок элементов. Однако обратите внимание – это зависит от версии и реализации! В самом общем случае обычные словари не хранят порядок элементов - в этом случае, если упорядоченность элементов словаря действительно нужна, то можно использовать специальный объект **ordereddict** – это словари с возможностью явной сортировки.

**Методы для работы со словарями**

**Важно!** Методы **keys, values, items** возвращают итерируемый объект, который можно перебрать в цикле или преобразовать в массив, кортеж и т.д. Этот итерируемый объект будет динамически менять при изменении словаря

prices = {"apple": 1.50, "tomato": 1.99, "orange": 0.99}

keys = prices.keys()

print(keys)  *# dict\_keys(['apple', 'tomato', 'orange'])*

prices["potato"] = 1

print(keys)  *# dict\_keys(['apple', 'tomato', 'orange', 'potato'])*

|  |  |
| --- | --- |
| **keys**  Возвращает итерируемый объект, состоящий из ключей словаря | prices = {"apple": 1.50, "tomato": 1.99, "orange": 0.99}  print(prices.keys())  *# dict\_keys(['apple', 'tomato', 'orange'])*  print(*list*(prices.keys()))  *# ['apple', 'tomato', 'orange']* |
| **values**  Возвращает итерируемый объект, состоящий из значений словаря | prices = {"apple": 1.50, "tomato": 1.99, "orange": 0.99}  print(prices.values())  *# dict\_values([1.5, 1.99, 0.99])*  print(*list*(prices.values()))  *# [1.5, 1.99, 0.99]* |
| **items**  Возвращает итерируемый объект, состоящий из кортежей (ключ, значение) словаря | prices = {"apple": 1.50, "tomato": 1.99, "orange": 0.99}  print(prices.items())  *# dict\_items([('apple', 1.5), ('tomato', 1.99), ('orange', 0.99)])*  print(*list*(prices.items()))  *# [('apple', 1.5), ('tomato', 1.99), ('orange', 0.99)]* |
| **get(key, default)**  Возвращает значение по ключу key. Если ключа нет, то возвращает значение default |  |
| **setdefault**  Установит переданное значение, только если ключ отсуствует в словаре | *# метод .setdefault() проверит есть ли ключ в словаре,*  *# если "да", значение не изменится*  person.setdefault('last name', 'Петров') |
| **len(d)**  Возвращает количество ключей в словаре |  |

|  |  |
| --- | --- |
| **del my\_dict[key]**  Удалить ключ из словаря. Если ключа нет, то вызывается исключение *KeyError*  **del my\_ dict**  Удаляет весь словарь целиком | my\_dict = {"a": 1, "b": 2, "c": 3}  *# Сначала проверяем, есть ли ключ*  if "b" in my\_dict:      del my\_dict["b"]      print("Ключ удален") |
| **pop(key, default)**  Возвращает значение по ключу key и удаляет его из словаря. Если ключа нет, то возвращает default | d = {"a": 1, "b": 2, "c": 3}  x = d.pop("a")  print(x)  *# 1*  y = d.pop("aa", "Default value")  print(y)  *# Default value*  При попытке удалить несуществующий ключ без указания дефолтного значения получим ошибку KeyError  d = {"a": 1, "b": 2, "c": 3}  z = d.pop("aaa")  *# KeyError: 'aaa'* |
| **popitem**  Удаляет последний добавленный элемент и выводит его |  |
| **clear**  Удалить все ключи и значения в словаре |  |
| **гpdate**  Добавить в текущий словарь пары ключ/значение из другого словаря | *# приведем пример того, какими могут быть значения словаря*  person: dict[*str*, *str* | *int* | list[*str*]] = {      "first name": "Иван",      "last name": "Иванов",  }  new\_elements: dict[*str*, *str* | *int*] = {"job": "программист", "experience": 7}  *# и присоединим его к существующему словарю с помощью метода .update()*  person.update(new\_elements)  print(person)  *# {*  *#     "first name": "Иван",*  *#     "last name": "Иванов",*  *#     "job": "программист",*  *#     "experience": 7,*  *# }* |
| **my\_dict.copy()**  Создает копию словаря |  |

## Объект

Под объектами понимают **экземпляры собственных классов**.

**Словарь (ditc)** — это встроенная структура данных, которая хранит пары "ключ-значение" (ключи должны быть хэшируемыми (в том числе неизменными))

У объектов отсутствуют по умолчанию методы словарей key(), values, items и пр.

Например, вот так можно перебрать собственные свойства объекта с помощью свойства **\_\_dict\_\_**

*class* Circle:

    pi = 3.14

*def* \_\_init\_\_(*self*, *radius*=1):

*self*.radius = radius

*# self.area = (radius\*\*2) \* self.pi*

*self*.area = (radius\*\*2) \* Circle.pi

my\_circle = Circle(30)

print(my\_circle.pi)  *# 3.14*

*# Т.к. у my\_circle есть собственное свойство pi, у my\_circle2 pi доступно только по цепочке атрибутов из класса*

for key in my\_circle.\_\_dict\_\_:

    print(key)  *# radius area*

## Диапазоны

Является функцией, не является отдельным типом данных

Функция **range()** в Python используется для генерации последовательности чисел , и чаще всего применяется в циклах for для повторения действий определённое количество раз.

* Не включает конец *range(1, 5) → 1, 2, 3, 4*
* Только целые числа *range(1.5, 5) ❌ — вызовет ошибку*
* Нельзя изменять элементы
* Экономит память – *range(1\_000\_000)* занимает мало памяти, т.к. хранит только начало, конец и шаг

**Только конец**

По умолчанию началом будет 0

range(5)

for i in range(5):

    print(i) *# 0 1 2 3 4*

**Начало и конец**

range(2, 7)  *# Генерирует: 2, 3, 4, 5, 6*

**С указанием шага**

for i in range(0, 10, 2):

    print(i)  *# 0 2 4 6 8*

**Обратный порядок (отрицательный шаг)**

for i in range(10, 0, -2):

    print(i)  *# Генерирует: 10, 8, 6, 4, 2*

**Преобразование в список**

print(*list*(range(1, 10, 2)))  *# [1, 3, 5, 7, 9]*

**Индексы списка**

words = ["cat", "window", "defenestrate"]

for i in range(len(words)):

    print(*f*"Слово {i}: {words[i]}")

# Работа с файлами и вводом

## stdin поток ввода

**stdin** — это стандартный поток ввода (standard input). Это один из трёх основных потоков ввода-вывода в программе (наряду с stdout и stderr).

Это канал, через который программа получает входные данные. По умолчанию — это ввод с клавиатуры (терминал).

stdin — это *итератор*, по его строкам можно пройти в цикле

from sys import stdin

lines = []

for line in stdin:

    lines.append(line)

print(lines)

Чтобы завершить ввод:

* в Windows — нажмите Ctrl + Z, затем Enter
* в Linux и macOS — нажмите Ctrl + D

Или так:

import sys

print("Введите текст (для завершения — пустая строка):")

text = ""

for line in sys.stdin:

    if line.strip() == "":

        break

    text += line

print("Вы ввели:")

print(text)

**Чтение одной строки** (пока ввод не будет прерван Ctrl + Z или по другому)

from sys import stdin

text = stdin.read()

print([text])

*# Пример вывода:*

*# ['Первая\nВторая\nТретья\n']*

**Чтение всех строк сразу**

from sys import stdin

lines = stdin.readlines()

print(lines)

*# Пример вывода:*

*# ['Первая\n', 'nВторая\n', 'nТретья\n']*

Можно использовать два вида циклов:

for line in stdin:

for line in stdin.readlines():

Разница в том, что в первом случае стоки читаются по одной, а во втором случае загружается в память сразу весь массив строк

**Чтение в цикле по одной строке**

from sys import stdin

for line in stdin:

    line = line.rstrip("\n")

    print(line)

**Ввод из файла**

Команда для запуска скрипта (передаем файл в скрипт)

**python script.py < input.txt**

from sys import stdin

for line in stdin:

    print("Получено:", line.strip())

## open

**Важно**: в Windows по умолчанию используется кодировка cp1251, поэтому нужно всегда явно указывать кодировку UTF-8, если вы хотите, чтобы программа работала кросс-платформенно.

**open** открывает файл для работы с ним

Принимает в том числе *mode* – режим работы с файлом

По умолчанию передается

*r* – *reading*

open("myfile.txt", *mode*="r", *encoding*="utf-8")

или просто

open("myfile.txt", *encoding*="utf-8")

Открыть файл в текущей директории

myfile = open("myfile.txt")

print(myfile) *# <\_io.TextIOWrapper name='myfile.txt' mode='r' encoding='cp1251'>*

**Файлы могут иметь разные кодировки**. Кодировка файла зависит от операционной системы. Если файл читается неправильно, можно явно указать кодировку

**Важно**: в Windows по умолчанию используется кодировка cp1251, поэтому нужно всегда явно указывать кодировку UTF-8, если вы хотите, чтобы программа работала кросс-платформенно.

myfile = open("myfile.txt", *encoding*="utf-8")

print(myfile)  *# <\_io.TextIOWrapper name='myfile.txt' mode='r' encoding='utf-8'>*

Открыть файл в другой директории

Для этого надо указать абсолютный путь к файлу.

*Windows* \\ (слэшей два, т.к. первый для экранирования)

myfile = open("C:\\materials\\python\\myfile.txt", *encoding*="utf-8")

print(myfile)  *# <\_io.TextIOWrapper name='C:\\materials\\python\\myfile.txt' mode='r' encoding='utf-8'>*

*MacOS* или *Linux* /

myfile = open("/Users/YourUsername/Folder/myfile.txt", *encoding*="utf-8")

**Построчное чтение файла**

file\_in = open("input\_1.txt", *encoding*="UTF-8")

for line in file\_in:

    print(line)

file\_in.close()

**Режимы open**

Принимает в том числе *mode* – режим работы с файлом

* *r* – *reading* по умолчанию

open("myfile.txt", *mode*="r", *encoding*="utf-8")

или просто

open("myfile.txt", *encoding*="utf-8")

* *w – writing – п*ерезаписывает файл или создает новый
* *a – append –* только добавление информации в конец файла
* *r+ – чтение и запись*
  + Файл должен существовать.
  + Указатель на начало файла (смещён в начало).
  + Можно читать и писать в файл.
  + Запись начинается с начала файла, перезаписывая содержимое, если новая информация короче старой *–* остаток остаётся.
* *w+ – запись и чтение* (перезаписывает файл или создает новый)
  + Если файл существует, он очищается (все данные удаляются).
  + Если файла нет, создаётся новый.
  + Указатель на начало файла.
  + Подходит для полной перезаписи файла или создания нового.
* *a+* – *чтение и добавление*.

Файл открывается в режиме добавления, но можно также его читать. При записи — курсор всегда в конце.

## close

Если после открытия файл не закрыть, то он будет заблокирован (например, не получится удалить)

myfile = open("myfile.txt", *encoding*="utf-8")

line1 = myfile.readline()

print(line1)  *# Первая строка*

myfile.close()

## Чтение файла

|  |  |
| --- | --- |
| **read**  Прочитать файл  В качестве аргумента принимает количество символов, которое надо прочитать. Если не передавать – будет прочтен весь файл | myfile = open("myfile.txt", *encoding*="utf-8")  print(myfile.read())  *# Первая строка*  *# Вторая строка*  *# Третья строка*  Но на самом деле строка будет содержать и спецсимволы  myfile = open("myfile.txt")  text = myfile.read())  print([ text ])  *# ['Первая строка\nВторая строка\nТретья строка\n']*  myfile = open("myfile.txt")  text = myfile.read(5))  print(text) *# Перва* |
| **readlines** | Возвращает массив строк из прочитанного файла  myfile = open("myfile.txt", *encoding*="utf-8")  print(myfile.readlines())  *# ['Первая строка\n', 'Вторая строка\n', 'Третья строка\n']* |
| **readline** | Возвращает текущую строку (видимо ту, на которой указатель находится в данный момент)  myfile = open("myfile.txt", *encoding*="utf-8")  line1 = myfile.readline()  print(line1)  *# Первая строка*  line2 = myfile.readline()  print(line2)  *# Вторая строка*  myfile.seek(0)  line3 = myfile.readline()  print(line3)  *# Первая строка* |
| **seak**  Переместить указатель | После чтения файла указатель оказывается в конце. Чтобы прочитать файл снова, надо переместить указатель в начало  myfile = open("myfile.txt", *encoding*="utf-8")  print(myfile.read())  *# Моя текстовая информация* print(myfile.read())  *# Пустая строка*  myfile.seek(0)  print(myfile.read())  *# Моя текстовая информация* |

## Менеджер контекста with

Автоматически закрывает **ресурсы** **(например, файлы, соединения с БД, сетевые подключения и т.д.)** после **завершения блока**

После работы с файлом (*и не только с файлом*) его необходимо закрыть с помощью метода close(). Это особенно важно, потому что:

* незакрытый файл может остаться заблокированным для других программ;
* данные могут не записаться полностью;
* при ошибке в программе вызов close() может не выполниться, и файл останется открытым.

Чтобы избежать этих проблем, используйте менеджер контекста — конструкцию **with**. Она гарантирует, что файл будет закрыт автоматически, даже если в процессе работы произойдёт ошибка:

with open("myfile.txt", *encoding*="utf-8") as myfile:

    line1 = myfile.readline()

print(line1)  *# Первая строка*

print(myfile)  *# <\_io.TextIOWrapper name='myfile.txt' mode='r' encoding='utf-8'>*

line2 = myfile.readline()  *# ValueError: I/O operation on closed file.*

## Запись в файл

**write**

запись строки в файл. Возвращает количество записанных символом

with open("output.txt", "w", *encoding*="UTF-8") as file\_out:

    amount = file\_out.write(

        "Это первая строка\nА вот и вторая\nИ третья -- последняя\n"

    )

print(amount)  *# 55*

*# output.txt*

*#*

*# Это первая строка*

*# А вот и вторая*

*# И третья -- последняя*

**writelines**

Запись строк из списка строк

lines = ["Это первая строка\n", "А вот и вторая\n", "И третья - последняя\n"]

with open("output.txt", "w", *encoding*="UTF-8") as file\_out:

    file\_out.writelines(lines)

*# output.txt*

*#*

*# Содержимое выходного файла:*

*# Это первая строка*

*# А вот и вторая*

*# И третья - последняя*

**print**

with open("output.txt", "w", *encoding*="UTF-8") as file\_out:

    print("Вывод в файл с помощью функции print()", *file*=file\_out)

*# output.txt*

*#*

*# Вывод в файл с помощью функции print()*

# JSON

Для работы с форматом json используется модуль json

**Чтение JSON-файла**

Метод **load**

import json

from pprint import pprint

with open("data.json", *encoding*="UTF-8") as file\_in:

    records = json.load(file\_in)

pprint(records)

**Запись JSON-файла**

Метод **dump**

dump принимает следующие параметры:

|  |  |
| --- | --- |
| **ensure\_ascii** | * True (по умолчанию) – все **не-ASCII**-символы сохраняются как \uXXXX * False – сохраняются как есть (нужно для букв на кириллице) |
| **indent** | Задаёт отступы для форматирования   * None (по умолчанию) – весь JSON в одну строку * число – количество пробелов на уровень вложенности |
| **sort\_keys** | если True, ключи в словаре будут отсортированы |

import json

with open("data.json", *encoding*="UTF-8") as file\_in:

    records = json.load(file\_in)

records[1]["group\_number"] = 2

with open("data.json", "w", *encoding*="UTF-8") as file\_out:

    json.dump(records, file\_out, *ensure\_ascii*=False, *indent*=2)

# Операторы и функции

## Операторы сравнения

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Пример** |
| **==** | Если значения двух операндов равны, то условие становится true | (a == b) не равно true. |
| **!=** | Если значения двух операндов не равны, то условие становится true. | (a != b) равно true. |
| **>** | Если значение левого операнда больше значения правого операнда, то условие становится true. | (a > b) не равно true. |
| **<** | Если значение левого операнда меньше значения правого операнда, то условие становится true. | (a < b) равно true. |
| **>=** | Если значение левого операнда больше или равно значению правого операнда, то условие становится true. | (a >= b) не равно true. |
| **<=** | Если значение левого операнда меньше или равно значению правого операнда, то условие становится true. | (a <= b) равно true. |

## Логические операторы

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Пример** |
| **and** | Возвращает True, если оба операнда равны True, и False в противном случае | 2 > 1 and 2 < 3  True  2 > 1 and 2 > 3  False |
| **or** | Возвращает True, если хотя бы один из операндов равен True, и False, если оба операнда – False | 2 > 1 or 2 > 3  True |
| **not** | Возвращает False, если операнд – True, и True, если операнд – False. Унарный оператор – имеет только 1 операнд | not 2 > 3  True  not 1 == 1  False |

## Операторы if, elif и else

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Пример** |
| **if** | Пробелы и отступы важны | hungry = True  if hungry:      print("FEED ME")  *# FEED ME* |
| **if else** |  | hungry = False  if hungry:      print("FEED ME")  else:      print("Im not hungry")  *# Im not hungry* |
| **If elif** |  | location = "Bank"  if location == "Auto Shop":      print("Cars are cool")  elif location == "Bank":      print("Money is cool")  *# Money is cool*  elif location == "Store":      print("Welcome to the store")  else:      print("Everything is cool") |

### Операции для объединения условий

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Пример** |
| **and** |  | x > 0 and x < 10 |
| **or** |  | x < 0 or x > 100 |
| **not** | Аналогично != | not x == 0 |

Объединение условий:

if 0 <= x < 100:

    pass

## match-case

*def* http\_error(*status*):

    match status:

        case 400:

            return "Bad request"

        case 404:

            return "Not found"

        case 418:

            return "I'm a teapot"

        case \_:

            return "Something's wrong with the internet"

или

color = input()

match color:

    case 'красный' | 'жёлтый':

        print('Стоп.')

    case 'зелёный':

        print('Можно ехать.')

    case \_:

        print('Некорректное значение.')

## Циклы

### for

Цикл *for* в Python используется для итерации (перебора) элементов из последовательности — например, списка, строки, кортежа, словаря, множества или любого итерируемого объекта.

