**Оглавление**

[Определение Python 3](#_Toc201044961)

[Типы данных 4](#_Toc201044962)

[Мутабельность и иммутабельность 6](#_Toc201044963)

[Параметры по умолчанию 8](#_Toc201044964)

[Переменные 9](#_Toc201044965)

[Описание типов данных 10](#_Toc201044966)

[Числа 10](#_Toc201044967)

[Строки 11](#_Toc201044968)

[Списки 14](#_Toc201044969)

[Кортежи 16](#_Toc201044970)

[Множества 17](#_Toc201044971)

[Словари 18](#_Toc201044972)

[Объект 21](#_Toc201044973)

[Диапазоны 22](#_Toc201044974)

[Работа с файлами 24](#_Toc201044975)

[Операторы и функции 27](#_Toc201044976)

[Операторы сравнения 27](#_Toc201044977)

[Логические операторы 27](#_Toc201044978)

[Операторы if, elif и else 28](#_Toc201044979)

[Циклы 29](#_Toc201044980)

[for 29](#_Toc201044981)

[while 31](#_Toc201044982)

[enumerate 32](#_Toc201044983)

[Генераторы списков List comprehension 33](#_Toc201044984)

[zip 35](#_Toc201044985)

[Распаковка с zip \* 35](#_Toc201044986)

[in 36](#_Toc201044987)

[Математические функции 37](#_Toc201044988)

[Ввод данных 38](#_Toc201044989)

[Импорты / экспорты 39](#_Toc201044990)

[Импорты 39](#_Toc201044991)

[Экспорты 41](#_Toc201044992)

[Перезагрузка модуля reload 42](#_Toc201044993)

[Пакеты и подмодули 44](#_Toc201044994)

[Функции 45](#_Toc201044995)

[\*args 46](#_Toc201044996)

[\*\*kwargs 47](#_Toc201044997)

[map 48](#_Toc201044998)

[filter 49](#_Toc201044999)

[Lambda-выражения 50](#_Toc201045000)

[Область видимости Scope. LEGB 51](#_Toc201045001)

[global и nonlocal 52](#_Toc201045002)

[Объектно-Ориентированное Программирование ООП 54](#_Toc201045003)

[Классы 54](#_Toc201045004)

[Атрибуты класса 55](#_Toc201045005)

[Цепочка атрибутов 56](#_Toc201045006)

[Наследование 58](#_Toc201045007)

[Полиморфизм 59](#_Toc201045008)

[Абстрактные классы 61](#_Toc201045009)

[Специальные методы классов 63](#_Toc201045010)

[PiPY и pip 65](#_Toc201045011)

[pip 65](#_Toc201045012)

[Виртуальное окружение 66](#_Toc201045013)

[Создание пакетов 68](#_Toc201045014)

[\_\_name\_\_ и \_\_main\_\_ 69](#_Toc201045015)

[Обработка ошибок 71](#_Toc201045016)

[try, except, else, finally 71](#_Toc201045017)

[Несколько except 71](#_Toc201045018)

[Тестирование 73](#_Toc201045019)

[Линтеры 73](#_Toc201045020)

[unittest 74](#_Toc201045021)

[Декораторы 75](#_Toc201045022)

# 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

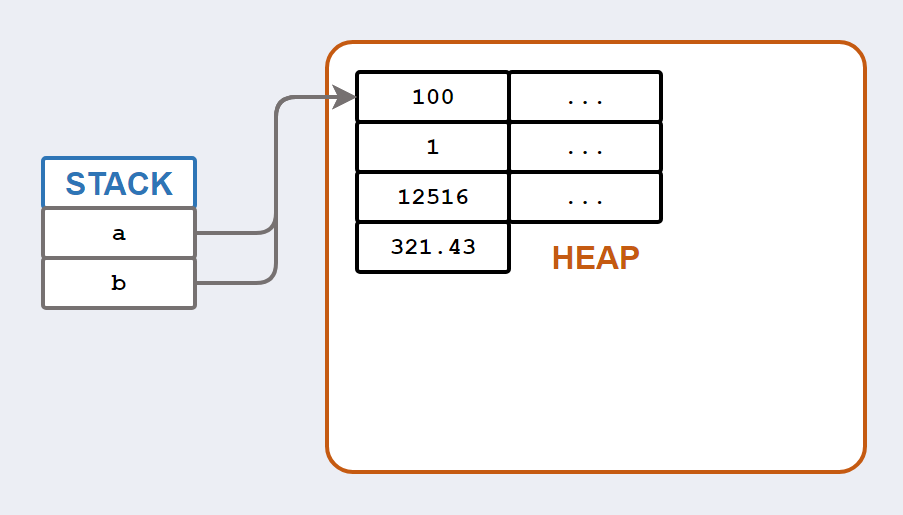
    print(*f*"a: id({id(a)})")