**Перебор массива**

my\_iterable = [1, 2, 3]

for item\_name in my\_iterable:

    print(item\_name)  *# 1 2 3*

**Перебор строки**

for letter in "Python":

    print(letter)  *# p y t h o n*

**Перебор словаря**

По умолчанию перебираются *только ключи*. Чтобы перебрать пары ключ:значение, надо использовать *items()*

person = {"name": "Alice", "age": 30, "city": "New York"}

for key in person:

    print(key, ":", person[key])

*# name : Alice*

*# age : 30*

*# city : New York*

Перебор ключей и значений

person = {"name": "Alice", "age": 30, "city": "New York"}

for key, value in person.items():

    print(*f*"{key} : {value}")

*# name : Alice*

*# age : 30*

*# city : New York*

**Перебор range (классический цикл)**

При переборе диапазона участвует начало, но не участвует конец диапазона (то есть перебирается, не включая 10 в примере ниже)

for i in range(0, 10, 2):

    print(i)  *# 0 2 4 6 8*

Обратный перебор

for i in range(10, 0, -2):

    print(i)  *# 8 6 4 2*

|  |  |  |
| --- | --- | --- |
| **break** | Прервать цикл | for i in range(10):      if i == 5:          break      print(i)  *# 0 1 2 3 4* |
| **continue** | Пропустить текущую итерацию | for i in range(5):      if i == 2:          continue      print(i)  *# 0 1 3 4* |
| **else** | выполнится после завершения цикла *(но не при break)*  Выполнится, даже если мы не попали в цикл:  i = 6  while i < 5:      print(i)  else:      print("Цикл завершен")  *# Цикл завершен* | for i in range(3):      print(i)  else:      print("Цикл завершён")  *# 0 1 2*  *# Цикл завершён* |
| **pass** | Ничего не делает. Используется в качестве заглушки для кода, который будет написан позже.  Это пустая инструкция, используемая в тех местах, где синтаксически требуется наличие кода, но вы пока не хотите его писать.  Используйте pass, когда вы осознанно оставляете место под будущий код.  Используется также в if, функциях, классах и т.д | i = 1  while i < 5:      pass  *# Поззже здесь будет написан код* |

### while

Повторное выполнения блока кода, пока выполняется некоторое условие

|  |  |  |
| --- | --- | --- |
| **break** | Прервать цикл |  |
| **continue** | Пропустить текущую итерацию |  |
| **else** | выполнится после завершения цикла *(но не при break)* |  |
| **pass** | Ничего не делает. Используется в качестве заглушки для кода, который будет написан позже.  Это пустая инструкция, используемая в тех местах, где синтаксически требуется наличие кода, но вы пока не хотите его писать.  Используйте pass, когда вы осознанно оставляете место под будущий код.  Используется также в if, функциях, классах и т.д | i = 1  while i < 5:      pass  *# Поззже здесь будет написан код* |

## Моржовый оператор :=

**walrus operator / Оператор присваивания с выражением**

Появился в **Python 3.8**

Позволяет **одновременно присвоить значение переменной и вернуть это значение в выражении**. Это помогает избежать повторных вычислений или дублирования кода.

if line := input("Введите что-нибудь: "):

    print(*f*"Вы ввели: {line}")

while (name := input("Введите имя: ")) != "СТОП":

    print(*f*"Привет, {name}!")

print("Программа завершена.")

filtered = [y for x in data if (y := func(x)) > 0]

## enumerate

**Функция enumerate()** используется для итерации по последовательности (список, строка, кортеж и т.д.) с автоматическим отслеживанием индекса элементов.

**enumerate()** принимает *итерируемый объект* (iterable) — например, список, строку, кортеж и т.д.,и возвращает *итератор* — объект, который генерирует пары (индекс, элемент) по мере итерации. Эти пары являются кортежами.

fruits = ["apple", "banana", "cherry"]

for index, fruit in enumerate(fruits, *start*=1):

    print(index, fruit)

*# 0 apple*

*# 1 banana*

*# 2 cherry*

Может принимать начальное значение индекса

fruits = ["apple", "banana", "cherry"]

for index, fruit in enumerate(fruits, *start*=1):

    print(index, fruit)

*# 1 apple*

*# 2 banana*

*# 3 cherry*

Каждая пара представляет собой *кортеж*

fruits = ["apple", "banana", "cherry"]

for pair in enumerate(fruits, *start*=1):

    print(*type*(pair))  *# <class 'tuple'>*

## List comprehension Генераторы списков

Представляют собой более короткую запись для генерации списков из какой-нибудь последовательности

Например, для генерации списка из строки мы могли бы использовать такую конструкцию

mystring = "hello"

mylist = []

for letter in mystring:

    mylist.append(letter)

print(mylist)  *# ['h', 'e', 'l', 'l', 'o']*

или такую

print(*list*("hello"))  *# ['h', 'e', 'l', 'l', 'o']*

Python позволяет нам использовать более короткий способ с помощью генераторов списков:

mystring = "hello"

mylist = [letter for letter in mystring]

print(mylist)*#['h', 'e', 'l', 'l', 'o']*

Можно выполнять другие операции:

*# # Перевод градусов в фаренгейты*

celsius = [0, 10, 20, 34.5]

fahrenheit = [((9 / 5) \* temp + 32) for temp in celsius]

print(fahrenheit)  *# [32.0, 50.0, 68.0, 94.1]*

Или использовать условие IF:

*# Использование условия if*

mylist = [num\*\*2 for num in range(0, 10) if num % 2 == 0]

print(mylist)  *# [0, 4, 16, 36, 64]*

Или даже IF ELSE (но сигнатура немного отличается):

*# Использование условия IF ELSE*

*# Возведем в квадрат все четные числа или увеличиваем на 100 все нечетные*

results = [x if x % 2 == 0 else x + 100 for x in range(0, 10)]

print(results)  *# [0, 101, 2, 103, 4, 105, 6, 107, 8, 109]*

Или работать с вложенными циклами:

*# Вложенные циклы*

mylist = []

for x in [2, 4, 6]:

    for y in [1, 10, 1000]:

        mylist.append(x \* y)

mylist = [x \* y for x in [2, 4, 6] for y in [1, 10, 1000]]

print(mylist)  *# [2, 20, 2000, 4, 40, 4000, 6, 60, 6000]*

## Generator Expressions Выражения-генераторы

Похожи на list comprehensions, но не создают список в памяти, а возвращают объект-генератор.

squares = (x \* x for x in range(10))

print(next(squares))  *# 0*

print(next(squares))  *# 1*

Подробнее про генераторы [Генераторы](#_Генераторы)

## Dictionary comprehension Генератор словаря

{key\_expression: value\_expression for item in iterable}

countries = {

    country: capital

    for country, capital in [

        ("Россия", "Москва"),

        ("Беларусь", "Минск"),

        ("Сербия", "Белград"),

    ]

}

print(countries)

"""

{

'Россия': 'Москва',

'Беларусь': 'Минск',

'Сербия': 'Белград'

}

"""

## Set comprehension Генераторы множеств

*# Создать множество нечетных чисел из массива numbers*

numbers = [1, 2, 3, 2, 1, 2, 3, 2, 1, 2]

my\_set = {num for num in numbers if num % 2 == 1}

print(my\_set)  *# {1, 3}*

## in

**in** используется для проверки, содержится ли значение в какой-либо последовательности (например, списке, строке, кортеже, словаре и т.д.).

"ba" in "banana"  *# True*

"f" in "banana"  *# False*

"banana" in ["apple", "banana", "cherry"]  *# True*

"mykey" in {"mykey": 123}  *# True*

123 in {"mykey": 123}.values()  *# True*

print(("mykey", 123) in {"mykey": 123}.items())  *# True*

Может быть медленным. Python последовательно перебирает каждый элемент, пока не найдёт нужный. Это называется линейный поиск. Поэтому для больших и частых проверок лучше использовать *set*.

## Математические функции

**min()**

Принимает или итерируемый объект, или набор аргументов (если передать словарь, то будет перебирать ключи)

my\_list = [1, 2, 3]

print(min(my\_list))  *# 1*

print(min(1, 2, 3))  *# 1*

**max()**

Принимает или итерируемый объект, или набор аргументов

**random**

Это библиотека

|  |  |
| --- | --- |
| **Shuffle**  Мутирующий | «Тасует» переданную последовательность (чаще всего массив)  from random import shuffle  my\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  shuffle(my\_list)  print(my\_list)  *# [4, 10, 2, 8, 1, 5, 9, 6, 7, 3]* |
| **randint** | Случайное целое число из диапазона *(включая границы диапазона)*  from random import randint  print(randint(0, 1000))  *# 973* |
| **randrange** | Случайное целое число из диапазона *(включая нижнюю границу, но не включая верхнюю границу диапазона)*  from random import randrange  print(randrange(0, 10, 2))  *# 0, 2, 4, 6 или 8*  print(randrange(0, 10, 5))  *# 5 или 0* |

## Ввод данных

**input()**

result = input()  *# введем 30*

print(result)  *# 30*

*type*(result)  *# <class 'str'>*

*type*(*int*(result))  *# <class 'int'>*

*type*(*float*(result))  *# <class 'float'>*

## zip

**Функция zip()** — это удобный инструмент для объединения нескольких итерируемых объектов (например, списков, кортежей) в один итератор, который возвращает кортежи (пары, тройки, четвёрки и т.д.) элементов по их порядковому номеру.

*zip ориентируется на самый короткий* из переданных объектов, остальные элементы в более длинных итерируемых объектах будут проигнорированы

**Возвращает:**

* Возвращает объект типа zip, который является итератором.
* Он генерирует *кортежи*, где каждый кортеж содержит элементы из соответствующих позиций переданных итерируемых объектов.

names = ["Alice", "Bob", "Charlie"]

ages = [25, 30, 35, 49]

cities = ("Moscow", "Berlin", "Paris", "New-York")

for name, age, city in zip(names, ages, cities):

    print(*f*"{name} — {age} years. Lives in {city}")

*# Alice — 25 years. Lives in Moscow*

*# Bob — 30 years. Lives in Berlin*

*# Charlie — 35 years. Lives in Paris*

names = ["Alice", "Bob", "Charlie"]

ages = [25, 30, 35, 49]

cities = ("Moscow", "Berlin", "Paris", "New-York")

my\_list = *list*(zip(names, ages, cities))

print(my\_list)

*# [('Alice', 25, 'Moscow'), ('Bob', 30, 'Berlin'), ('Charlie', 35, 'Paris')]*

Чтобы избежать неожиданных сокращений (когда итерируемые объекты имеют разную длину), можно включить проверку **strict=True**:

print(*list*(zip("ABCDE", [1, 2, 3], *strict*=True)))

*# Вывод:*

*# ValueError: zip() argument 2 is shorter than argument 1*

### Распаковка с zip \*

Если нужно вернуть исходные последовательности из списка кортежей — используйте звёздочку **\***:

my\_list = (["Alice", 31], ["Mark", 33], ["Ivan", 39])

names, ages = zip(\*my\_list)

print(names)  *# ('Alice', 'Mark', 'Ivan')*

print(ages)  *# (31, 33, 39)*

Это эквивалентно:

names, ages = zip(["Alice", 31], ["Mark", 33], ["Ivan", 39])

print(names)  *# ('Alice', 'Mark', 'Ivan')*

print(ages)  *# (31, 33, 39)*

То есть, это не какая-то «обратная операция»:

zip откработает точно так же: берет первые элементы и «сшивает» их в один кортеж, затем вторые элементы и т.д.

Можно передавать список списков, кортеж список и прочие комбинации итераторов

# itertools

**itertools** — стандартной библиотекой Python, созданной специально для обработки коллекций.

С помощью itertools можно строить **итераторы**, которые:

* экономят память (не создают список целиком);
* позволяют легко комбинировать и фильтровать данные;
* ускоряют работу с коллекциями в циклах.

**Важно**! почти все функции itertools возвращают **итераторы, а не списки**. Их можно обойти в *цикле* или преобразовать в список с помощью *list().*

|  |  |
| --- | --- |
| **count()** | **Cчётчик с шагом**  Создаёт бесконечную числовую последовательность. Можно задать старт и шаг. Работает с целыми и вещественными числами. |
| from itertools import count  for value in count(0, 0.1):      if value > 1:          break      print(round(value, 1))  *# 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0* | |
| **cycle()** | Зацикливание коллекции  Бесконечно перебирает элементы переданной коллекции.  Полезно, когда нужно «крутить» одно и то же в цикле |
| from itertools import cycle  s = ""  for letter in cycle("ABC"):      if len(s) == 10:          break      s += letter  print(s)  *# ABCABCABCA* | |
| **repeat()** | Повтор одного значения  Повторяет одно и то же значение. Можно указать, сколько раз. |
| from itertools import repeat  print(*list*(repeat("ABC", 5)))  *# ['ABC', 'ABC', 'ABC', 'ABC', 'ABC']*  *# Аналогично*  print(["ABC"] \* 5)  *# ['ABC', 'ABC', 'ABC', 'ABC', 'ABC']* | |
| **accumulate()** | **Накопление значений**  Создаёт итератор, в котором каждое значение — это сумма всех предыдущих. Полезно, если нужно посчитать прогресс, баллы или накопленные значения в процессе. |
| from itertools import accumulate  for value in accumulate([1, 2, 3, 4, 5]):      print(value)  *# Вывод:*  *# 1, 3, 6, 10, 15* | |
| **chain()** | **Объединение коллекций**  Позволяют объединить несколько итерируемых объектов в один. Это удобнее и быстрее, чем склеивать списки вручную. |
| from itertools import chain  values = *list*(chain("АБВ", "ГДЕ", "ЖЗИ"))  print(values)  *# Вывод программы:*  *# ['А', 'Б', 'В', 'Г', 'Д', 'Е', 'Ж', 'З', 'И']* | |
| **chain.from\_iterable()** | **Объединение коллекций**  Позволяют объединить несколько итерируемых объектов в один. Это удобнее и быстрее, чем склеивать списки вручную.  Если у вас список строк, можно использовать *chain.from\_iterable()* |
| from itertools import chain  values = *list*(chain.from\_iterable(["АБВ", "ГДЕ", "ЖЗИ"]))  print(values)  *# Вывод программы:*  *# ['А', 'Б', 'В', 'Г', 'Д', 'Е', 'Ж', 'З', 'И']* | |
| **product()** | **Декартово произведение**  Создаёт все возможные упорядоченные комбинации элементов из нескольких коллекций. Это называется **декартовым произведением**.  Полезна при генерации вариантов:   * комбинаций настроек * паролей * расписаний * всех пар ключей и значений и др. |
| from itertools import product  values = *list*(product([1, 2, 3], "АБВГ"))  print(values)  *# Вывод программы:*  *# [(1, 'А'), (1, 'Б'), (1, 'В'), (1, 'Г'), (2, 'А'), (2, 'Б'), (2, 'В'), (2, 'Г'), (3, 'А'), (3, 'Б'), (3, 'В'), (3, 'Г')]* | |
| **product(repeat = N)** | Повторные сочетания с параметром repeat  Значение в repeat означает, сколько раз повторить декартово произведение одного и того же итерируемого объекта.  **repeat=N** означает повторить итерируемые объекты **N** раз  Например, аргумент repeat=2 означает, что каждая коллекция участвует дважды |
| from itertools import product  *# Декартово произведение одного и того же итерируемого объекта 2 раза*  result = *list*(product([1, 2], *repeat*=2))  print(result)  *# [(1, 1), (1, 2), (2, 1), (2, 2)]*  *# Все 4-значные PIN-коды из цифр 0-9*  digits = "0123456789"  pin\_codes = product(digits, *repeat*=4)  *# Показать первые 10*  for i, pin in enumerate(pin\_codes):      if i >= 10:          break      print("".join(pin))  *# 0000 0001 0002 0003 0004 0005 0006 0007 0008 0009*  from itertools import product  values = product([1, 2, 3], "АБВГ", *repeat*=2)  print(*list*(values))  *# Вывод*  *# [(1, 'А', 1, 'А'), (1, 'А', 1, 'Б'), (1, 'А', 1, 'В'), (1, 'А', 1, 'Г'), (1, 'А', 2, 'А'),... ]*  *# АНАЛОГИЧНО ЗАПИСИ*  values = product([1, 2, 3], "АБВГ", [1, 2, 3], "АБВГ")  print(*list*(values)) | |
| **permutations()** | **Перестановки без повторений**  Создаёт все возможные упорядоченные варианты **перестановок из элементов**. По умолчанию длина каждой перестановки равна длине исходного объекта, но можно указать другую длину через аргумент **r** |
| from itertools import permutations  values = permutations("АБВ")  print(*list*(values))  *# Вывод:*  *# [('А', 'Б', 'В'), ('А', 'В', 'Б'), ('Б', 'А', 'В'), ('Б', 'В', 'А'), ('В', 'А', 'Б'), ('В', 'Б', 'А')]*  values = permutations("АБВ", *r*=2)  print(*list*(values))  *# Вывод:*  *# [('А', 'Б'), ('А', 'В'), ('Б', 'А'), ('Б', 'В'), ('В', 'А'), ('В', 'Б')]*  Для сравнения: product работает с повторениями элементов  from itertools import permutations, product  *# Для сравнения: product работает с повторениями элементов*  values = product("АБВ", *repeat*=3)  print(*list*(values))  *# Вывод:*  *# [('А', 'А', 'А'), ('А', 'А', 'Б'), ('А', 'А', 'В'), ('А', 'Б', 'А'), ('А', 'Б', 'Б'), ('А', 'Б', 'В'), ('А', 'В', 'А'), ('А', 'В', 'Б'), ...]* | |
| **combinations()** | **Cочетания без повторений**  Создаёт неупорядоченные комбинации заданной длины r. Элементы внутри кортежей идут по порядку, но не повторяются. Пары вроде ('А', 'Б') и ('Б', 'А') считаются одинаковыми, и в результате будет только одна из них. |
| from itertools import combinations  values = *list*(combinations("АБВ", 2))  print(values)  *# Вывод:*  *# [('А', 'Б'), ('А', 'В'), ('Б', 'В')]* | |
| **combinations\_with**  **\_replacement()** | **Сочетания с повторениями**  Похож на combinations(), но допускает повторы. То есть элементы могут встречаться в одном сочетании больше одного раза. |
| from itertools import combinations\_with\_replacement  values = *list*(combinations\_with\_replacement("АБВ", 2))  print(values)  *# Вывод:*  *# [('А', 'А'), ('А', 'Б'), ('А', 'В'), ('Б', 'Б'), ('Б', 'В'), ('В', 'В')]* | |

# Функции

## Docstring Документирование

*def* say\_hello(*name*="ИМЯ\_ПО\_УМОЛЧАНИЮ"):

    """

    DOCSTRING: функция для приветствия пользователя

    INPUT:

    OUTPU:

    """

    print(*f*"Function was called")

    return *f*"Hello, {name}"

result1 = say\_hello()  *# Hello, ИМЯ\_ПО\_УМОЛЧАНИЮ*

result2 = say\_hello("Ivan")  *# Hello, Ivan*

print(result1)

print(result2)

## Параметры по умолчанию

В Python **не рекомендуется** использовать изменяемые объекты в качестве значений параметров по умолчанию по причине:

1. Значения по умолчанию вычисляются 1 раз при определении функции, а не при каждом вызове;
2. Если использовать изменяемый объект (список, словарь), то изменения в нём будут сохраняться между вызовами функции. Это может привести к неочевидному поведению и трудноуловимым ошибкам при многократном вызове функции;

В качестве альтернативы можно задавать значение по умолчанию None, в теле функции создавать новый изменяемый объект, если значение не передано. Такое решение делает поведение программы понятным и предсказуемым.

*def* test\_function\_two(*listing*=None) -> None:

    if listing is None:

        listing = []

    listing.append(1)

    print(listing)

## return

Функция в Питон может возвращать несколько значений, которые будут объединены в кортеж

*def* is\_only\_even(*numbers*):

    for i, x in enumerate(numbers):

        if x % 2 != 0:

            return False, i

    return True

print(is\_only\_even([2, 4, 6]))  *# True*

print(is\_only\_even([1, 2, 3]))  *# (False, 0)*

## \*args

**\*args** — это специальный синтаксис в Python, который позволяет функции принимать произвольное количество позиционных аргументов.

Слово *args* — это просто общепринятое название (можно использовать любое другое), а звёздочка \* перед ним говорит Python: «Все дополнительные позиционные аргументы собери в кортеж»

**Особенности:**

* **\*args** собирает только *позиционные* аргументы.
* **\*args** собирает параметры в **кортеж**
* После **\*args** могут идти только *именованные* аргументы (**\*\*kwargs**)

Функцию

*def* myfunc(*a*, *b*, *c*=0, *d*=0, *e*=0):

*# Returns 5% of the sum of arguments*

    return sum((a, b, c, d, e)) \* 0.05

print(myfunc(10, 10, 10, 10, 10))  *# 2.5*

Можно заменить на

*def* myfunc1(\**args*):

    print(args)  *# (10, 10, 10, 10, 10)*

    return sum(args) \* 0.05

print(myfunc1(10, 10, 10, 10, 10))  *# 2.5*

## \*\*kwargs

Именованные аргументы позволяют не соблюдать порядок, указанный в определении функции, — если вы явно указываете имена. Это делает вызовы функций более читаемыми. Позиционные аргументы всегда должны **идти первыми**, а именованные — после них.