    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**. 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

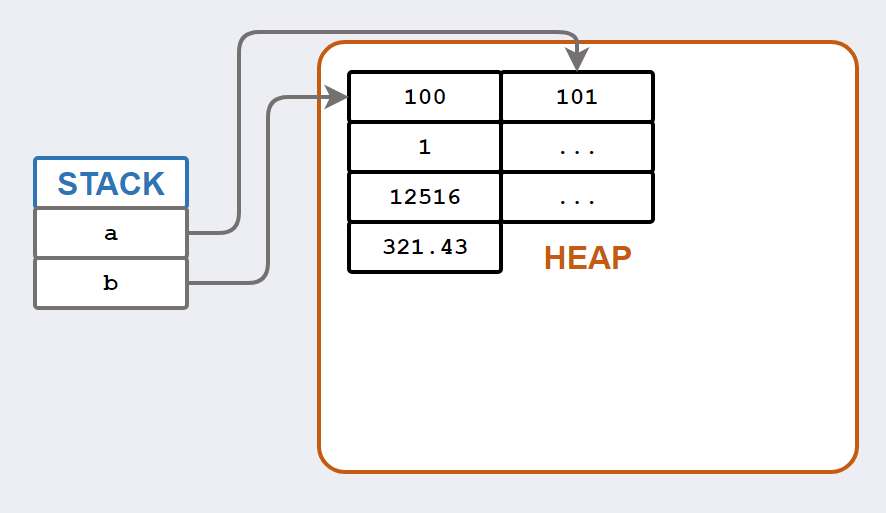
    print(*f*"a: id({id(a)})")

test\_heap\_function()