**\*\*kwargs** – *key word args* – собирает именованные параметры в **словарь** dictionary

Слово *kwargs* — это просто общепринятое название (можно использовать любое другое)

*def* myfunc(\*\**kwargs*):

    if "fruit" in kwargs:

        print("My fruit of choice is {}".format(kwargs["fruit"]))

    else:

        print("I did not find any fruit here")

myfunc(*fruit*="apple")  *# My fruit of choice is apple*

myfunc(*fruit1*="apple")  *# I did not find any fruit here*

Комбинация **args** и **kwargs**

*def* newfunc(\**args*, \*\**kwargs*):

    print(args)  *# (10, 20, 30)*

    print(kwargs)  *# {'fruit': 'orang', 'food': 'eggs'}*

    print(" I would like {} {}".format(args[0], kwargs["food"]))

newfunc(10, 20, 30, *fruit*="orang", *food*="eggs")  *# I would like 10 eggs*

Или даже так (комбинация с обычными параметрами)

*def* newfunc(*value*, \**args*, \*\**kwargs*):

    print(value)  *# 10*

    print(args)  *# (20, 30)*

    print(kwargs)  *# {'fruit': 'orang', 'food': 'eggs'}*

    print(" I would like {} {}".format(args[0], kwargs["food"]))

newfunc(10, 20, 30, *fruit*="orang", *food*="eggs")  *# I would like 20 eggs*

## map

**map** – применяет заданную функцию к каждому элементу итерируемого объекта (например, списка) и возвращает итератор с результатами.

*def* square(*num*):

    return num\*\*2

my\_nums = [1, 2, 3, 4, 5]

for item in map(square, my\_nums):

    print(item)

print(*list*(map(square, my\_nums)))  *# 1 4 9 16 25*

*def* splicer(*mystring*):

    if len(mystring) % 2 == 0:

        return "EVEN"

    return mystring[0]

names = ["Andy", "Eve", "Sally"]

print(*list*(map(splicer, names)))  *# ['EVEN', 'E', 'S']*

## filter

**filter** – фильтрует элементы итерируемого объекта (например, списка), оставляя только те, которые соответствуют заданному условию:

* Принимает функцию, которая должна возвращать *True* или *False*.
* Возвращает итератор, состоящий только из тех элементов, для которых функция вернула *True*.

Чтобы отфильтровать все истинные значения, можно использовать None вместо функции:

values = [0, 1, False, True, "", "hello", None, 42]

truthy\_values = *list*(filter(None, values))

print(truthy\_values)  *# [1, True, 'hello', 42]*

*def* check\_even(*num*):

    return num % 2 == 0

even\_list = *list*(filter(check\_even, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))

print(even\_list)  *# [2, 4, 6, 8, 10]*

## Lambda-выражения

**Lambda-выражение** — это анонимная функция, то есть функция без имени. Она используется, когда нужно выполнить простую операцию, которую нецелесообразно оформлять как полноценную функцию через def.

Эту анонимную функцию можно или сохранить в переменную

square = *lambda* *num*: num\*\*2

print(square(5))  *# 25*

Или сразу передавать как параметр в другую функцию

result = map(*lambda* *num*: num\*\*2, [1, 2, 3, 4, 5, 6, 7])

print(*list*(result))  *# [1, 4, 9, 16, 25, 36, 49]*

mylist = [1, 2, 3, 4, 5, 6, 7]

result = filter(*lambda* *num*: num % 2 == 0, mylist)

print(*list*(result))  *# [2, 4, 6]*

## Сортировка

|  |  |
| --- | --- |
| **sorted(iterable)** | Создает новый отсортированный **список** из итератора.  За счет создания нового списка работает чуть медленнее, чем list.sort |
| **list.sort()** | Сортирует список на месте  Существует только у списков |

Python использует алгоритм **Timsort** — гибридный, стабильный алгоритм, разработанный Тимом Питерсом в 2002 году.

Особенности Timsort:

* **Стабильный** – если два элемента равны, их относительный порядок сохраняется.
* **Адаптивный** – работает быстрее на частично отсортированных данных.
* Сложность:

Лучший случай: O(n)

Средний и худший: **O(n log n)**

**key**

С помощью параметра **key** можно задать ключ сортировки – функцию, которая возвращает значение, по которому будет осуществлена сортировка

Например, *сортировка по длине строки*:

words = ["python", "is", "awesome"]

sorted(words, *key*=len)  *# ['is', 'python', 'awesome']*

Или *сортировка словарей:*

users = [

    {"name": "Alice", "age": 25},

    {"name": "Bob", "age": 30}

]

*# сортирует по возрасту*

sorted(users, *key*=*lambda* *u*: u["age"])

**Сортировка по убыванию**

**reverse=True**

sorted([3, 1, 4], *reverse*=True)  *# [4, 3, 1]*

**Сортировка по нескольким ключам:**

Если надо отсортировать по нескольким параметрам, то из-за стабильности сортировки

1. Сначала сортируй по наименее важному критерию
2. Потом — по более важному
3. В конце — по самому важному

**Пример:**

В примере ниже надо отсортировать студентов по имени, а при равенстве имен по баллам

*# Надо отсортировать студентов по имени, а при равенстве имен - по баллам*

students = [

    ("Алиса", 85),

    ("Боб", 90),

    ("Алиса", 95),

    ("Боб", 85),

    ("Чарли", 90)

]

*# Сначала по имени*

students = sorted(students, *key*=*lambda* *x*: x[0])

*# Потом по баллам*

students = sorted(students, *key*=*lambda* *x*: x[1])

print(students)

*# [('Алиса', 85), ('Боб', 85), ('Боб', 90), ('Чарли', 90), ('Алиса', 95)]*

Если бы мы вначале отсортировали по имени, а потом по баллам, то это сломало бы сортировку по имени – студенты были бы отсортированы по баллам.

**Альтернативный способ – сортировка за один вызов**

Для этого надо в качестве ключа передать кортеж, где на первом месте будет стоять более важный параметр (ключ) сортировки, на втором и далее – менее важные по уменьшению важности:

sorted(students, *key*=*lambda* *x*: (x[0], x[1]))

**Как происходит сортировка кортежей, списков и т.д?**

Python использует лексикографическое сравнение для составных типов (любых упорядоченных последовательностей: строки, списки, кортежи)

Если я правильно понял, Python по очереди сравнивает элементы составных типов.

Сравниваем элементы по одному, начиная с начала.

1. Если A[i] < B[i], значит A меньше B A < B
2. Если элементы равны — переходим к следующему.
3. Если одна последовательность закончилась, а другая — нет:

Короткая меньше, чем длинная

Короткая < длинная (если все её элементы совпали с началом длинной).

[1, 2] < [1, 3]   *# True*

[1, 2, 1] < [1, 2, 2]  *# True*

[1, 3] < [2, 1]   *# True — сравнивается по первому элементу*

В Python нельзя сравнивать разные типы данных.

**min и max**

Помимо *sort* и *sorted* параметр **key** можно передать и в функции **min()** и **max()**

Найти самую длинную строку

lines = ["aaaaaaa", "bbbbb", "ccc"]

print(min(lines))  *# aaaaaaa - значение найдено лексографически*

print(min(lines, *key*=*lambda* *line*: len(line)))  *# ccc*

*# аналогично*

print(min(lines, *key*=len))  *# ccc*

# Декораторы

## Декораторы функций

**Декораторы** — это вызываемые объекты (чаще всего функции), которые принимают функцию или класс, оборачивают её (или его) в другую функцию или класс, и возвращают модифицированную версию, расширяя или изменяя её поведение без изменения исходного кода.

**Декораторы – это синтаксический сахар для замыкания** (в отношении функций)

Вот декоратор:

*# Декоратор принимает функцию f как аргумент*

*def* count(*fn*):

    total = 0

*def* decorated(\**args*, \*\**kwargs*):

*# Переменная total объявлена нелокальной для возможности ее изменения*

*# из внутренней функции*

*nonlocal* total

        total += 1

        return fn(\*args, \*\*kwargs), total

    return decorated

Можно вызвать его таким образом:

*def* hello(*name*):

    return *f*"Привет, {name}!"

decorated\_hello = count(hello)

print(decorated\_hello("Пользователь\_1"))  *# ('Привет, Пользователь\_1!', 1)*

print(decorated\_hello("Пользователь\_2"))  *# ('Привет, Пользователь\_1!', 1)*

А можно добиться аналогичного результата таким образом:

@count

*def* hello(*name*):

    return *f*"Привет, {name}!"

print(hello("Пользователь\_1"))  *# ('Привет, Пользователь\_1!', 1)*

print(hello("Пользователь\_2"))  *# ('Привет, Пользователь\_1!', 1)*

## Типизация декораторов функций

from typing import Callable, ParamSpec, TypeVar

*# ParamSpec — это переменная типа для сигнатуры функции: она «запоминает»*

*# все аргументы (позиционные и именованные) функции,*

*# чтобы их можно было точно передать дальше с сохранением типобезопасности.*

*# Основное применение - функции высшего порядка (в отм числе декораторы)*

Params = ParamSpec("Params")

*# TypeVar — это переменная типа, которая позволяет писать обобщённые (generic)*

*# функции и классы, сохраняя связь между входными и выходными типами.*

Return = TypeVar("Return")

*def* simple\_decorator(

*func*: Callable[Params, Return],

) -> Callable[Params, Return]:

    """Простой декоратор."""

*def* wrapper(\**args*: Params.args, \*\**kwargs*: Params.kwargs) -> Return:

        """Функция-обертка."""

        print("Текст до вызова функции func().")

        result = func(\*args, \*\*kwargs)

        print("Текст после вызова функции func().")

        return result

    return wrapper

*def* say\_hello\_1() -> None:

    """Функция-приветствие."""

    print("Привет!")

## Декораторы классов

**Пример синглтона**

@singleton

*class* DatabaseConnection:

*def* \_\_init\_\_(*self*):

        print("Создание подключения к БД")

*# Первый вызов создаёт экземпляр*

db1 = DatabaseConnection()  *# Создание подключения к БД*

*# Второй — берёт существующий*

db2 = DatabaseConnection()  *# Ничего не будет выведено*

print(db1 is db2)  *# True*

## Типизация декораторов класса

from typing import Callable, TypeVar

*# TypeVar — это переменная типа, которая позволяет писать обобщённые (generic)*

*# функции и классы, сохраняя связь между входными и выходными типами.*

Class = TypeVar("Class", *bound*=*type*)

*def* add\_attribute(

*attribute\_name*: *str*, *attribute\_value*: *str*

) -> Callable[[Class], Class]:

    """Декоратор класса, добавляющий указанный атрибут к классу."""

*def* wrapper(*cls*: Class) -> Class:

        """Функция-обертка."""

        setattr(*cls*, attribute\_name, attribute\_value)

        return *cls*

    return wrapper

@add\_attribute("species", "кошка")

*class* CatClass5:

    """Класс кот."""

*def* \_\_init\_\_(*self*, *color*: *str*) -> None:

        """Инициализация кота с цветом."""

*self*.color = color

*self*.type\_ = "cat"

print(CatClass5.species) *#кошка*

## Встроенные декораторы

Встроенных декораторов больше, ниже список наиболее часто используемых:

|  |  |
| --- | --- |
| **@property** | Превращает метод в свойство |
| **@staticmethod** | Метод, который не принимает *self* |
| **@classmethod** | Метод, который принимает *cls* |
| **@wraps(func)** | Используется при написании своих декораторов, чтобы сохранить имя, докстринг и сигнатуру исходной функции. |
| **@functools.lru\_cache** | Кэширование результатов функции |
| **@abstractmethod** | Задаёт абстрактный метод |
| **@contextmanager** | Создание контекстных менеджеров |
| **@dataclass** | Генерация \_\_init\_\_", "\_\_repr\_\_ и т.д. |

## lru\_cache кеширование

Допустим мы хотим использовать рекурсивную функцию для вычисления последовательности Фибоначчи:

from timeit import timeit

*def* fib(*n*):

    if n in (0, 1):

        return 1

    return fib(n - 1) + fib(n - 2)

print(

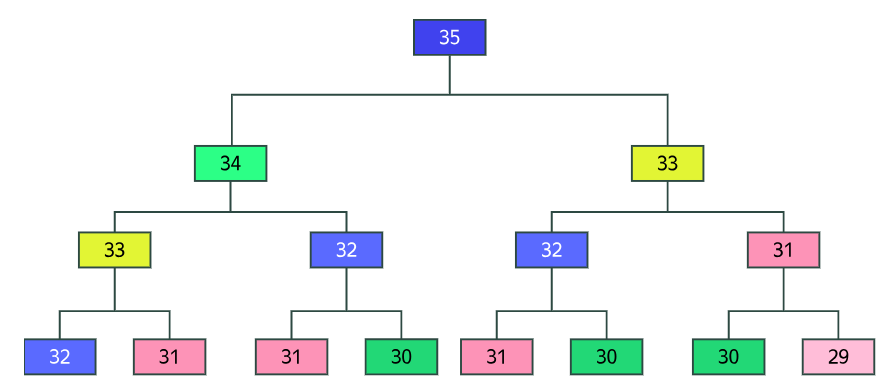
*f*"Среднее время вычисления: "

*f*"{round(timeit('fib(35)', *number*=10, *globals*=globals()) / 10, 3)} с."

)  *# около 1 секунды*

Среднее время выполнения будет 1-2 секунды.

При том, что при использовании итеративного подхода время выполнения сократиться до нескольких микросекунд. Огромная разница!

Причина в том, что при каждом вызове fib(n) функция вызывает саму себя дважды: один раз для fib(n - 1) и один раз для fib(n - 2). Это приводит к тому, что одни и те же значения пересчитываются много раз. Получается ветвящееся рекурсивное дерево, в котором одни и те же подзадачи повторяются снова и снова:

Для вычисления *35-го* члена последовательности потребуется *29860703* вызовов функции!

Это поведение можно существенно ускорить, если добавить кэширование в функцию.

**Но!** Если даже с кэшем попытаться посчитать 1000-ый член последовательности Фибоначчи, мы получим ошибку:

*RecursionError*: maximum recursion depth exceeded

Потому что Python по умолчанию **глубина рекурсии ограничена (обычно до 1000)**. Чтобы увеличить лимит, используйте *setrecursionlimit* из модуля *sys*:

from sys import setrecursionlimit

from timeit import timeit

*def* fib(*n*):

    if n not in cache:

        cache[n] = fib(n - 1) + fib(n - 2)

    return cache[n]

setrecursionlimit(2000)

cache = {0: 1, 1: 1}

print(

*f*"Среднее время вычисления: "

*f*"{round(timeit('fib(1000)', *number*=10, *globals*=globals()) / 10, 6)} с."

)   *# несколько микросекунд*

**Важно:** максимально допустимая глубина рекурсии зависит от операционной системы. Увеличивать её можно, но не бесконечно.

Код стал гораздо сложнее и менее читаемым.

Чтобы не писать дополнительный код вручную, можно поручить кеширование самому интерпретатору. В стандартной библиотеке Python для этого предусмотрен удобный инструмент — декоратор **lru\_cache из модуля functools**.

from functools import lru\_cache

from timeit import timeit

@lru\_cache(*maxsize*=1000)

*def* fib(*n*):

    if n in (0, 1):

        return 1

    return fib(n - 1) + fib(n - 2)

print(

*f*"Среднее время вычисления: "

*f*"{round(timeit('fib(35)', *number*=10, *globals*=globals()) / 10, 6)} с."

)  *# 2e-06 с*

**cache\_clear** очистка кэша:

fib.cache\_clear() *# очистить кэш*

**cache\_info** информация о кэше

print(fib.cache\_info())

*# CacheInfo(hits=42, misses=36, maxsize=1000, currsize=36)*

# Генераторы

## Генераторы и итераторы

**Генератор** — это **функция** (или выражение), которая возвращает **специальный итератор (генераторный итератор)**, используя ключевое слово *yield*.

**Итератор** — это объект, который реализует метод **\_\_next\_\_()**, возвращающий следующий элемент последовательности (или вызывает *StopIteration*, если элементы закончились).

Итератор не хранит всю последовательность (в отличие от массива, например), а хранит информацию о том, как получить следующее значение

Этот код:

*def* create\_cubes(*n*):

    result = []

    for x in range(n):

        result.append(x\*\*3)

    return result

print(create\_cubes(10))

for x in create\_cubes(10):

    print(x)

Можно заменить этим кодом, если у нас нет потребности иметь весь массив одновременно, а надо только перебрать элементы массива:

*def* create\_cubes(*n*):

    for x in range(n):

        yield x\*\*3

print(create\_cubes(10))  *# <generator object create\_cubes at 0x000001FE0FF80380>*

print(*list*(create\_cubes(10)))  *# [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]*

for x in create\_cubes(10):

    print(x)  *# 0, 1, 8, 27, 64, 125, 216, 343, 512, 729*

*Это позволяет существенно экономить память!*

Пример: последовательность Фибоначчи

*def* generate\_fibonacci(*n*):

    a = 1

    b = 1

    for i in range(n):

        yield a

*# Это синтаксис кортежей: создаем кортеж (b, a+b), а затем распаковываем его в переменные a, b*

        a, b = b, a + b

for x in generate\_fibonacci(10):

    print(x)  *# 1, 1, 2, 3, 5, 8, 13, 21, 34, 55*

## next

Функция *next* возвращает следующий элемент итератора.

«Под капотом» вызывается метод \_\_next\_\_() у итератора.

Если элементов больше нет, то можно указать значение по умолчанию, либо будет выброшено исключение *StopIteration*.

*def* simple\_gen():

    for x in range(3):

        yield x

print(*list*(simple\_gen()))  *# [0, 1, 2]*

generator = simple\_gen()

print(next(generator))  *# 0*

print(next(generator))  *# 1*

print(next(generator))  *# 2*

print(next(generator, 333))  *# 333*

print(next(generator))  *# Ошибка StopIteration*

## iter

**iter** – возвращает *итератор* из объекта (например, списка, словаря, генератора)

string = "hello"

next(string)  *# TypeError: 'str' object is not an iterator*

string\_iterator = iter(string)

print(next(string\_iterator))  *# h*

## Отличия: Итераторы, генераторы и итерируемые объекты

**Итерируемый объект ITERABLE**

Например, список, строка, множество и пр.

Это объект, у которого есть метод **\_\_iter\_\_()**, благодаря чем мы можем проитерироваться по нему в цикле **for**

*Итерируемый объект* можно передать в **iter( iterable )** и получить *итератор*

**Итератор ITERATOR**

Это объект, у которого есть методы **\_\_iter\_\_()** и **\_\_next\_\_()**

Это объект можно использовать вместе с методом **next(itarator).**

После исчерпания итератора*, next()* пробрасывает исключение *StopIteration*

Итератор ***одноразовый***: после исчерпания надо создавать новый

**Генератор GENERATOR**

**Итератор**, созданный функцией с yield или генераторным выражением

Генератор часто используются вместе с next(generator)

Генератор ***одноразовый***: после исчерпания надо создавать новый

Имеет дополнительные методы:

**.send(value)** – передача значение в генератор

**.close()** – закрытие генератора

**.throw()** – бросает исключение в генератор

# Импорты / экспорты

## Импорты

import math

или

from math import sqrt

**Что происходит под капотом**

1. **Поиск модуля**

Python ищет модуль в следующих местах (в порядке приоритета):

|  |  |
| --- | --- |
| Текущая директория | Где запущен скрипт |
| Содержимое переменной окружения *PYTHONPATH* | Аналогично *PATH*, но для Python |
| Стандартные библиотеки Python |  |
| Установленные пакеты *(site-packages)* | Каталог с установленными через pip библиотеками |

1. **Загрузка модуля**

Загружает его (читает файл *.py*)

Компилирует в байт-код (*.pyc*), если возможно

Выполняет весь код модуля (например, объявления функций, классов, переменных)

1. **Кэширование**

После первого импорта модуль сохраняется в словаре *sys.modules*. Это значит, что повторный импорт того же модуля не приведёт к повторному выполнению кода.

**Модуль** — это просто файл с расширением .py , который содержит: переменные, функции, классы, инструкции на уровне модуля (например, print())

**Виды импортов**

|  |  |
| --- | --- |
| **Полный импорт** | import math |
| **Частичный импорт** | from math import sqrt, pi |
| **Импорт с псевдонимом** | import numpy as np  from collections import defaultdict as dd |
| **Импорт всех имён**  Не рекомендуется, т.к. Добавляет всё в текущее пространство имён → может вызвать конфликты имён.  Кроме тех, которые начинаются с *нижнего подчеркивания \_* | from math import \* |

## Экспорты

В Python *все* имена (функции, классы, переменные), определённые в модуле, по умолчанию *доступны для импорта*, если только ты их не скрываешь намеренно.

Но есть способ явно указать, что должно быть доступно при импорте— через специальную переменную **\_\_all\_\_**

Имена объектов, которые не предназначены для экспорта, начинают *с нижнего подчеркивания \_*

*# mymodule.py*

*def* public\_func():

    print("I'm public")

*def* \_private\_func():

    print("I'm private")

*class* PublicClass:

    pass

PI = 3.14

\_\_all\_\_ = ['public\_func', 'PublicClass', 'PI']

from mymodule import \*

print(PI)  *# 3.14*

public\_func()  *# I'm public*

\_private\_func()  *# NameError: name '\_private\_func' is not defined*

* без \_\_all\_\_ при массовом импорте будут импортированы все имена, которые не начинаются с \_ (т.е. кроме приватных)
* с \_\_all\_\_ будут импортированы только те имена, которые указаны в \_\_all\_\_

## Перезагрузка модуля reload

**Перезагрузка модуля** — это процесс повторного импорта уже загруженного модуля, чтобы применить изменения в его коде без перезапуска интерпретатора. Это может быть полезно при разработке и отладке.

Когда вы первый раз импортируете модуль, Python:

1. Находит файл модуля.
2. Выполняет его код.
3. Кэширует модуль в sys.modules.

Повторные импорты не выполняют код модуля снова — используется закэшированная версия.

Чтобы обновить модуль, нужно использовать функцию **importlib.reload()**

**Ограничения и важные моменты**

1. Если модуль содержит вложенные импорты, то они тоже должны быть перезагружены отдельно, если нужно.
2. Изменения в определении классов могут не повлиять на уже созданные экземпляры.
3. Состояние объектов, созданных до перезагрузки, может остаться прежним.
4. Сами импортированные объекты останутся прежними – их надо импортировать повторно

from mymodule import MyClass

obj1 = MyClass()

obj1.greet()  *# Здравствуйте!*

import importlib

import mymodule

input()

importlib.reload(mymodule)  *# Перезагружаем модуль*

obj2 = MyClass()

obj2.greet()  *# Здравствуйте!*

from mymodule import MyClass

obj2 = MyClass()

obj2.greet()  *# Привет!*

Или использовать импорт через точку

import mymodule

obj1 = mymodule.MyClass()

obj1.greet()  *# Здравствуйте!*

import importlib

input()

importlib.reload(mymodule)  *# Перезагружаем модуль*

obj2 = mymodule.MyClass()

obj2.greet()  *# Привет!*

## Пакеты и подмодули

**Пакет** — это каталог с файлом *\_\_init\_\_.py* (может быть пустым).

**Пакет (package)** — это способ организации модулей, который позволяет группировать связанные модули в иерархическую структуру каталогов. Пакеты позволяют избежать конфликтов имён и упрощают управление большими проектами.

**Основные особенности пакета:**

1. Пакет — это каталог , содержащий один или несколько модулей Python.
2. В этом каталоге обязательно должен быть файл \_\_init\_\_.py (раньше он был обязательным для интерпретатора, чтобы отличать пакеты от обычных папок; в Python 3.3+ его наличие необязательно, но часто используется).
3. Файл \_\_init\_\_.py может быть пустым или содержать код, который будет выполняться при импорте пакета (например, настройка переменных, подключение библиотек и т.д.).

Пример структуры:

mypackage/

├── \_\_init\_\_.py

├── module1.py

└── module2.py

*# mymodule.py*

*def* greet():

    print("Hello from mymodule!")

import mypackage.module1

mypackage.module1.greet()

или

from mypackage.module1 import greet

greet()

или

from mypackage.module1 import \*

greet()

## \_\_name\_\_ и \_\_main\_\_

**\_\_name\_\_** и **\_\_main\_\_** — это важные понятия в Python, которые помогают определить, как запущен скрипт:

* как основная программа
* или как импортированный модуль

Это особенно важно на уровне сеньора, когда ты пишешь библиотеки, CLI-утилиты, приложения с точкой входа, или организуешь тестирование.

При импорте кода из какого-нибудь модуля, весь код из этого модуля будет выполнен.

**\_\_name\_\_** — это встроенная переменная, которая указывает имя текущего модуля. Она содержит значение **\_\_main\_\_,** если файл запущен из командной строки напрямую, или имя файла (например, my\_script), если код модуля вызван во время импорта

**one.py** запускаем командой *python one.py*

*def* func():

    print("Функция func из файла one.py")

print("Значение \_\_name\_\_ в файле one.py равно", \_\_name\_\_)

if \_\_name\_\_ == "\_\_main\_\_":

    print("one.py запускается напрямую")

else:

    print("one.py бл импортирован")

*# Значение \_\_name\_\_ в файле one.py равно \_\_main\_\_*

*# one.py запускается напрямую*

**two.py** запускаем командой *python two.py*

import one

print("Верхний уровень внутри two.py")

one.func()

if \_\_name\_\_ == "\_\_main\_\_":

    print("two.py запускается напрямую")

else:

    print("two.py бл импортирован")

*# one.py был импортирован*

*# Верхний уровень внутри two.py*

*# Функция func из файла one.py*

*# two.py запускается напрямую*

Это может быть полезным:

1. Точка входа для программы

Позволяет писать код, который будет выполняться только при прямом запуске файла:

if \_\_name\_\_ == "\_\_main\_\_":

    run\_application()

1. Предотвращение побочных эффектов при импорте

if \_\_name\_\_ == "\_\_main\_\_":

    heavy\_computation()

1. Написании CLI-инструментов
2. Юнит-тесты внутри модулей

# Область видимости Scope. LEGB

**Область видимости Scope** – та часть кода, в которой переменная видна и может быть использована

**Правило LEGB:**

**L – Local** – переменные, созданные любым способом внутри функции

**E – Enclosing functions locals** – переменные, которые являются локальными в любой из родительских функций, внутри которых мы сейчас находимся, от ближних по уровню вложенности к дальним

**G – Global** (глобально на уровне модуля) – переменные, объявленные на верхнем уровне модуля или с помощью *специального* синтаксиса внутри файла.

**B – Built-it** – встроенные методы и переменные: print, open, list, range и прочие

**Пример 1:**

*# global*

name = "Это глобальная строка"

*def* greet():

*# enclosing*

    name = "Иван"

*def* hello():

*# local*

        open = "Локальная строка"

        print(open)

    hello()

greet()

**Пример 2:**

x = 50

*def* func(*x*):

    print(*f*"X равно {x}")  *# 50*

    x = 200

    print(*f*"Я поменял значение X на {x}")  *# 200*

func(x)

print(*f*"Глобальный X все еще равен {x}")  *# 50*

## global и nonlocal

**global**

Ключевое слово **global** используется для объявления переменной, которая находится в глобальной области видимости, внутри функции, если вы хотите изменить её значение.

* Позволяет читать и изменять переменные из глобальной области (*вне всех функций*).
* Полезно, если ты хочешь повлиять на переменную, объявленную на уровне модуля или скрипта.

Лучше не использовать

count = 0  *# Глобальная переменная*

*def* increment():

*global* count

    count += 1

increment()

print(count)  *# 1*

**nonlocal**

* Позволяет читать и изменять переменные из внешней функции, *но не из глобальной области*.
* Используется только внутри вложенных функций (замыканий).

Можно использовать

*def* outer():

    count = 0  *# Переменная во внешней функции*

*def* inner():

*nonlocal* count

        count += 1

    inner()

    print(count)  *# 1*

outer()

Пример без **nonlocal**

*def* outer():

    x = 10

*def* inner():

        x = 20  *# Создаёт локальную переменную, а не меняет внешнюю*

    inner()

    print(x)  *# Всё ещё 10*

**globals()**

Возвращает список всех глобальных переменных, доступных в текущем модуле

print(globals())

*# {'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <\_frozen\_importlib\_external.SourceFileLoader object at 0x0000020A276880B0>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>, '\_\_file\_\_': 'C:\\materials\\python\\test.py', '\_\_cached\_\_': None}*

# Объектно-Ориентированное Программирование ООП

## Классы

**\_\_itni\_\_** – это конструктор класса – это основной метод, который должен быть определен в классе (аналог constructor)

*class* Dog:

*class* Dog:

*# Аттрибуты класса, одинаковые для всех инстансов этого класса*

    species = "mammal"

*def* \_\_init\_\_(*self*, *breed*, *name*):

        print("Instance of Dog was created")

*self*.breed = breed

*self*.name = name

*def* bark(*self*, *number*):

        print("{woof}My name is {name}.".format(*name*=*self*.name, *woof*="WOOF! " \* number))

*def* \_\_str\_\_(*self*):

        return *f*"""

breed: {*self*.breed}

name: {*self*.name}

species: {*self*.species}

"""

*# аналогчино Dog(breed="Labrador",name="Sammy")*

my\_dog = Dog("Labrador", "Sammy")

print((my\_dog))  *# <\_\_main\_\_.Dog object at 0x00000225A9B407D0>*

print(*type*(my\_dog))  *# <class '\_\_main\_\_.Dog'>*

print(my\_dog.breed)  *# Labrador*

print(my\_dog.name)  *# Sammy*

print(my\_dog.species)  *# mammal*

my\_dog.bark(3)  *# WOOF! WOOF! WOOF! My name is Sammy.*

## Атрибуты класса

Атрибут класса – свойство, одинаковое для всех экземпляров класса

В примере выше это *species*.

Атрибуты доступны как на экземпляре класса, так и на самом классе:

*class* Circle:

    pi = 3.14

*def* \_\_init\_\_(*self*, *radius*=1):

*self*.radius = radius

*# self.area = (radius\*\*2) \* self.pi*

*self*.area = (radius\*\*2) \* Circle.pi

my\_circle = Circle(30)

print(my\_circle.pi)  *# 3.14*

print(Circle.pi)  *# 3.14*

## Собственные атрибуты

Если *атрибуты класса* создаются *в теле класса*, то *собственные атрибуты* объекта создаются в методе *\_\_init\_\_ и других методах* через *self*

*class* Circle:

    pi = 3.14

*def* \_\_init\_\_(*self*, *radius*):

*self*.radius = radius

*def* make\_blue(*self*):

*self*.color = "blue"

*def* make\_green(*self*):

*self*.color = "green"

circle = Circle(33)

print(circle.\_\_dict\_\_)  *# {'radius': 33}*

circle.make\_blue()

print(circle.\_\_dict\_\_)  *# {'radius': 33, 'color': 'blue'}*

print(Circle.\_\_dict\_\_)  *# Объект с множеством свойств, в том числе*

*# {*

*#     ...*

*#     'pi': 3.14,*

*#     'make\_blue': <function Circle.make\_blue at 0x0000027D2CD59080>,*

*#     'make\_green': <function Circle.make\_green at 0x0000027D2CD5AAC0>,*

*#     ...*

*# }*

**Важно!**

**И атрибуты, и методы хранятся в \_\_dict\_\_**

Собственные методы и атрибуты хранятся в *\_\_dict\_\_* объекта. Атрибуты и методы класса в *\_\_dict\_\_* класса

## Цепочка атрибутов

**Где хранится значение атрибута класса?** В самом классе или в экземпляре?

**И атрибуты, и методы хранятся в \_\_dict\_\_**

*Собственные* методы и атрибуты хранятся в *\_\_dict\_\_* объекта. Атрибуты и методы *класса* в *\_\_dict\_\_* класса

Ссылка на класс, который надо проверять дальше в цепочке атрибутов, хранится в свойстве объекта **\_\_class\_\_**

Python **ищет атрибут** по **цепочке атрибутов** в таком порядке:

1. Сначала проверяет, есть ли атрибут у самого экземпляра (my\_dict.\_\_dict\_\_)
2. Если нет — ищет в классе (MyClass.\_\_dict\_\_)
3. Если и там нет — идёт по цепочке наследования

*class* Circle:

    pi = 3.14

*def* \_\_init\_\_(*self*, *radius*=1):

*self*.radius = radius

*# self.area = (radius\*\*2) \* self.pi*

*self*.area = (radius\*\*2) \* Circle.pi

my\_circle = Circle(30)

print(my\_circle.pi)  *# 3.14*

print(Circle.pi)  *# 3.14*

*# оператор is проверяет, ссылаются ли два значения на одну и ту же область памяти*

print(my\_circle.pi is Circle.pi)  *# True*

my\_circle.pi = 4

print(my\_circle.pi)  *# 4*

print(Circle.pi)  *# 3.14*

my\_circle.pi = 3.14

print(my\_circle.pi)  *# 3.14*

print(Circle.pi)  *# 3.14*

*# Из-за того, что Питон кэширует числа от -5 до 256, они снова ссылаются на одну и ту же ячейку памяти*

print(my\_circle.pi is Circle.pi)  *# False*

Circle.pi = 4

my\_circle2 = Circle(3)

*# К этому моменту у my\_circle есть собственный атрибут pi*

print(my\_circle.pi)  *# 3.14*

print(Circle.pi)  *# 4*

*# а my\_circle2 берет его из цепочки атрибутов*

print(my\_circle2.pi)  *# 4*

*# Т.к. у my\_circle есть собственное свойство pi, у my\_circle2 pi доступно только по цепочке атрибутов из класса*

for key in my\_circle.\_\_dict\_\_:

    print(key)  *# pi radius area*

for key in my\_circle2.\_\_dict\_\_:

    print(key)  *# radius area*

## Наследование

*class* Animal:

*def* \_\_init\_\_(*self*):

        print("Animal created")

*def* who\_am\_i(*self*):

        print("I am an animal")

*def* eat(*self*):

        print("I am eating")

my\_animal = Animal()  *# Animal created*

my\_animal.eat()  *# I am eating*

my\_animal.who\_am\_i()  *# I am an animal*

*class* Dog(*Animal*):

*def* \_\_init\_\_(*self*):

        Animal.\_\_init\_\_(*self*)

        print("Dog created")

*def* who\_am\_i(*self*):

        print("I am a dog")

*def* bark(*self*):

        print("WOOF!")

my\_dog = Dog()  *# Animal created,Dog created*

my\_dog.who\_am\_i()  *# I am a dog*

Более современный и «правильный вариант» с использованием **super (использовать аккуратно. Возможно, лучше использовать первый способ выше)**

*class* Player:

*def* \_\_init\_\_(*self*, *name*: *str*):

*self*.name = name

*self*.balance = 100

*class* Dealer(*Player*):

*def* \_\_init\_\_(*self*):

*super*().\_\_init\_\_("Дилер")

## Множественное наследование

*class* GreetingFormal:

*def* \_\_init\_\_(*self*):

*self*.formal\_greeting = "Добрый день,"

*def* greet\_formal(*self*, *name*):

        return *f*"{*self*.formal\_greeting} {name}!"

*class* GreetingInformal:

*def* \_\_init\_\_(*self*):

*self*.informal\_greeting = "Привет,"

*def* greet\_informal(*self*, *name*):

        return *f*"{*self*.informal\_greeting} {name}!"

*class* GreetingMix(*GreetingFormal*, *GreetingInformal*):

*def* \_\_init\_\_(*self*):

        GreetingFormal.\_\_init\_\_(*self*)

        GreetingInformal.\_\_init\_\_(*self*)

mixed\_greeting = GreetingMix()

print(mixed\_greeting.greet\_formal("Пользователь"))

print(mixed\_greeting.greet\_informal("Пользователь"))

При множественном наследовании есть способ организовать всё через вызовы super, но лучше явно указывать класса, которые мы вызываем.

## MRO Method Resolution Order

**MRO (Method Resolution Order)** — это порядок разрешения методов в Python, который определяет, в каком порядке Python будет искать атрибуты и методы при обращении к ним через объект.

**MRO** — это заранее вычисленный список классов, определяющий порядок поиска методов и атрибутов при обращении к ним через объект. Python использует этот список для разрешения любых неоднозначностей в наследовании.

*class* A:

    pass

*class* B(*A*):

    pass

*class* C(*A*):

    pass

*class* D(*B*, *C*):

    pass

obj = D()

*# Тип объекта:*

print(*type*(obj))  *# <class '\_\_main\_\_.D'>*

print(obj.\_\_class\_\_)  *# <class '\_\_main\_\_.D'>*

*# MRO определяется классом объекта:*

print(D.\_\_mro\_\_)

*# (<class '\_\_main\_\_.D'>,*

*#  <class '\_\_main\_\_.B'>,*

*#  <class '\_\_main\_\_.C'>,*

*#  <class '\_\_main\_\_.A'>,*

*#  <class 'object'>)*

|  |  |  |
| --- | --- | --- |
| **\_\_mro\_\_** | Возвращает кортеж классов | (<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>) |
| **mro()** | Возвращает список классов | [<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>] |
| **\_\_bases\_\_** | Возвращает кортед прямых родителей | (<class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>) |

## Полиморфизм

Полиморфизм — это один из ключевых принципов объектно-ориентированного программирования (ООП), который позволяет объектам разных классов обрабатываться как объекты одного и того же базового класса.

*class* Dog:

*def* \_\_init\_\_(*self*, *name*):

*self*.name = name

*def* speak(*self*):

        return *self*.name + " says woof!"

*class* Cat:

*def* \_\_init\_\_(*self*, *name*):

*self*.name = name

*def* speak(*self*):

        return *self*.name + " says meow!"

niko = Dog("Noko")

felix = Cat("Felix")

niko.speak()  *# Noko says woof!*

felix.speak()  *# Felix says meow!*

for pet in [niko, felix]:

    print(*type*(pet))

    print(pet.speak())

*def* pet\_speak(*pet*):

    print(pet.speak())

pet\_speak(niko)

pet\_speak(felix)

**Примеры полиморфизма в Питон:**

1. **Перегрузка методов** (в ограниченном виде)
2. **Переопределение методов** в наследниках
3. **Абстрактные базовые классы**
4. Единый интерфейс для разных типов данных
5. Разные имплементации одного метода в разных классах (экземпляры разных классов можно перебирать в for, передавать в функцию как в примере выше)

**Переопределение методов**

*class* Animal:

*def* speak(*self*):

        print("Animal speaks")

*class* Dog(*Animal*):

*def* speak(*self*):

        print("Woof!")