*# a: id(140088361431424)*

*# b: id(140088361431424)*

*# a: id(140088361431456)*

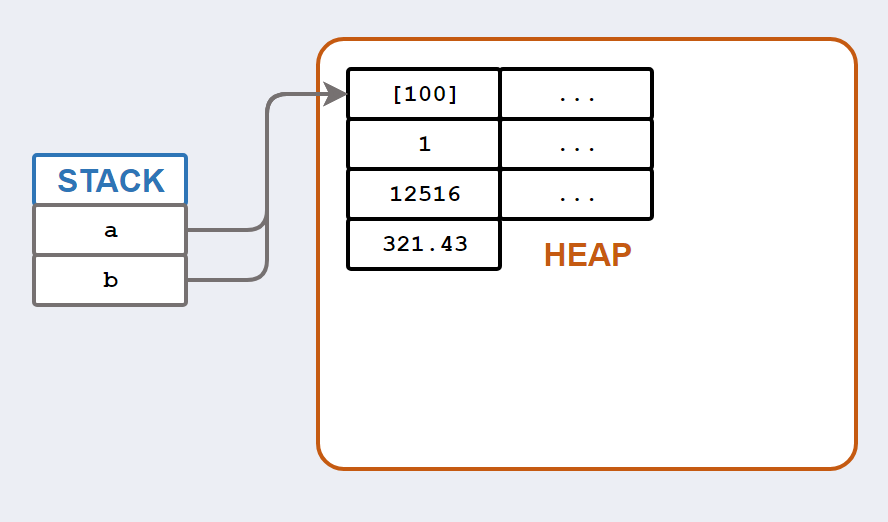


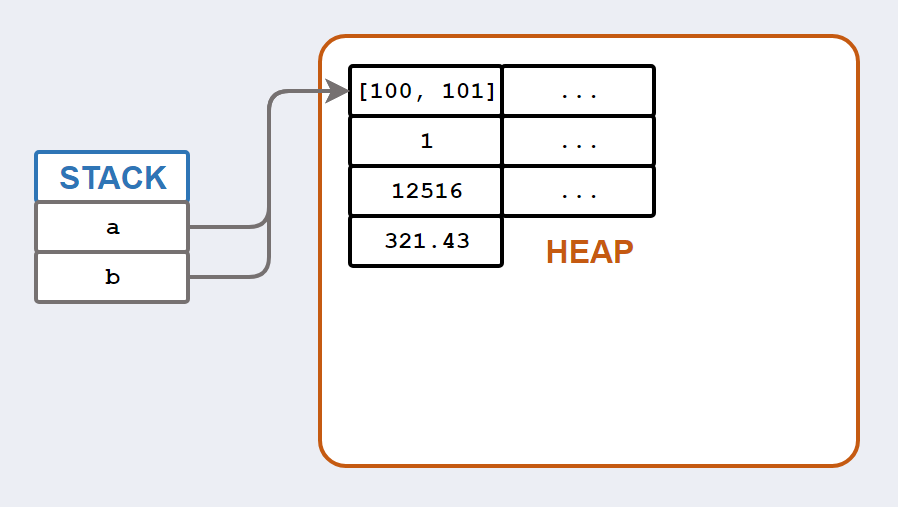
При работе с изменяемыми типами данных происходит изменение имеющегося объекта (списка), например, в него будет добавлена ссылка на новый объект (101)

list\_a = [100]

list\_b = list\_a

list\_a.append(101)

 =====>



## 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)

# 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.

**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 |  |
| **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* |

**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* |

## 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 |
| **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** | my\_list = ["one", "two", "three"]  new\_list = ["four", "five"]  print(my\_list + new\_list)  *# ['one', 'two', 'three', 'four', 'five']* |
| **append**  Добавление элемента в конец | my\_list = ["one", "two", "three"]  my\_list.append("four")  print(my\_list)  *# ['one', 'two', 'three', 'four']* |
| **pop**  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']* |
| **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]* |
| **\***  Умножение списка | *list* = [0, 1]  list\_x3 = *list* \* 3  print(*list*)  *# [0, 1]*  print(list\_x3)  *# [0, 1, 0, 1, 0, 1]* |

## 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.

## 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** | 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** | 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 набор кортежей | 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)]* |

## 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

**Методы для работы с файлами**

|  |  |
| --- | --- |
| **open**  Открыть файл  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  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")  print(myfile) |
| **close**  Закрыть файл | 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() |
| **with**  Автоматически закрывает ресурсы (например, файлы) после завершения блока | Например, при работе с соединениями с БД, сетевыми подключениями. Или чтобы не вызывать close вручную:  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.* |
| **Чтение файла** | |
| **read**  Прочитать 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")  myfile.read()  *# 'Первая строка\nВторая строка\nТретья строка\n'*  myfile = open("myfile.txt", *encoding*="utf-8")  contents = myfile.read()  print(contents)  *# Моя текстовая информация* |
| **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* |

**Режимы open**

Открыть файл

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.

# 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") |

## 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* |

## 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*

## zip

**Функция zip()** — это удобный инструмент для объединения нескольких итерируемых объектов (например, списков, кортежей) в один итератор, который возвращает пары (или 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')]*

### 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)*

Можно передавать список списков, кортеж список и прочие комбинации итераторов

## 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'>*

# 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()

# Функции

*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)

## \*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

**\*\*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]*

# 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*

## Цепочка атрибутов

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

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**

*class* Player:

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

*self*.name = name

*self*.balance = 100

*class* Dealer(*Player*):

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

*super*().\_\_init\_\_("Дилер")