*class* Cat(*Animal*):

*def* speak(*self*):

        print("Meow!")

## Абстрактные классы

**Абстрактный класс** — это класс, который не может напрямую создавать экземпляры (нельзя вызвать MyClass()) и содержит один или несколько абстрактных методов .

**Основные особенности:**

* Содержит абстрактные методы, которые не имеют реализации.
* Подклассы обязаны реализовать все абстрактные методы.
* Предназначен для наследования и задания интерфейса.
* Может содержать обычные методы с реализацией, которыми могут пользоваться дочерние классы

**Нативный способ**

*class* Animal:

*def* \_\_init\_\_(*self*, *name*):

*self*.name = name

*def* speak(*self*):

        raise *NotImplementedError*("Subclass must implemented this abstract method")

*def* greeting(*self*):

        print(*f*"Hello! I am {*self*.name}")

*class* Dog(*Animal*):

*def* \_\_init\_\_(*self*, *name*):

        Animal.\_\_init\_\_(*self*, name)

*def* speak(*self*):

        print("Woof!")

*class* Cat(*Animal*):

*def* \_\_init\_\_(*self*, *name*):

        Animal.\_\_init\_\_(*self*, name)

dog = Dog("Niko")

cat = Cat("Felix")

dog.speak()  *# Woof!*

cat.greeting()  *# Hello! I am Felix*

cat.speak()  *# NotImplementedError: Subclass must implemented this abstract method*

**Модуль abc (Abstract Base Classes)**

from abc import ABC, abstractmethod

*class* Animal(ABC):

*def* \_\_init\_\_(*self*, *name*):

*self*.name = name

    @abstractmethod

*def* speak(*self*):

        pass

*def* greeting(*self*):

        print(*f*"Hello! I am {*self*.name}")

*class* Dog(*Animal*):

*def* \_\_init\_\_(*self*, *name*):

        Animal.\_\_init\_\_(*self*, name)

*def* speak(*self*):

        print("Woof!")

*class* Cat(*Animal*):

*def* \_\_init\_\_(*self*, *name*):

        Animal.\_\_init\_\_(*self*, name)

dog = Dog("Niko")

cat = Cat(

    "Felix"

)  *# TypeError: Can't instantiate abstract class Cat without an implementation for abstract method 'speak'*

## Магические методы классов

**Магические методы** (часто называются также **специальными методами**) — это методы, имена которых окружены двойными подчёркиваниями, например: *\_\_init\_\_, \_\_str\_\_, \_\_add\_\_*.

Такие методы вызываются автоматически при выполнении встроенных операций и функций и позволяют настроить поведение объектов под нужды вашей программы.

Этот механизм называется **перегрузкой операторов operator overloading**. Вы можете определить, как именно ваш объект будет вести себя при сложении, сравнении, индексировании, отображении и других операциях.

Вот несколько примеров часто используемых магических методов:

* **\_\_repr\_\_** — вызывается функцией repr() и возвращает строку, представляющую объект в виде, пригодном для отладки и воссоздания. Также используется при отображении объектов в коллекциях
* **\_\_str\_\_** — вызывается функцией str() и при печати объектов. Должен возвращать удобочитаемое строковое представление
* **\_\_add\_\_** — позволяет задать поведение для оператора + (если объект стоит слева, например, *obj + 1*)
* **\_\_eq\_\_**— определяет, как объекты сравниваются на равенство (==)
* **\_\_len\_\_** — используется функцией len() для получения длины объекта

**Разница** между **\_\_str\_\_** и **\_\_repr\_\_** в том, что:

* **\_\_str\_\_** возращает представление объекта для пользователя. Например, в функциях *print()* и *str()*
* **\_\_repr\_\_** — «точное» строковое представление для разработчика.

Вызывается при *repr(obj)*, в интерактивной оболочке, в логах**, при выводе в списках** (и других коллекциях) и т.д.

Должна быть **однозначной** и, по возможности, позволять воссоздать объект.

**Идеально**: **eval(repr(obj)) == obj** (не всегда достижимо, но это цель).

*class* Person:

*def* \_\_init\_\_(*self*, *name*, *age*):

*self*.name = name

*self*.age = age

*def* \_\_str\_\_(*self*):

        return *f*"Человек по имени {*self*.name}, возраст {*self*.age}"

*def* \_\_repr\_\_(*self*):

        return *f*"Person(name={*self*.name*!r*}, age={*self*.age*!r*})"

alice = Person("Alice", 38)

print(alice)

*# человек по имени Alice, возраст 38*

persons\_list = [alice]

*# print(repr(p))*

print(persons\_list)

*# [Person(name='Alice', age=38)]*

### Список магических методов:

|  |  |  |
| --- | --- | --- |
| **Методы для операций сравнения** | | |
| \_\_lt\_\_(self, other) | < | |
| \_\_le\_\_(self, other) | <= | |
| \_\_eq\_\_(self, other) | == | |
| \_\_ne\_\_(self, other) | != | |
| \_\_gt\_\_(self, other) | > | |
| \_\_ge\_\_(self, other) | >= | |
|  | | |
| **Методы для работы с объектом как с коллекцией** | | |
| \_\_getitem\_\_(self, key) | obj[key] | |
| \_\_setitem\_\_(self, key, value) | obj[key] = value | |
| \_\_delitem\_\_(self, key) | del obj[key] | |
| \_\_len\_\_(self) | len(obj) | |
| \_\_contains\_\_(self, item) | item in obj | |
|  | | |
| **Математические операции**  В первом столбце метод, если объект стоит **слева**, например, *obj + 1*  Во втором столбце метод, если объект стоит **справа**, например, *1 + obj* | | |
| \_\_add\_\_(self, other) | \_\_radd\_\_ | + |
| \_\_sub\_\_(self, other) | \_\_rsub\_\_ | - |
| \_\_mul\_\_(self, other) | \_\_rmul\_\_ | \* |
| \_\_matmul\_\_(self, other) | \_\_rmatmul\_\_ | @ матричное умножение |
| \_\_truediv\_\_(self, other) | \_\_rtruediv\_\_ | / |
| \_\_floordiv\_\_(self, other) | \_\_rfloordiv\_\_ | // |
| \_\_mod\_\_(self, other) | \_\_rmod\_\_ | % |
| \_\_divmod\_\_(self, other) | \_\_rdivmod\_\_ | divmod(self, other) |
| \_\_pow\_\_(self, other) | \_\_rpow\_\_ | \*\* |
| Побитовые операторы | | |
| \_\_lshift\_\_(self, other) | \_\_rlshift\_\_ | << |
| \_\_rshift\_\_(self, other) | \_\_rrshift\_\_ | >> |
| \_\_and\_\_(self, other) | \_\_rand\_\_ | & |
| \_\_xor\_\_(self, other) | \_\_rxor\_\_ | ^ |
| \_\_or\_\_(self, other) | \_\_ror\_\_ | | |
|  | | |
| **Методы in-place**  Изменяют объект «на месте», а не создают новый объект | | |
| \_\_iadd\_\_(self, other) | += | |
| \_\_isub\_\_(self, other) | -= | |
| \_\_imul\_\_(self, other) | \*= | |
| \_\_imatmul\_\_(self, other) | @= Матричное умножение с присваиванием | |
| \_\_itruediv\_\_(self, other) | /= | |
| \_\_ifloordiv\_\_(self, other) | //= | |
| \_\_imod\_\_(self, other) | %= | |
| \_\_ipow\_\_(self, other) | \*\*= | |
| Побитовые операторы | | |
| \_\_ilshift\_\_(self, other) | <<= Побитовый сдвиг влево с присваиванием | |
| \_\_irshift\_\_(self, other) | >>= Побитовый сдвиг вправо с присваиванием | |
| \_\_iand\_\_(self, other) | &= Побитовое И с присваиванием | |
| \_\_ixor\_\_(self, other) | ^= Побитовое XOR с присваиванием | |
| \_\_ior\_\_(self, other) | |= Побитовое ИЛИ с присваиванием | |
|  | | |
| **Метод вызова объекта как функции** | | |
| \_\_call\_\_(self, \*args, \*\*kwargs) | вызывается, когда объект вызывается как функция: obj(...) | |

### Примеры

*class* A:

*def* \_\_init\_\_(*self*):

*self*.value = 10

*def* \_\_add\_\_(*self*, *other*):

        print("Выполняется метод \_\_add\_\_.")

        return *self*.value \* 10 + other

*def* \_\_radd\_\_(*self*, *other*):

        print("Выполняется метод \_\_radd\_\_.")

        return *self*.value \* 100 + other

*def* \_\_iadd\_\_(*self*, *other*):

        print("Выполняется метод \_\_iadd\_\_.")

*self*.value += other

        return *self*

*def* \_\_str\_\_(*self*):

        return *f*"\_\_str\_\_: value: {*self*.value}."

a = A()

print(a + 1, *end*="\n\n")

*# Выполняется метод \_\_add\_\_.*

*# 101*

print(1 + a, *end*="\n\n")

*# Выполняется метод \_\_radd\_\_.*

*# 1001*

a += 1

print(a, *end*="\n\n")

*# Выполняется метод \_\_iadd\_\_.*

*# \_\_str\_\_: value: 11.*

**\_\_str\_\_**

Метод для строкового представления объекта

Автоматически вызывается функциями *str* и *print*, если передать ей объект в качестве аргумента

*class* Book:

*def* \_\_init\_\_(*self*, *title*, *author*, *pages*):

*self*.title = title

*self*.author = author

*self*.pages = pages

book = Book("Руководство по Python", "Иван Иванов", 333)

print(book)  *# <\_\_main\_\_.Book object at 0x0000017D904F0950>*

*class* Book:

*def* \_\_init\_\_(*self*, *title*, *author*, *pages*):

*self*.title = title

*self*.author = author

*self*.pages = pages

book = Book("Руководство по Python", "Иван Иванов", 333)

print(book)  *# Иван Иванов: Руководство по Python*

**\_\_len\_\_**

Определяет, как будет работать встроенная функция len() для экземпляра класса

Должен возвращать **неотрицательное целое число >= 0**

*class* Book:

*def* \_\_init\_\_(*self*, *title*, *author*, *pages*):

*self*.title = title

*self*.author = author

*self*.pages = pages

*def* \_\_len\_\_(*self*):

        return *self*.pages

book = Book("Руководство по Python", "Иван Иванов", 333)

print(len(book))  *# 333*

**\_\_del\_\_**

Дополнительные действия, которые будут выполнены при удалении объекта

*class* Book:

*def* \_\_init\_\_(*self*, *title*, *author*, *pages*):

*self*.title = title

*self*.author = author

*self*.pages = pages

*def* \_\_len\_\_(*self*):

        return *self*.pages

*def* \_\_del\_\_(*self*):

        print(*f*"Книга {*self*.title} автора {*self*.author} удалена")

book = Book("Руководство по Python", "Иван Иванов", 333)

print(len(book))  *# 333*

del book  *# Книга Руководство по Python автора Иван Иванов удалена*

print(book)  *# NameError: name 'book' is not defined.*

# Даты datetime

## Объект datetime

from datetime import datetime, timedelta

|  |  |  |
| --- | --- | --- |
| datetime.**now()** | Текущая дата и время | cur\_dt = datetime.now()  print(cur\_dt)  *# 2025-10-13 10:11:28.644413* |
| **Аттрибуты**  year  month  day  hour  minute  second  microsecond | Аттрибуты текущей даты | print(      cur\_dt.year,      cur\_dt.month,      cur\_dt.day,      cur\_dt.hour,      cur\_dt.minute,      cur\_dt.second,      cur\_dt.microsecond,  *end*=" ",  )  *# 2025 10 13 10 13 33 813549* |
| cur\_dt.**weekday()**  cur\_dt.**isoweekday()** | Номер дня недели:  начинается с 0  начинается с 1 | *# Для Понедельника*  print(cur\_dt.weekday())  *# 0*  print(cur\_dt.isoweekday())  *# 1* |
| cur\_dt.**tzinfo** | TimeZone  Информаиця о часовм поясе | *# Если часовой пояс не устанавливали, то будет None*  print(cur\_dt.tzinfo)  *# None* |
| cur\_dt.**timestamp()** | Временная метка  Число секунд, прошедших с 00:00:00 UTC 1 января 1970 года | cur\_dt = datetime.now()  print(cur\_dt.timestamp())  *# секунды + микросекунды как дробная часть*  *# 1760341842.155744* |
| datetime.**fromtimestamp(**  **timestamp)** | Создание даты из временной метки | print(datetime.fromtimestamp(  1760341842.155744  ))  *# 2025-10-13 10:50:42.155744* |
| datetime.**strptime(строка, шиблон)** | Создание даты из строки  (см. ниже информацию о шаблонах) | str\_to\_dt = "2007-12-02 12:30:45"  res\_dt = datetime.strptime(str\_to\_dt, "%Y-%m-%d %H:%M:%S")  print(res\_dt)  *# 2007-12-02 12:30:45* |
| datetime.**strftime(дата, шаблон)** | Перевод даты в строку  (см. ниже информацию о шаблонах) | current = datetime.now()  str\_from\_dt = datetime.strftime(current, "%A, %B %d, %Y")  print(str\_from\_dt)  *# Monday, October 13, 2025* |

## Создание объекта datetime

1. **Текущая дата datetime.now()**

cur\_dt = datetime.now()

print(cur\_dt)

*# 2025-10-13 10:11:28.644413*

1. **Определенная дата**

date\_t = datetime(2007, 11, 19, 12, 33, 13)

print(date\_t)

*# 2007-11-19 12:33:13*

1. **Из строки datetime.strptime()**

str\_to\_dt = "2007-12-02 12:30:45"

res\_dt = datetime.strptime(str\_to\_dt, "%Y-%m-%d %H:%M:%S")

print(res\_dt)

*# 2007-12-02 12:30:45*

## Форматирование дат

str\_to\_dt = "2007-12-02 12:30:45"

res\_dt = datetime.strptime(str\_to\_dt, "%Y-%m-%d %H:%M:%S")

print(res\_dt)

*# 2007-12-02 12:30:45*

Для формирования шаблнов строк в методах **.strptime()** и **.strftime()**:

|  |  |  |
| --- | --- | --- |
| Код | Описание | Пример |
| %a | Сокращенное название дня недели | Sun, Mon, … |
| `%A | Полное название дня недели | Sunday, Monday, … |
| %w | День недели как число, Вс - 0, Пн - 1, ... Сб - 6 | 0, 1, …, 6 |
| %d | День месяца в виде числа с нулями | 01, 02, …, 31 |
| %-d | День месяца в виде числа без нулей | 1, 2, …, 31 |
| %b | Сокращенное название месяца | Jan, Feb, …, Dec |
| %B | Полное название месяца | January, February, … |
| %m | Месяц в виде числа с нулями | 01, 02, …, 12 |
| %-m | Месяц в виде числа без нулей | 1, 2, …, 12 |
| %y | Год без века как число с нулями | 00, 01, …, 99 |
| %-y | Год без века как число без нулей | 0, 1, …, 99 |
| %Y | Год с веком | 1999, 2019, ... |
| %H | Час (в 24-часовом формате) в виде числа с нулями | 00, 01, …, 23 |
| %-H | Час (в 24-часовом формате) в виде числа без нулей | 0, 1, …, 23 |
| %I | Час (12-часовой формат) в виде числа с нулями | 01, 02, …, 12 |
| %-I | Час (12-часовой формат) в виде числа без нулей | 1, 2, …, 12 |
| %p | AM или PM | AM, PM |
| %M | Минуты в виде числа с нулями | 00, 01, …, 59 |
| %-M | Минуты в виде числа без нулей | 0, 1, …, 59 |
| %S | Секунды в виде числа с нулями | 00, 01, …, 59 |
| %-S | Секунды в виде числа без нулей | 0, 1, …, 59 |
| %j | День года в виде числа с нулями | 001, 002, …, 366 |
| %-j | День года в виде числа без нулей | 1, 2, …, 366 |
| %c | Полная дата и время | Sun Nov 21 10:38:12 2021 |
| %x | Дата | 11/21/21 |
| %X | Время | 10:43:51 |

## Промежуток времени timedelta, арифметика дат

Даты можно сравнивать:

print(datetime(2007, 12, 2) > datetime(2003, 11, 19))

*# True*

Если вычесть одну дату из другой, то получим промежуток дат объект класса **timedelta**

date1 = datetime(2007, 12, 2)

date2 = datetime(2002, 11, 19)

diff = date1 - date2

print(diff)  *# 1839 days, 0:00:00*

print(*type*(diff))  *# <class 'datetime.timedelta'>*

Объект **timedelta** можно также создать вручную:

from datetime import timedelta

delta = timedelta(*days*=100)

hundred\_days\_before = current - delta

print(current)  *# 2025-10-13 11:38:16.675710*

print(hundred\_days\_before)  *# 2025-07-05 11:38:16.675710*

Иногда бывает удобно воспользоваться циклом while, чтобы создать перечень дат:

from datetime import datetime, timedelta

cur\_date = datetime(2021, 1, 1)

end\_date = datetime(2021, 1, 5)

while cur\_date <= end\_date:

*# выведем cur\_date в формате "месяц число, год"*

    print(cur\_date.strftime("%b %d, %Y"))

*# прибавим к выводимой дате один день*

    cur\_date += timedelta(*days*=1)

*# Jan 01, 2021*

*# Jan 02, 2021*

*# Jan 03, 2021*

*# Jan 04, 2021*

*# Jan 05, 2021*

## Часовые пояса

Для изменения часового пояса нужно импортировать модуль pytz

import pytz

from datetime import datetime

*# для изменения часового пояса нужно импортировать модуль pytz*

import pytz

cur\_dt = datetime.now()

*# Если часовой пояс не устанавливали, то будет None*

print(cur\_dt.tzinfo)

*# None*

Установим часовой пояс:

*# выведем текущее время в Москве*

dt\_moscow = datetime.now(pytz.timezone("Europe/Moscow"))

print(dt\_moscow)  *# 2025-10-13 10:28:57.618230+03:00*

print(dt\_moscow.tzinfo)  *# Europe/Moscow*

# Обработка ошибок

## Подходы LBYL и EAFP

**Исключения** в Python — это события, которые происходят при нарушении корректного выполнения кода и останавливают его работу.

**LBYL (Look Before You Leap** – *«Cначала проверь, потом действуй»***)**

В программе, реализующей такой подход, проверяются возможные условия возникновения ошибок до исполнения основного кода.

*# Для каждого числа из введенного диапазона вывести их обратные значения*

*# (x -> 1/x)*

*# При этом большую часть программы занимает проверка корректности*

*# входных данных:*

*# - оба значения можно конвертировать в число*

*#   (в том числе с учетом отрицательных числа))*

*# - в диапазоне нет ноля 0*

start = input()

end = input()

if not (start.lstrip("-").isdigit() and end.lstrip("-").isdigit()):

    print("Необходимо ввести два числа.")

else:

    interval = range(*int*(start), *int*(end) + 1)

    if 0 in interval:

        print("Диапазон чисел содержит 0.")

    else:

        print(";".join(*str*(1 / x) for x in interval))

**EAFP (Easier to Ask Forgiveness than Permission** – *«Проще попросить прощения, чем разрешения»***)**

В этом подходе сначала исполняется код, а в случае возникновения ошибок происходит их обработка. Подход EAFP реализован в Python в виде механизма обработки исключений.

try:

    print(";".join(*str*(1 / x) for x in range(*int*(input()), *int*(input()) + 1)))

except *ZeroDivisionError*:

    print("Диапазон чисел содержит 0.")

except *ValueError*:

    print("Необходимо ввести два числа.")

|  |  |
| --- | --- |
| **LBYL (Look Before You Leap)** | **EAFP (Easier to Ask Forgiveness than Permission)** |
| Сначала проверяем, можно ли выполнить действие | Сначала пытаемся выполнить, а потом обрабатываем ошибки |
| Защитный, с большим числом условий | Оптимистичный, с обработкой ошибок постфактум |
| Много условий, код перегружен проверками | Возможны неожиданные ошибки, если не учесть исключения |
| Возможные ошибки малочисленны или не требуют сложной проверки | Ошибки возникают на разных уровнях вложенности кода, но приводят к равноценным вариантам обработки |

## try, except, else, finally

Фактически, блок else можно назвать избыточным, т.к. весь код для успешного сценария можно поместить в try (как в JavaScript). Но использование else считается хорошей практикой и улучшает структуру и читаемость кода.

try:

    print("Введите число")

    number2 = input()  *# number2 всегда будет строкой*

    result = number1 + number2

*# если в try возникла ошибка*

except:

    print("\nКажется, что мы складываем не два числа\n")

*# если код в try выполнен успешно*

else:

    print(*f*"\nСложение прошло успешно. Результат: {result}\n")

finally:

    print("Это просто блок finally, который выполнится в любом случае\n")

print("Это код после блока try/catch/else/finally")

*# Введите число*

*# Кажется, что мы складываем не два числа*

*# Это просто блок finally, который выполнится в любом случае*

*# Это код после блока try/catch/else/finally*

## Несколько except

try:

    file = open("testfile", "w")

    file.write("Записываем строку в файл")

except *TypeError*:

    print("Произошла ошибка TypeError")

except *ValueError*:

    print("Произошла ошибка ValueError")

except *OSError*:

    print("Произошла ошибка OSError")

except:

    print("Все другие ошибки")

finally:

    print("Это строка выполняется в любом случае")

Список встроенных ошибок Python <https://docs.python.org/3.15/library/exceptions.html>

Если первым стоит *except Exception*, то он перехватит практически любое исключение — почти все встроенные ошибки наследуются от Exception. В этом случае интерпретатор не дойдёт до остальных блоков, и они просто не выполнятся. Это означает, что более специфические обработчики окажутся бесполезными — их код никогда не сработает.

# Тестирование

## Линтеры

Тестирует стиль кода и форматирование

* pylint
* pyflakes
* pep8

black – это форматтер кода, который часто используется вместе с линтерами

**pylint**

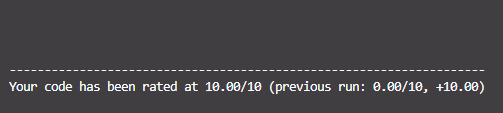
Запуск

**pylint test.py**

Запуск со статистикой

**pylint test.py --reports=yes**

В конце статистики можно увидеть общую оценку кода



## unittest

**unittest** – это встроенная библиотека Python для написания и запуска модульных тестов

**my\_script.py**

*def* cap\_text(*text*):

    """

    Input: строка

    Output: строка, в которой каждое слово написано с большой буквы

    """

*# return text.title() # Это корректный метод, который переводит первую букву каждого слова в верхний регистра*

    return text.capitalize() *# Переводит только первую букву первого слова в верхний регистр*

**test\_my\_script.py**

import unittest

import my\_script

*class* TestCap(*unittest*.*TestCase*):

*def* test\_one\_word(*self*):

        text = "python"

        result = my\_script.cap\_text(text)

*self*.assertEqual(result, "Python")

*def* test\_multiple\_words(*self*):

        text = "monty python"

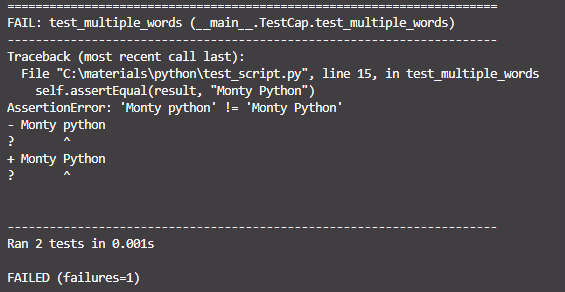
        result = my\_script.cap\_text(text)

*self*.assertEqual(result, "Monty Python")

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

Выполняем команду **python test\_script.py**



# PiPY и pip

**PyPI (Python Package Index)** — это официальный репозиторий пакетов для Python.

## pip

**pip** — это стандартный инструмент командной строки для установки пакетов из PyPI.

|  |  |
| --- | --- |
| **pip install package\_name** | Установить пакет |
| **pip install package\_name==1.0.0** | Установить конкретную версию |
| **pip install --upgrade package\_name** | Обновить пакет |
| **pip uninstall package\_name** | Удалить пакет |
| **pip list** | Посмотреть установленные пакеты в текущем окружении (виртуальном или глобальном) |
| **pip show package\_name** | Информация о конкретном пакете |
| **pip freeze > requirements.txt** | Сохранить список зависимостей |
| **pip install -r requirements.txt** | **Установить зависимости из файла** |
| **pip list –outdated**  **pip list --outdated --format=json** | Проверить устаревшие пакеты в текущем виртуальном окружении  В формате json |

## Виртуальное окружение

**Виртуальное окружение (virtual environment)** — это независимая копия Python и установленных пакетов. Оно позволяет:

* Устанавливать библиотеки изолированно от других проектов
* Избежать конфликтов версий
* Точнее воспроизводить зависимости (например, в CI/CD или production)

Виртуальные окружения в Python похожи на проекты с отдельным package.json в JavaScript (Node.js).

**Сходства**

|  |  |
| --- | --- |
| **Python** | **JavaScript (Node.js)** |
| Виртуальное окружение (**venv**) | Проект с собственным package.json |
| Установленные пакеты изолированы от других проектов | Установленные node\_modules относятся только к этому проекту |
| requirements.txt или pyproject.toml аналогичен package.json | package.json описывает зависимости и версии |
| pip install package → устанавливается в текущее окружение | npm install package → устанавливается в node\_modules проекта |

**Отличия**

|  |  |  |
| --- | --- | --- |
| Особенность | Python (venv + pip) | JavaScript (npm/yarn/pnpm) |
| Управление версиями зависимостей | Требуется ручное обновление requirements.txt | package.json автоматически сохраняет установленные пакеты |
| Локальная установка пакетов | По умолчанию глобальной установки нет | Есть глобальная (npm install -g) |
| Механизм изоляции | Отдельная копия Python и site-packages | node\_modules в каждом проекте |
| Поддержка lock-файлов | Нет встроенного, но можно использовать pip-tools или Poetry | Есть package-lock.json, yarn.lock, pnpm-lock.yaml |

**Создание окружения MacOS**

*# Создание окружения*

python -m venv venv

*# Активация для Mac. Активацию для Win смотри ниже*

source .venv/bin/activate

*# Установка зависимостей*

pip install flask requests

*# Сохраняем зависимости*

pip freeze > requirements.txt

*# Установить зависимости из файла (например, из requirements.txt)*

pip install -r requirements.txt

Note:

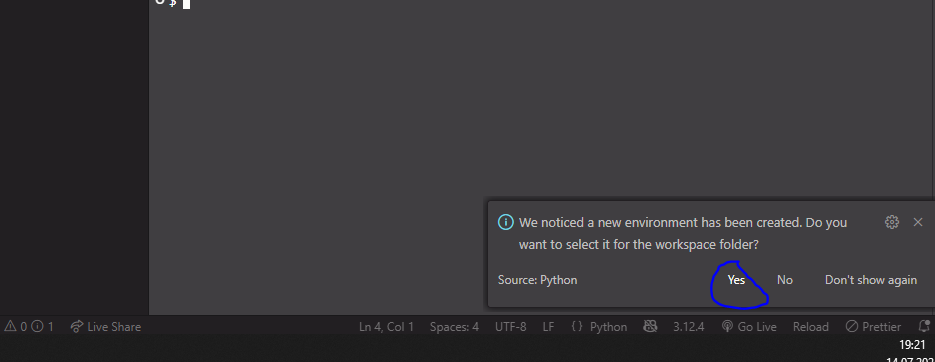
В команде *python -m venv venv* первый venv означает, что мы обращаемся к инструменту, создающему виртуальное, а второй venv – название папки, где будут хранится необходимые файлы

**Важно! Для Windows в VSCode**

*# Создание окружения*

python -m venv venv

Справа должно появится приглашение переключиться на новое окружение



Выбираем Yes

Но окружение еще не активно. В VSCode под Windows надо ввести следующую команду

source venv/Scripts/activate

Если активация не произошла, можно использовать команду (не испытывал)

Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Unrestricted -Force

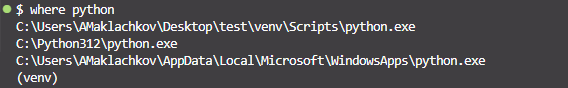
**Проверка:**

1. Должна появится надпись venv в командной строке терминала (после нажатия enter)

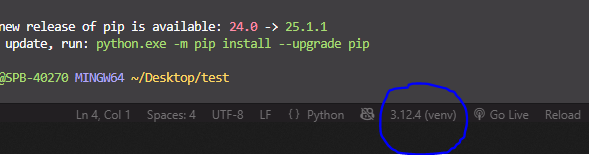


1. Команда

where python

должна показывать список путей к Питону. На первом месте должен идти путь к Питон в нашем виртуальном окружении(в нашем проекте), например, 

1. По идее, внизу должен тоже быть выбран нужный интерпретатор (из нашего виртуального окружения). Но он автоматически не выбирался, приходилось вручную



4. Команда

python -c "import sys; print(sys.executable)"

(это аналог того, что мы записали бы этот код в файле питон и запустили бы его)

Должна вывести путь к нашему интерпретатору:



Установку зависимостей из файла requirements.txt смотри в разделе [pip](#_pip)

## Создание пакетов

**Пакет (package)** — это просто директория с файлом *\_\_init\_\_.py*, которая может содержать модули (.py файлы) и другие пакеты (*подпакеты*). Начиная с *Python 3.3+*, файл *\_\_init\_\_.py* не обязателен, но рекомендован для явной организации.

Пакеты нужны, если требуется:

* Организовать код в структуру, удобную для повторного использования
* Опубликовать его на PyPI
* Работать с большими проектами

**Структура**

*# MyPackage/                       # Основной пакет*

*# ├── SubPackage/                  # ПодпакетОсновной пакет*

*# │   ├── \_\_init\_\_.py              # Инициализация пакета*

*# │   ├── subscript.py             # Модуль с функцией subreport*

*# ├── \_\_init\_\_.py                  # Инициализация пакета*

*# └── main\_script.py               # Модуль с функцией report\_main*

*#*

*#*

*# myprogram.py*

**myprogram.py**

from MyPackage import main\_script

from MyPackage.SubPackage import subscript

main\_script.report\_main()  *# Мы находимся в функции внутри main\_script.py*

subscript.subreport()  *# Мы находимся в функции внутри subscript.py*

**Важно!**

Обратить внимание: для установки пакета в качестве локального пакета или пакета для тестирования, надо в корне пакета создать файл **setup.py** или более продвинутый **pyproject.toml** (инструкцию по наполнению этих файлов смотри отдельно)

## Conda

**Conda** – пакетный менеджер для разработчиков data-science.

* Устанавливает, обновляет и удаляет пакеты (библиотеки).
* Создаёт изолированные окружения (environments), чтобы избежать конфликтов между версиями пакетов.
* Работает не только с Python, но и с другими языками (R, Julia и др.).
* Управляет зависимостями автоматически.

**Anaconda** – это **дистрибутив** (набор программного обеспечения), который включает в себя:

* Conda (менеджер пакетов и сред)
* Python
* Сотни предустановленных популярных пакетов для Data Science, Machine Learning, анализа данных (numpy, pandas, matplotlib, scikit-learn, jupyter и т.д.)
* Графический интерфейс Anaconda Navigator (для тех, кто не любит командную строку)
* Инструменты вроде Spyder, Jupyter Notebook

Скачиваем Anaconda с официального сайта <https://www.anaconda.com/> и устанавливаем.

|  |  |
| --- | --- |
| **conda init** | Выполнять только один раз, при первом использовании Conda на новой системе или в новой оболочке (shell).  Чтобы иметь возможность использовать команды conda activate, conda deactivate без ошибок |
| **conda env list** | Показывает список всех виртуальных окружений conda, которые есть на компьютере |
| **conda create -n env\_name python=3.5** | Создает новое виртуальное окружение с именем env\_name и устанавливает в него Python 3.5. |
| **conda env update -n env\_name -f file.yml** | Conda добавит или обновит пакеты в указанном окружении в соответствии с содержимым файла file.yml. |
| **conda env create -f environment.yml** | Создает новое виртуальное окружение Conda на основе конфигурации из файла environment.yml |
| **conda activate env\_name**  Устаревший аналог:  **source activate env\_name** | Активация окружения Conda. |
| **conda deactivate**  Устаревший аналог:  **source deactivate** | Деактивация окружения Conda. |
| **conda clean -a** | Полностью очищает кэш Conda, удаляя все ненужные файлы.  Флаг **-a** означает «*all*» — выполнить все виды очистки. |
| **conda install package-name** | Установить зависимость |
| **conda env export --name ENV\_NAME > environment.yml** | Выгрузить зависимости в файл environment.yml |

Важно!  
Если окружение не создается или зависимости не ставятся из-за ошибки сертификата, можно добавлять флаг --insecure

**conda create -n env\_name python=3.5 --insecure**

**conda install requests –insecure**

## poetry

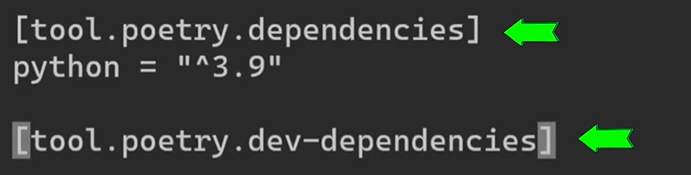
**Poetry** – это современный, мощный и удобный менеджер зависимостей и упаковки Python-проектов

Он позволяет:

* Управлять зависимостями (как pip + requirements.txt, но умнее).
* Создавать виртуальные окружения автоматически.
* Упаковывать и публиковать проекты (как setuptools + twine, но проще).
* Управлять версиями проекта, метаданными, скриптами и т.д.

В отличие от pip, при установке зависимостей он **poerty** сразу сохраняет зависимости в дерево зависимостей в **pyproject.toml**

Как в NPM, в poerty позволяет в одном файле управлять зависимостями и для продакшена, и для разработки



При работе с pip обычно хранят два файла requirements.txt и requirements-dev.txt

**Использование:**

1. **pip install poetry**
2. **poetry init**

Будет создан *pyproject.toml*

1. **poetry env use python3.10**

Создает виртуальное окружение с Питон 3.10

При этом в папке проекта не будет создана папка виртуального окружения (в отличии от классического *python -m venv venv*)

1. **env activate**

Активировать виртуальное окружение

1. **poerty add numpy**

|  |  |
| --- | --- |
| **poetry init** | Инициализация проекта |
| **poetry env use python3.10**  или  **poetry env use python** (для использования дефолтной версии Питон)  Иногда poetry env use python3.12 не находит Питон. Использовать команду:  **poetry env use "C:\Users\ТВОЙ\_ПОЛЬЗОВАТЕЛЬ\AppData**  **\Local\Programs\Python\Python312\python.exe"** | Создает виртуальное окружение с определенной версией Python. При этом в папке проекта не будет создана папка виртуального окружения (в отличии от классического *python -m venv venv*) |
| **env activate**  **poetry shell** – старая команда для Windows сейчас используем env activate) | Активировать виртуальное окружение |
| **poetry show --tree** | Вывести дерево зависимостей |
| **poetry show --latest** | Показать доступные обновления пакетов. |
| **poetry add numpy** | Установить зависимость |
| **poetry remove numpy** | Удалить зависимость |
| **poetry config --list** | Список конфигов виртуального окружения |
| **poetry config --unset certificates.PyPI.cert** | Удалить конкретный конфиг |

### Фикс установки зависимостей в poetry

Если зависимости не устанавливаются по причине ошибки сертификатов

**poetry add requests -v**

Вывод:

Exception:

| HTTPSConnectionPool(host='pypi.org', port=443): Max retries exceeded with url: /simple/requests/ (Caused by SSLError(SSLCertVerificationError(1, '[SSL: CERTIFICATE\_VERIFY\_FAILED] certificate verify failed: unable

to get local issuer certificate (\_ssl.c:1028)')))

Варианты решения:

1. Отключить проверку сертификатов для PyPi – тут важен регистр (но этот способ не рекомендуется как небезопасный)

Регистр букв в PyPi важен!!!

**poetry config certificates.PyPI.cert false**

1. Если не помогло, то такой набор команд:

poetry source add fpho https://files.pythonhosted.org

Adding source with name fpho

poetry config certificates.fpho.cert false

poetry source add pypi

poetry config certificates.PyPI.cert false

poetry config certificates.pypi.cert false

1. Или сменить в Windows DNS на DNS гугла (не пробовал).

Посмотреть текущий список конфигов:

**poetry config --list**

Удалить конкретный конфиг

**poetry config --unset certificates.PyPI.cert**

# Numpy, Pandas, math

Python — интерпретируемый язык, а интерпретируемые языки менее производительные, чем компилируемые. При этом он повсеместно используется для задач, в которых нужна высокая производительность, — например, в машинном обучении и анализе больших данных.

Это возможно благодаря специализированным библиотекам, таким как *math, numpy, pandas*, которые реализованы на более низкоуровневых компилируемых языках (например, *C и Fortran*).

## math

Модуль math покрывает широкий спектр математических задач, связанных с отдельными числами и стандартными функциями. Однако при работе с большими объёмами данных или сложными структурами — такими как векторы, матрицы или таблицы чисел — его возможностей становится недостаточно.