## 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'*

## Специальные методы классов

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

Метод для строкового представления объекта

*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.*

# 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 |
| **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*

python -m venv .venv

*# Активация*

source .venv/bin/activate

*# Установка зависимостей*

pip install flask requests

*# Сохраняем зависимости*

pip freeze > requirements.txt

## 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)

## \_\_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

# Обработка ошибок

## 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>

# Тестирование

## Линтеры

Тестирует стиль кода и форматирование

* pylint
* pyflakes
* pep8

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

**pylint**

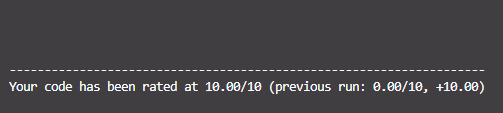
Запуск

**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"

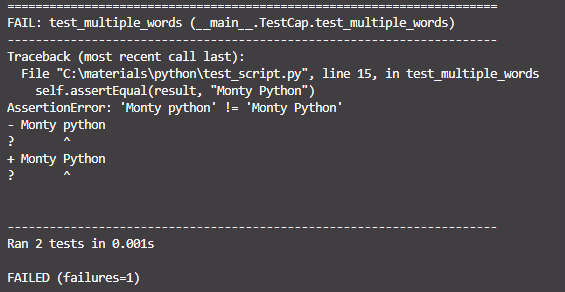
        result = my\_script.cap\_text(text)

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

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

    unittest.main()

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



# Декораторы

**Декоратор** – это функция, которая принимает другую функцию (или класс) и возвращает новую функцию (или объект), добавляя к ней какое-то поведение.

Декораторы позволяют изменять или расширять поведение функций или классов без изменения их исходного кода.

Работа декораторов основана на механизме замыкания и функциях первого порядка

## Декораторы функций

В примере ниже *new\_decorator – это декоратор* в полном смысле

*def* new\_decorator(*original\_func*):

*def* wrap\_func():

        print("Некоторый код, выполняемый до функции original\_func")

        original\_func()

        print("Некоторый код, выполняемый до функции original\_func")

    return wrap\_func

*def* func\_needs\_decorator():

    print("Эта функция нуждает в декораторе")

decorated\_func = new\_decorator(func\_needs\_decorator)

decorated\_func()

*# Некоторый код, выполняемый до функции original\_func*

*# Эта функция нуждает в декораторе*

*# Некоторый код, выполняемый до функции original\_func*

Но в Python для декораторов предусмотрен особый синтаксис с использованием **@**

Пример ниже равносилен примеру выше:

*def* new\_decorator(*original\_func*):

*def* wrap\_func():

        print("Некоторый код, выполняемый до функции original\_func")

        original\_func()

        print("Некоторый код, выполняемый до функции original\_func")

    return wrap\_func

@new\_decorator

*def* func\_needs\_decorator\_with\_decorator():

    print("Эта функция нуждает в декораторе")

func\_needs\_decorator\_with\_decorator()

*# Некоторый код, выполняемый до функции original\_func*

*# Эта функция нуждает в декораторе*

*# Некоторый код, выполняемый до функции original\_func*

**Пример: декоратор для измерения времени**

import time

from functools import wraps

*def* timer(*func*):

    @wraps(func)

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

        start\_time = time.time()

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

        duration = time.time() - start\_time

        print(*f*"Функция '{func.\_\_name\_\_}' выполнилась за {duration*:.4f*} секунд")

        return result

    return wrapper

@timer

*def* calculate\_sum(*n*):

    return sum(range(n))

calculated\_sum = calculate\_sum(

    10\_000\_000

)  *# Функция 'calculate\_sum' выполнилась за 0.1935 секунд*

print(calculated\_sum)  *# 49999995000000*

В этом примере декоратор **@wraps** сохраняет оригинальные метаданные оборачиваемой функции

*# с декоратором @wraps*

print(calculate\_sum)  *# <function calculate\_sum at 0x000001C1A41EAB60>*

*# без декоратора @wraps*

print(calculate\_sum)  *# <function timer.<locals>.wrapper at 0x000001A9AC21AB60>*

## Декораторы над классами

**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*

## 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. |

# 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*

# 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