В таких случаях на помощь приходит библиотека numpy, которая предназначена для эффективной работы с массивами и числовыми данными в целом. Именно она лежит в основе большинства научных и аналитических расчётов в Python.

|  |  |  |
| --- | --- | --- |
| **Функции теории чисел и представления**  Эти функции пригодятся для вычислений, связанных с комбинаторикой, делимостью и произведением чисел. | | |
| **comb(n, k)** | Вычисляет число сочетаний из n по k. Это количество способов выбрать k объектов из n, без учёта порядка и без повторений. | import math  result = math.comb(12, 3)  print(result) *# 220* |
| **factorial(x)** | Возвращает факториал числа x, то есть произведение всех целых чисел от 1 до x. | result = math.factorial(5)  print(result)  *# 120* |
| **gcd(\*integers)** | Возвращает наибольший общий делитель (НОД) для всех аргументов. | result = math.gcd(120, 210, 360)  print(result)  *# 30* |
| **lcm(\*integers)** | Возвращает наименьшее общее кратное (НОК). | result = math.lcm(10, 20, 30, 40)  print(result)  *# 120* |
| **perm(n, k=None)** | Возвращает количество размещений из n по k элементов (учитывается порядок). Если k не указано, считается количество перестановок — размещений всех элементов. | result1 = math.perm(4, 2)  print(result1)  *# 12*  result2 = math.perm(4)  print(result2)  *# 24* |
| **math.prod(iterable, start=1)** | Перемножает все элементы в переданном итерируемом объекте (например, списке или диапазоне). Если объект пуст, возвращается значение аргумента start (по умолчанию — 1) | result1 = math.prod(range(10, 21))  print(result1)  *# 6704425728000*  result2 = math.prod(range(10, 10), *start*=777)  print(result2)  *# 777* |
| **Степенные и логарифмические функции**  Эти функции используются для работы с экспонентами, логарифмами и возведением в степень. | | |
| **exp(x)** | Возвращает значение e^x, где *e* — математическая константа (приблизительно *2.71828*) | result = math.exp(3.5)  print(result)  *# 33.11545195869231* |
| **log(x, base)** | Вычисляет логарифм x по основанию base. Если основание не указано, возвращается натуральный логарифм (по основанию e). | result1 = math.log(10)  print(result1)  *# 2.302585092994046*  result2 = math.log(10, 2)  print(result2)  *# 3.3219280948873626* |
| **pow(x, y)** | Аналог оператора \*\*, но оба аргумента преобразуются в float, что обеспечивает точность при вещественных вычислениях. | result1 = math.pow(2, 10)  print(result1)  *# 1024.0*  result2 = math.pow(4.5, 3.7)  print(result2)  *# 261.1477575641718* |
| **Тригонометрические функции**  Функции из этой группы работают с углами и используют радианы (а не градусы) по умолчанию | | |
| **sin(x)**  **cos(x)**  **tan(x)** | Стандартные тригонометрические функции. |  |
| **asin(x)**  **acos(x)**  **atan(x)** | Обратные функции. |  |
| **dist(p, q)** | *Евклидово расстояние* – расстояние между точками p и q в пространстве. Каждая точка задаётся как итерируемый объект. | result = math.dist((0, 0, 0), (1, 1, 1))  print(result)  *# 1.7320508075688772* |
| **hypot(\*coordinates)** | Для двух координат – *гипотенуза*.  В остальных случаях – длина многомерного вектора, идущего от начала координат. | result1 = math.hypot(1, 1, 1)  print(result1)  *# 1.7320508075688772*  result2 = math.hypot(3, 4)  print(result2)  *# 5.0* |
| **Преобразование углов** | | |
| **degrees(x)** | Переводит угол из радиан в градусы. | result = math.degrees(math.asin(0.5))  print(round(result, 1))  *# 30.0* |
| **radians(x)** | Переводит угол из градусов в радианы. | result = math.sin(math.radians(30))  print(round(result, 1))  *# 0.5* |
| **Гиперболические функции** | | |
| **sinh(x)**  **cosh(x)**  **tanh(x)** | Гиперболические аналоги   * синуса * косинуса * тангенса |  |
| **asinh(x)**  **acosh(x)**  **atanh(x)** | Функции, обратные стандартным гиперболическим функциям |  |
| **Специальные функции** | | |
| **gamma(x)** | Это обобщение факториала на вещественные числа.  Для целых n выполняется gamma(n) = (n-1)!. | result1 = math.gamma(3)  print(result1)  *# 2.0*  result2 = math.gamma(3.5)  print(result2)  *# 3.323350970447842*  result3 = math.gamma(4)  print(result3)  *# 6.0* |
| **Математические константы** | | |
| **pi** | Значение числа π  (3.1415926…) | print(math.pi)  *# 3.141592653589793* |
| **e** | Основание натурального логарифм  (2.71828…) | print(math.e)  *# 2.718281828459045* |

## numpy

Модуль math покрывает широкий спектр математических задач, связанных с отдельными числами и стандартными функциями. Однако при работе с большими объёмами данных или сложными структурами — такими как векторы, матрицы или таблицы чисел — его возможностей становится недостаточно.

В таких случаях на помощь приходит библиотека numpy, которая предназначена для эффективной работы с массивами и числовыми данными в целом. Именно она лежит в основе большинства научных и аналитических расчётов в Python.

### Массивы numpy

**Массивы** — это структуры данных, в которых:

* все элементы имеют **одинаковый тип** (например, только числа с плавающей точкой);
* данные хранятся компактно и последовательно в памяти;
* возможна высокая производительность при выполнении операций сразу над всеми элементами.

**Создать массив np.array()**

import numpy as np

a = np.array([1, 2, 3, 4])

b = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])

print(*f*"a[0] = {a[0]}")  *# a[0] = 1*

print(*f*"b[0] = {b[0]}")  *# b[0] = [1 2]*

**Аттрибуты массивов**

Массивы в numpy — это объекты класса **ndarray** (англ. *n-dimensional array*), и каждый такой объект хранит информацию о своей структуре и содержимом.

|  |  |
| --- | --- |
| **ndim** | Количество осей массива (его размерность) |
| **shape** | Кортеж с количеством элементов по каждой из осей |
| **size** | Общее число элементов в массиве |
| **dtype** | Тип данных, хранящихся в массиве |
| **itemsize** | Количество байт, которое занимает один элемент |

При создании массива можно *явно* указать тип хранимых данных.

a = np.array([1, 2, 3, "4"], *dtype*="uint8")

print(*type*(a.dtype))  *# <class 'numpy.dtypes.UInt8DType'>*

print(*type*(a[3]))  *# <class 'numpy.uint8'>*

При этом мы можем добавлять в массив элемент, если он может быть приведен в указанному типу (например, положить str в массив uint8)

Если не указать явно тип при создании массива, numpy попытается определить его автоматически на основе всех переданных значений.

Если смешать числа и строки, которые нельзя привести к числу, то получим тип **<U32**, который обозначает *Unicode-строки длиной до 32 символов*.

a = np.array([1, 2, "3"])

print(a)  *# <['1' '2' '3']*

print(*type*(a.dtype))  *# <class 'numpy.dtypes.StrDType'>*

b = np.array(["text", 1, 2.5])

print(b)  *# ['text' '1' '2.5']*

print(b.dtype)  *# <U32*

**Операции с массивами**

|  |  |  |
| --- | --- | --- |
| **zeros()** | Создания массива (или матрицу) из нулей | a = np.zeros((2, 3))  print(a)  *# [[0. 0. 0.]*  *#  [0. 0. 0.]]* |
| **ones()** | Создания массива (или матрицу) из единиц |  |
| **eye()** | Создание единичной матрицы (единицы на главной диагонали, нули в остальных ячейках) | a = np.eye(3, 3, *dtype*="int8")  print(a)  *# [[1 0 0]*  *#  [0 1 0]*  *#  [0 0 1]]* |
| **arange()** | Создаёт массив, аналогичный range(), но возвращает именно массив numpy. Работает как с целыми, так и с вещественынми числами. | a = np.arange(1, 5, 0.4)  print(a)  *# [1.  1.4 1.8 2.2 2.6 3.  3.4 3.8 4.2 4.6]* |
| **linspace()** | Создаёт массив из заданного числа равномерно распределённых значений на отрезке | a = np.linspace(1, 5, 10)  print(a)  *# Десять равномерно распределенных чисел на отрезке от 1 до 5*  *# [1. 1.44444444 1.88888889 2.33333333 2.77777778 3.22222222 3.66666667 4.11111111 4.55555556 5.]* |
| **reshape()** | Позволяет задать новую форму массива. Количество элементов при этом должно сохраняться | a = np.zeros((4, 3), *dtype*="uint8")  a = a.reshape((2, 6))  print(a)  *# [[0 0 0 0 0 0]*  *#  [0 0 0 0 0 0]]* |
| **resize()** | Работает как *reshape,* но изменяет исходнйы массив.  Если в reshape() указать -1 по одной из осей, numpy автоматически вычислит нужное значение | a = np.array([[1, 2, 3], [4, 5, 6]])  d = a.reshape(-1, 3)  print(d)  *# [[1 2 3]*  *#  [4 5 6]]* |
| **+, -, \*, /** | *Арифметические операции*  Применяются поэлементно. Размерности массивов при этом должны совпадать |  |
| **@** или **dot()** | Матричное умножение |  |
| **transpose()** | Транспонирование массива  Меняет оси местами |  |
| **rot90()** | Поворачивает массив на 90 градусов (по умолчанию — против часовой стрелки) |  |
| **sum()** | Сумма всех элементов массива | Можно указать ось, по который делать вычисления:  a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  print(a.sum(*axis*=0))  *# Сумма по столбцам*  *# [12 15 18]* |
| **min()** | Минимальный элемент в массиве |
| **max()** | Максимальный элемент в массиве |
| **roll(array, shift, axis=None)**  **array –** n-мерный массив  **shift –** на сколько сдвигаем  **axios –** ось сдвига | Циклически сдвигает элементы массива вдоль указанной оси.  Элементы, которые "выходят" за край, появляются с противоположной стороны. | array = np.array([0, 1, 2, 3, 4])  print(np.roll(array, 1))  *# [4 0 1 2 3]*  matrix = np.array([[1, 2, 3], [4, 5, 6]])  *#*  print(np.roll(matrix, 1))  *# → [[6 1 2]*  *#  [3 4 5]]*  *# Сдвиг по строкам (axis=0): строки двигаются вниз*  print(np.roll(matrix, 1, *axis*=0))  *# → [[4 5 6]*  *#     [1 2 3]]*  *# Сдвиг по столбцам (axis=1): элементы в каждой строке сдвигаются вправо*  print(np.roll(matrix, 1, *axis*=1))  *# → [[3 1 2]*  *#     [6 4 5]]* |
| **np.tile(array, reps)** | Возвращает матрицу, в которой повторяется переданный массив заданное количество раз вдоль указанных измерений | a = np.array([1, 2, 3])  result1 = np.tile(a, 3)  print(result1)  *# → [1 2 3 1 2 3 1 2 3]*  result2 = np.tile(a, (3, 1))  *# 3 раза по оси 0, 1 раз по оси 1*  print(result2)  *# → [[1 2 3]*  *#     [1 2 3]*  *#     [1 2 3]]* |
| **flat** | Итератор. Линеаризация массива  (перевод матрицы в простой список значений) | a = np.arange(1, 13).reshape(3, 4)  print("; ".join(*str*(el) for el in a.flat))  *# 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12* |

**Срезы**

a = np.arange(1, 13).reshape(3, 4)

print(a)

print(a[:2, 2:])  *# Срез: 2 строки, последние 2 столбца*

print(a[:, ::2])  *# Срез всех строк, каждый второй столбец*

## pandas

В библиотеке pandas определены **два основных класса объектов** для работы с данными:

* Series — одномерный массив, способный хранить значения любого типа. По своей структуре напоминает *словарь*: каждому значению присваивается *метка (индекс), которая может быть как числом, так и строкой.*
* **DataFrame** — двумерная таблица, в которой строки и столбцы имеют имена. *Каждый столбец* — это объект класса *Series*, а сами данные удобно организованы для анализа и преобразований.

### Series

Объект **Series** во многом напоминает словарь: каждому значению соответствует собственная метка.

s = pd.Series(data, *index*=index)

* **data** может быть *массивом numpy, словарём* или *скаляром (числом).*
* *index* — список меток, по умолчанию это целые числа от 0 до n-1. Если *data — словарь*, а index не задан, то в качестве меток используются *ключи* словаря

Если передаем число, то количество строк будет равно количеству индексов:

s = pd.Series(5, *index*=["a", "b", "c"])

print(s)

*# a    5*

*# b    5*

*# c    5*

*# dtype: int64*

s = pd.Series(np.arange(5), *index*=["a", "b", "c", "d", "e"])

print(s)

*# Вывод программы*

*# a    0*

*# b    1*

*# c    2*

*# d    3*

*# e    4*

*# dtype: int32*

s = pd.Series(np.linspace(0, 1, 5))

print(s)

*# 0    0.00*

*# 1    0.25*

*# 2    0.50*

*# 3    0.75*

*# 4    1.00*

*# dtype: float64*

Если индексы не совпадают с ключами, то по несуществующему индекс/ключу будет NaN

d = {"a": 10, "b": 20, "c": 30, "g": 40}

s = pd.Series(d, *index*=["a", "b", "c", "d"])

print(s)

*# a    10.0*

*# b    20.0*

*# c    30.0*

*# d     NaN*

*# dtype: float64*

### Индексация, срезы и арифметика

Объекты Series поддерживают индексирование, срезы и математические операции:

s = pd.Series(np.arange(5), *index*=["a", "b", "c", "d", "e"])

print(s[["a", "d"]])  *# Несколько элементов*

*# a    0*

*# d    3*

*# dtype: int64*

print(s[3:])  *# Срез*

*# d    3*

*# e    4*

*# dtype: int64*

print(s + s)  *# Поэлементное сложение*

*# a    0*

*# b    2*

*# c    4*

*# d    6*

*# e    8*

*# dtype: int64*

### Фильтрация Series

Можно использовать булевы условия для отбора значений:

s = pd.Series(np.arange(5), *index*=["a", "b", "c", "d", "e"])

print(s[s > 2])

*# d    3*

*# e    4*

*# dtype: int64*

### Атрибуты Series

**name** – имя самой *серии* (будет именем столбца при преобразовании в DataFrame).

**index.name –** имя индекса (метка оси строк)

s = pd.Series(np.arange(3), *index*=["a", "b", "c"])

s.name = "Данные"

s.index.name = "Это индекс"

print(s)

*# Это индекс*

*# a    0*

*# b    1*

*# c    2*

*# Name: Данные, dtype: int64*

### DataFrame

**DataFrame** — это таблица с именованными столбцами и индексами строк.

*Каждый столбец* DataFrame на самом деле представляет собой отдельный *объект Series*.

**Создать DataFrame**

students\_marks\_dict = {

    "student": ["Студент\_1", "Студент\_2", "Студент\_3"],

    "math": [5, 3, 4],

    "physics": [4, 5, 5],

}

students = pd.DataFrame(students\_marks\_dict)

print(students)

*#      student  math  physics*

*# 0  Студент\_1     5        4*

*# 1  Студент\_2     3        5*

*# 2  Студент\_3     4        5*

Или так:

data = [

    ['Анна', 25, 'Москва'],

    ['Борис', 30, 'СПб'],

    ['Вера', 35, 'Новосибирск']

]

df = pd.DataFrame(data, *columns*=['Имя', 'Возраст', 'Город'])

print(df)

Или так:

data = [

    {'Имя': 'Анна', 'Возраст': 25, 'Город': 'Москва'},

    {'Имя': 'Борис', 'Возраст': 30, 'Город': 'СПб'},

    {'Имя': 'Вера', 'Возраст': 35}

]

df = pd.DataFrame(data)

print(df)

Или так:

*# Пустой DataFrame*

df = pd.DataFrame()

*# или с заданными столбцами:*

df = pd.DataFrame(*columns*=["Имя", "Возраст"])

У объекта DataFrame есть *индексы по строкам и по столбцам*. Их можно получить с помощью свойств **index** и **columns**:

print(students.index)

*# RangeIndex(start=0, stop=3, step=1)*

print(students.columns)

*# Index(['student', 'math', 'physics'], dtype='object')*

По умолчанию строки нумеруются от 0. При необходимости индексы можно переопределить вручную:

students.index = ["A", "B", "C"]

print(students.index)

*# Index(['A', 'B', 'C'], dtype='object')*

### Индексация и срезы DataFrame

Для доступа к строкам по индексам используют **.loc[]**, включая срезы:

Можно использовать и стандартный синтаксис, но более читаемым является *.loc*

*Примечание:* при использовании среза с метками (в примере .loc[‘B’:]) mypy будет ругаться на то, что индексы в срезах должны быть числами. Один из рекомендованных способов – использовать # type: ignore[index] или # type: ignore[misc]

students\_marks\_dict = {

    "student": ["Студент\_1", "Студент\_2", "Студент\_3"],

    "math": [5, 3, 4],

    "physics": [4, 5, 5],

}

students = pd.DataFrame(students\_marks\_dict)

students.index = ["A", "B", "C"]

print(students["B":])  *# type: ignore[misc]*

*#      student  math  physics*

*# B  Студент\_2     3        5*

*# C  Студент\_3     4        5*

print(students[:2])

*#      student  math  physics*

*# A  Студент\_1     5        4*

*# B  Студент\_2     3        5*

|  |  |
| --- | --- |
| dataFrame.**head(n)** | Получение первых n строк датасета.  По умолчанию равно 5. возвращается пять первых строк. |
| dataFrame.**tail(n)** | Получение последних n строк датасета.  По умолчанию равно 5. |

### Методы для работы с DataFrame

**Важно!** Далее в примерах в этом разделе будет использована загрузка датасета из файла .csv на подобии:

"gender","race/ethnicity","parental level of education","lunch","test preparation course","math score","reading score","writing score"

"female","group B","bachelor's degree","standard","none","72","72","74"

"female","group C","some college","standard","completed","69","90","88"

students = pd.read\_csv("StudentsPerformance.csv")

**Фильтрация**

В качестве индекса можно использовать условия для фильтрации данных.

Выберем студентов, прошедших подготовку – у которых в колонке *"test preparation course"* указано *"completed"*

with\_course = students[students["test preparation course"] == "completed"]

**Сортировка sort\_values()**

Из предыдущей выборки возьмем первых 5 студентов (метод head() без аргументов) и отсортируем по количеству баллов по трем дисциплинам *math, reading, writing*

Сортируем по убыванию (*ascending=False*). Сортировка будет происходить вначале значениям из столбца *math score*, если значения в нем равны, то по *reading score* и т.д.

sorted\_students\_with\_course = (

    with\_course[["math score", "reading score", "writing score"]]

    .sort\_values(

        ["math score", "reading score", "writing score"], *ascending*=False

    )

    .head()

)

**Добавление новой колонки**

Добавим колонку tota score – общее количество баллов по трем предметам.

1. Первый способ – **мутируем** имеющийся DataFrame

students["total score"] = (

    students["math score"]

    + students["reading score"]

    + students["writing score"]

)

*# Сортируем по убыванию общего количества баллов и выводим первых 5*

print(students.sort\_values("total score", *ascending*=False).head())

1. Второй способ – **создаем новый** Data Frame с помощью метода **assign()** (через lambda-функцию)

scores = students.assign(

*total\_score*=*lambda* *x*: x["math score"]

    + x["reading score"]

    + x["writing score"]

)

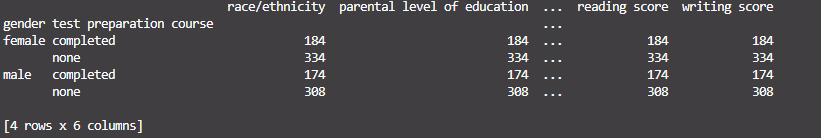
*# Сортируем по убыванию общего количества баллов и выводим первых 5*

print(scores.sort\_values("total\_score", *ascending*=False).head())

**Группировка groupby()**

Метод groupby() позволяет сгруппировать записи по признаку. Например, узнаем, сколько студентов разного пола прошли подготовку:

print(students.groupby(["gender", "test preparation course"]).count())



Вместо count можно использовать другие методы для получения сводных данных **agg** (среднее, медиана, сумма и т. п.).

Например

agg\_functions = {"math score": ["mean", "median"]}

print(students.g

### Чтение и запись в файл

Обычно табличные данные хранятся в файлах. Такие наборы принято называть датасетами (англ. dataset, набор данных). Файлы с данными могут быть в разных форматах: CSV, Excel, JSON, SQL и другие. Библиотека pandas поддерживает как чтение, так и сохранение таблиц в этих форматах.

* **CSV-файлы**

*# Считывание:*

dataFrame = pd.read\_csv("имя\_файла.csv")

*# Сохранение:*

dataFrame.to\_csv("имя\_файла.csv")

* **Excel-файлы (формат 2007+)**

*# Считывание:*

dataFrame = pd.read\_excel("имя\_файла.xlsx")

*# Сохранение:*

dataFrame.to\_excel("имя\_файла.xlsx")

* **JSON-файлы**

*# Считывание:*

dataFrame = pd.read\_json("имя\_файла.json")

*# Сохранение:*

dataFrame.to\_json("имя\_файла.json")

## matplotlib

Для построения графиков pandas использует библиотеку matplotlib

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

students = pd.read\_csv("StudentsPerformance.csv")

plt.hist(students["math score"], *label*="Тест по математике")

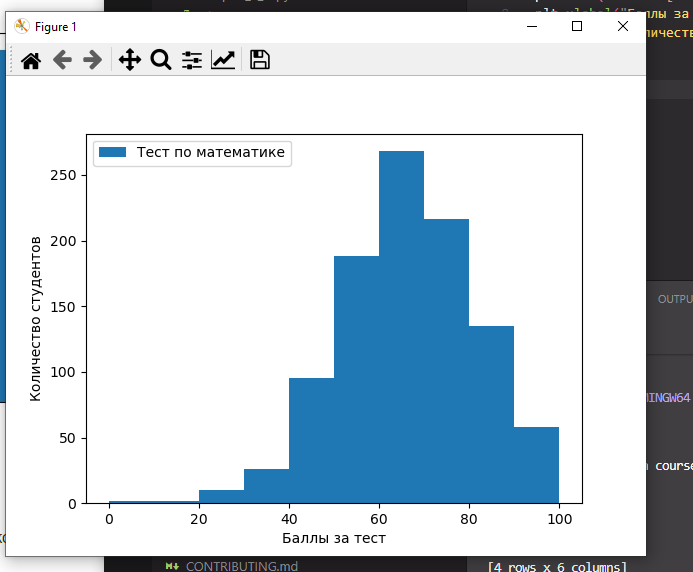
plt.xlabel("Баллы за тест")

plt.ylabel("Количество студентов")

plt.legend()

plt.show()

В результате выполнения кода открывается новое окно**(!)**, в котором мы увидим вот такой график



## requests

### Фикс ошибки сертификатов

Для исправления ошибки

certificate verify failed: unable to get local issuer certificate

Можно установить бибилоитеку:

pip install pip-system-certs

### Пример запроса

from requests import get

*# Получаем файл-изображение участка карты по указанным координатам*

response = get(

    "https://static-maps.yandex.ru/1.x/?"

    "ll=37.677751,55.757718&"

    "spn=0.016457,0.00619&"

    "l=map"

)

print(response)

*# Вывод программы*

*# <Response [200]>*

*# Записываем изображение в файл map.png на диск*

with open("map.png", "wb") as file:

    file.write(response.content)

# Линтеры

## Настройка pre-commit

Одним из инструментов является <https://pre-commit.com>

На главной странице есть инструкция по установке:

1. **pip install pre-commit**
2. Командой **pre-commit –version** проверяем, что установился

pre-commit --version

pre-commit 4.3.0

1. Добавляем в корень проекта файл **.pre-commit-config.yaml**

  - repo: https://github.com/pylint-dev/pylint

    rev: "v3.3.1"

    hooks:

      - id: pylint

        name: pylint

        entry: pylint

        language: python

        types: [python]

        args:

          [

            "--ignore=no\_check\*,\_\_init\_\_.py",

            "--max-line-length=90",

            "--const-naming-style=any",

            "--disable=E0401,W0104,R0903,R1721,E1101,E0611,F0002,C0305,line-too-long,C0303,E2515"

          ]

        additional\_dependencies: [pylint]

**Важно!**

В disable записываем коды ошибок, которые надо игнорировать

1. Далее устанавливаем хуки git

Эта команда устанавливает Git-хук *pre-commit* в *.git/hooks/*, который будет **автоматически** вызываться при каждом *git commit*.

pre-commit install

pre-commit installed at .git/hooks/pre-commit

1. Чтобы запустить хуки вручную

pre-commit run --all-files

## Список линтеров

Это основные линтеры

* **docformatter** - форматирует docstrings.
* **Add trailing commas** - добавляет запятые в конце списков/словарей/аргументов.
* **black** - автоматический форматтер кода.
* **nbqa-black** - применяет black для файлов \*.ipynb.
* **pyupgrade** - обновляет синтаксис Python до более современного.
* **nbqu-pyupgrade** - применяет pyupgrade для файлов \*.ipynb.
* **isort** - сортирует и группирует импорты.
* **nbqu-isort** - применяет isort для файлов \*.ipynb.
* **blacken-docs** - применяет Black к коду внутри документации.
* **trim trailing whitespaces** - удаляет пробелы в конце строк.
* **check yaml** - проверяет синтаксис YAML-файлов.
* **debug statements (python)** - ищет оставленные print(), pdb.set\_trace(), breakpoint() и т.п.
* **python tests naming** - проверяет, что тесты названы по соглашению (например, test\_\*.py, Test\*, test\_\*())
* **fix requirements.txt** - нормализует/исправляет requirements.txt.
* **codespell** - орфографические ищет ошибки в коде и комментариях.
* **flake8** - проверяет стиль кода и ошибки.
* **nbqa-flake8** - применяет flake8 для файлов \*.ipynb.
* **mypy** - статический анализ типов.
* **nbqa-mypy** - применяет mypy для файлов \*.ipynb.
* **pylint** - мощный линтер, который проверяет стиль, ошибки, архитектура, дублирование кода.
* **nbqa-pylint** - применяет pylint для файлов \*.ipynb.
* **nbqa-pydocstyle** - проверяет соответствие docstrings стандарту PEP 257 для файлов \*.ipynb.

## pyproject.toml

**pyproject.toml** — это стандартный конфигурационный файл в формате TOML, используемый в Python-проектах для определения метаданных проекта, зависимостей и настроек инструментов.

То есть в нем можно, например, кастомизировать настройки линтеров (чтобы все настройки были в одном месте) и перенести в него директивы args, удалив их из .pre-commit-config.yaml

[tool.poetry]

name = "my\_project"

version = "0.1.0"

description = "Пример проекта"

authors = ["Ваше Имя <you@example.com>"]

license = "MIT"

[tool.poetry.dependencies]

python = "^3.10"

[tool.poetry.dev-dependencies]

black = "^24.3.0"

pylint = "^3.0.0"

[build-system]

requires = ["poetry-core>=1.0.0"]

build-backend = "poetry.core.masonry.api"

[tool.black]

line-length = 88 # flake у Сенаторов имеет максимальную длину 79. Лучше указывать одинаково (например, 79), чтобы не было конфликтов

target-version = ["py310"]

[tool.pylint]

max-line-length = 88

disable = ["C0114", "C0115", "C0116"]  # Отключение предупреждений о недостающих docstring'ах

## Конфигурация .pre-commit-config.yaml

Это конфигурация для указанных выше линтеров от Сенаторов

repos:

  - repo: https://github.com/mwouts/jupytext

    rev: "v1.16.4b"

    hooks:

     - id: jupytext

       entry: jupytext

       language: python

       types: [jupyter]

       args:

         - --from=ipynb

         - --to=py:light

         - --set-formats=ipynb,py:light

         - --quiet

         - --sync

         - --warn-only

       exclude: '.\*\.md$'

  - repo: https://github.com/PyCQA/docformatter

    rev: "eb1df347edd128b30cd3368dddc3aa65edcfac38"

    hooks:

      - id: docformatter

        args: [--in-place, --wrap-descriptions=90, --style=google]

  - repo: https://github.com/psf/black.git

    rev: "24.10.0"

    hooks:

      - id: black

  - repo: https://github.com/nbQA-dev/nbQA

    rev: 1.9.0

    hooks:

      - id: nbqa-black

        name: nbqa-black

        description: Run 'black' on a Jupyter Notebook

        entry: nbqa black

        language: python

        require\_serial: true

        types\_or: [jupyter, markdown]

        additional\_dependencies: [black]

  - repo: https://github.com/asottile/pyupgrade

    rev: "v3.19.0"

    hooks:

      - id: pyupgrade

        args: [--py39-plus]

  - repo: https://github.com/pycqa/isort

    rev: "5.13.2"

    hooks:

      - id: isort

        entry: isort

        args:

          - --profile=black

          - --float-to-top

  - repo: https://github.com/nbQA-dev/nbQA

    rev: "1.9.0"

    hooks:

      - id: nbqa

        entry: nbqa blacken-docs

        name: nbqa-blacken-docs

        alias: nbqa-blacken-docs

        additional\_dependencies: [blacken-docs]

        args:

          - --nbqa-md

  - repo: https://github.com/nbQA-dev/nbQA

    rev: 1.9.0

    hooks:

      - id: nbqa-pyupgrade

        name: nbqa-pyupgrade

        description: Run 'pyupgrade' on a Jupyter Notebook

        entry: nbqa pyupgrade

        language: python

        require\_serial: true

        types\_or: [jupyter, markdown]

        additional\_dependencies: [pyupgrade]

      - id: nbqa-isort

        name: nbqa-isort

        description: Run 'isort' on a Jupyter Notebook

        entry: nbqa isort

        language: python

        require\_serial: true

        types\_or: [jupyter, markdown]

        additional\_dependencies: [isort]

        args:

          - --profile=black

          - --float-to-top

  - repo: https://github.com/codespell-project/codespell

    rev: v2.3.0

    hooks:

      - id: codespell

        additional\_dependencies:

          - tomli

        args:

          - --ignore-regex=^\s\*"image\/(jpeg|png|gif|bmp|tiff)":\s.\*

          - --ignore-regex=[A-Za-z0-9+/]{100,}

          - --skip=\*.js,\*.html,\*.css,\*.svg",\*.json,\*.png,\*.jpg,\*.yml,\*.yaml

  - repo: https://github.com/nbQA-dev/nbQA

    rev: 1.9.0

    hooks:

      - id: nbqa-flake8

        args:

          - --ignore=E501,E712,W291,F632,E203,F821,F403,W391,F401

          - --exclude=.\*,\_\_init\_\_.py

        name: nbqa-flake8

        description: Run 'flake8' on a Jupyter Notebook

        entry: nbqa flake8

        language: python

        require\_serial: true

        types\_or: [jupyter, markdown]

        additional\_dependencies:

          - flake8-variables-names

          - pep8-naming

          - flake8-functions-names

  - repo: https://github.com/PyCQA/flake8

    rev: 7.1.1

    hooks:

      - id: flake8

        args:

          - --ignore=E501,E712,W291,F632,E203,F821,F403,W391,F401

          - --exclude=.\*,\_\_init\_\_.py

        additional\_dependencies:

          - flake8-variables-names

          - pep8-naming

          - flake8-functions-names

  - repo: https://github.com/nbQA-dev/nbQA

    rev: 1.9.0

    hooks:

      - id: nbqa-mypy

        name: nbqa-mypy

        description: Run 'mypy' on a Jupyter Notebook

        entry: nbqa mypy

        language: python

        require\_serial: true

        types\_or: [jupyter, markdown]

        additional\_dependencies:

          - mypy

          - pandas-stubs

          - git+https://github.com/numpy/numpy-stubs

          - mypy-extensions

          - types-requests

          - types-PyYAML

          - types-setuptools

        args:

*# - --cache-dir=/dev/null*

*# - --cache-dir=nul*

*# - --no-incremental*

          - --non-interactive

          - --install-types

          - --explicit-package-bases

          - --ignore-missing-imports

          - --disallow-untyped-calls

          - --disallow-untyped-defs

          - --disallow-untyped-decorators

          - --strict

          - --extra-checks

          - --disallow-any-decorated

          - --disallow-any-generics

          - --local-partial-types

          - --pretty

          - --force-uppercase-builtins

          - --force-union-syntax

          - --warn-unreachable

          - --warn-redundant-casts

          - --warn-return-any

          - --disallow-any-explicit

  - repo: https://github.com/pre-commit/mirrors-mypy

    rev: v1.13.0

    hooks:

      - id: mypy

        args:

*# - --cache-dir=/dev/null*

*# - --cache-dir=nul*

*# - --no-incremental*

          - --non-interactive

          - --install-types

          - --explicit-package-bases

          - --ignore-missing-imports

          - --disallow-untyped-calls

          - --disallow-untyped-defs

          - --disallow-untyped-decorators

          - --strict

          - --extra-checks

          - --disallow-any-decorated

          - --disallow-any-generics

          - --local-partial-types

          - --pretty

          - --force-uppercase-builtins

          - --force-union-syntax

          - --warn-unreachable

          - --warn-redundant-casts

          - --warn-return-any

          - --disallow-any-explicit

        additional\_dependencies:

          - mypy

          - pandas-stubs

          - git+https://github.com/numpy/numpy-stubs

          - mypy-extensions

          - types-requests

          - types-PyYAML

          - types-setuptools

  - repo: https://github.com/nbQA-dev/nbQA

    rev: 1.9.0

    hooks:

      - id: nbqa-pylint

        name: nbqa-pylint

        description: Run 'pylint' on a Jupyter Notebook

        entry: nbqa pylint

        language: python

        require\_serial: true

        types\_or: [jupyter, markdown]

        additional\_dependencies: [pylint]

        args:

          - --ignore=no\_check\*,\_\_init\_\_.py

          - --max-line-length=90

          - --const-naming-style=any

          - --disable=E0401,W0104,R0903,R1721,E1101,E0611,F0002,C0305,C0303,E2515

  - repo: https://github.com/pylint-dev/pylint

    rev: "v3.3.1"

    hooks:

      - id: pylint

        name: pylint

        entry: pylint

        language: python

        types: [python]

        args:

          [

            "--ignore=no\_check\*,\_\_init\_\_.py",

            "--max-line-length=90",

            "--const-naming-style=any",

            "--disable=E0401,W0104,R0903,R1721,E1101,E0611,F0002,C0305,line-too-long,C0303,E2515"

          ]

        additional\_dependencies: [pylint]

  - repo: https://github.com/nbQA-dev/nbQA

    rev: 1.9.0

    hooks:

      - id: nbqa-pydocstyle

        name: nbqa-pydocstyle

        description: Run 'pydocstyle' on a Jupyter Notebook

        entry: nbqa pydocstyle

        language: python

        require\_serial: true

        types\_or: [jupyter, markdown]

        additional\_dependencies: [pydocstyle]

*# - repo: https://github.com/christophmeissner/pytest-pre-commit*

*#   rev: 1.0.0*

*#   hooks:*

*#     - id: pytest*

*#       name: "Run pytest with HTML report"*

*#       entry: pytest*

*#       args:*

*#          - "--html=senatorov\_report.html"*

*#          - "--self-contained-html"*

*#       language: python*

*#       pass\_filenames: false*

*#       always\_run: true*

*#       additional\_dependencies: [pytest-html]*

  - repo: https://github.com/codespell-project/codespell

    rev: v2.3.0

    hooks:

      - id: codespell

        additional\_dependencies:

          - tomli

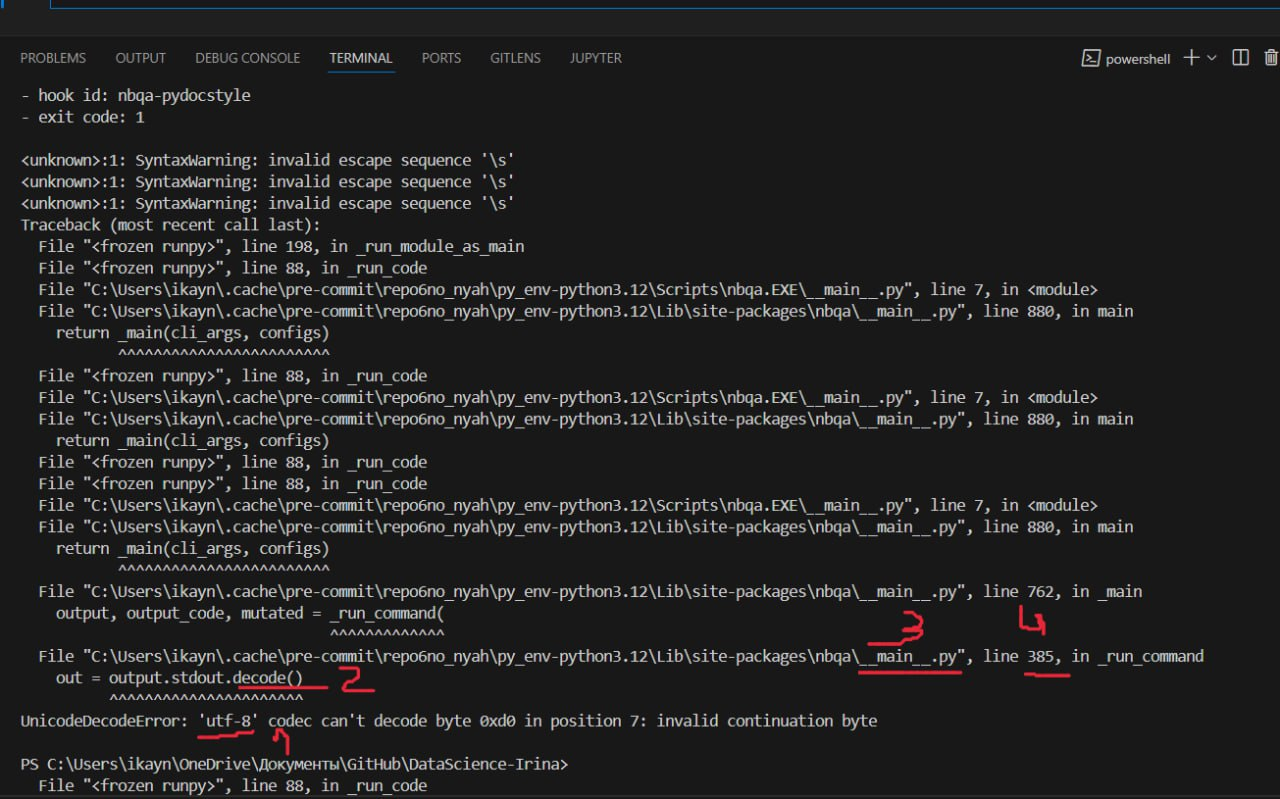
        args:

          - --ignore-regex=^\s\*"image\/(jpeg|png|gif|bmp|tiff)":\s.\*

          - --ignore-regex=[A-Za-z0-9+/]{100,}

          - --skip=\*.js,\*.html,\*.css,\*.svg",\*.json,\*.png,\*.jpg,\*.yml,\*.yaml

## Фикс ошибки pylint UnicodeDecodeError



1 - параметр, который требует функция

2 - место куда нужно вставить параметр

3 - файл с ошибкой

4 - строка с ошибкой

Алгоритм:

1. копируем путь до файла с п.3
2. ищем номер строки из п.4
3. копируем параметр из п.1
4. вставляем вместо в п.2. Вторым параметром добавляем ‘ignore’

Вместо

out = output.stdout.decode()

Должно быть

out = output.stdout.decode('utf-8', "ignore")

# Плагины VSCode

Подборка от Senatorov

<https://github.com/SENATOROVAI/intro/blob/main/README.md>

ms-python.pylint

ms-python.flake8

ms-python.mypy-type-checker

matangover.mypy

ms-pyright.pyright

ms-python.black-formatter

njpwerner.autodocstring

njqdev.vscode-python-typehint

KevinRose.vsc-python-indent

mintlify.document

streetsidesoftware.code-spell-checker

eamodio.gitlens

ms-toolsai.jupyter

ms-vsliveshare.vsliveshare

njqdev.vscode-python-typehint

EricSia.pythonsnippets3

congyiwu.vscode-jupytext

streetsidesoftware.code-spell-checker-russian