



# РАБОТА С ОСНОВНЫМИ СТРУКТУРАМИ ДАННЫХ В PYTHON

# ПОЛЕЗНЫЕ ССЫЛКИ И ИСТОЧНИКИ

- <https://pymanual.github.io/>
- <https://docs.python.org/3/index.html> (включить переводчик)
- Уэс Маккинни Python и анализ данных: Первичная обработка данных с применением pandas, NumPy и Jupiter / пер. с англ. А. А. Слинкина. 3-е изд. — М.: МК Пресс, 2023. — 536 с.: ил.
- Бейдер Дэн, Эймос Дэвид, Яблонски Джоанна, Хейслер Флетчер Знакомство с Python. — СПб.: Питер, 2023. — 512 с.: ил. — (Серия «Библиотека программиста»).

# СЕМАНТИКА ЯЗЫКА

В Python для структурирования кода используются пробелы (или знаки табуляции):

```
for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

Рекомендуется использовать 4 пробела в качестве величины отступа по умолчанию и настроить редактор так, что бы он заменял знаки табуляции четырьмя пробелами

Предложения в Python не обязаны завершаться точкой с запятой. Но ее можно использовать, чтобы отделить друг от друга предложения, находящиеся в одной строке

```
a = 5; b = 6; c = 7
```

Впрочем, писать несколько предложений в одной строке не рекомендуется, потому что код из-за этого становится труднее читать.

# ВСЁ ЯВЛЯЕТСЯ ОБЪЕКТОМ

- Все числа, строки, структуры данных, функции, классы, модули называются *объектами* Python
- С каждым объектом ассоциирован *тип* (например, целое число, строка или функция) и *внутренние данные*.

## КОММЕНТАРИИ

Комментарии выделяются знаком решетки #. Иногда желательно исключить какие-то блоки кода, не удаляя их.

```
results = []
for line in file_handle:
    # пока оставляем пустые строки
    # if len(line) == 0:
    #     continue
    results.append(line.replace("foo", "bar"))
```

# ВЫЗОВ ФУНКЦИИ И МЕТОДА ОБЪЕКТА

После имени функции ставятся круглые скобки, внутри которых может быть нуль или более параметров. Возвращенное значение может быть присвоено переменной:

```
result = f(x, y, z)  
g()
```

Почти со всеми объектами в Python ассоциированы функции, которые имеют доступ к внутреннему состоянию объекта и называются методами. Синтаксически вызов методов выглядит так

```
obj.some_method(x, y, z)
```

Функции могут принимать позиционные и именованные аргументы

```
result = f(a, b, c, d=5, e="foo")
```

# ПЕРЕМЕННЫЕ И ТИПЫ ДАННЫХ

Python — язык с динамической типизацией. Примеры основных ТИПОВ:

```
# Числа
age = 25                # int (целое)
price = 19.99           # float (дробное)

# Текст
name = "Анна"           # str (строка)
message = 'Привет!'     # Можно и одинарные кавычки

# Логические значения
is_active = True        # bool (True/False)
has_items = False

# Вывод на экран
print("Имя:", name, "Возраст:", age)
```

Вывод на экран:

```
# Вывод на экран
print("Имя:", name, "Возраст:", age)
```

Имя: Анна Возраст: 25

## Что важно:

- Нет необходимости объявлять тип переменной
- Регистр букв имеет значение: age не равно Age

# ПЕРЕМЕННЫЕ

Присваивание значения переменной (или имени) в Python приводит к созданию ссылки на объект, стоящий в правой части

при

```
In [8]: a = [1, 2, 3]
```

Понимать семантику ссылок в Python и знать, когда, как и почему данные копируются, особенно важно при работе с большими

наборами данных. Операцию присваивания называют также связыванием, потому что мы связываем имя с объектом. Имена переменных, которым присвоено значение, иногда называют связанными переменными.



# ПЕРЕДАЧА АРГУМЕНТОВ

Когда объекты передаются функции в качестве аргументов, создаются новые локальные переменные, ссылающиеся на исходные объекты, – копирование не производится. Если новый объект связывается с переменной внутри функции, то переменная с таким же именем в «области видимости» вне этой функции («родительской области видимости») не перезаписывается. Поэтому функция может модифицировать внутреннее содержимое изменяемого аргумента.

```
In [13]: def append_element(some_list, element):  
        ....:     some_list.append(element)
```

Тогда:

```
In [14]: data = [1, 2, 3]
```

```
In [15]: append_element(data, 4)
```

```
In [16]: data
```

```
Out[16]: [1, 2, 3, 4]
```



# ДИНАМИЧЕСКИЕ ССЫЛКИ, СТРОГИЕ ТИПЫ

С переменными в Python не связан никакой тип; достаточно выполнить присваивание, чтобы переменная стала указывать на объект

Переменные – это имена объектов в некотором пространстве имен; информация о типе хранится в самом объекте.

Знать тип объекта важно, и полезно также уметь писать функции, способные обрабатывать входные параметры различных типов. Проверить, что объект является экземпляром определенного типа, позволяет функция `isinstance`

```
In [17]: a = 5
```

```
In [18]: type(a)
```

```
Out[18]: int
```

```
In [19]: a = "foo"
```

```
In [20]: type(a)
```

```
Out[20]: str
```

```
In [21]: "5" + 5
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-21-7fe5aa79f268> in <module>  
----> 1 "5" + 5  
TypeError: can only concatenate str (not "int") to str
```

```
In [26]: a = 5
```

```
In [27]: isinstance(a, int)
```

```
Out[27]: True
```

# ОСНОВНЫЕ ОПЕРАЦИИ



# Арифметика

```
x = 10 + 5    # Сложение
y = 20 - 3    # Вычитание
z = 7 * 2     # Умножение
w = 15 / 4    # Деление (возвращает float)
q = 15 // 4   # Целочисленное деление
r = 15 % 4    # Остаток от деления
```

# Сравнение

```
print(10 > 5)  # True
print(10 == 5) # False
print(10 != 5) # True
```

# Логические операции

```
print(True and False) # False
print(True or False)  # True
print(not True)        # False
```

# УСЛОВНЫЕ КОНСТРУКЦИИ



```
# Простое условие
temperature = 23
if temperature > 30:
    print("Жарко")
elif temperature > 20:
    print("Тепло")
else:
    print("Прохладно")

# Тернарный оператор
status = "Взрослый" if age >= 18 else "Ребёнок"
```

# ЦИКЛЫ



```
# Цикл for (перебор последовательности)
fruits = ["яблоко", "банан", "апельсин"]
for fruit in fruits:
    print(fruit)

# Цикл while (пока условие истинно)
count = 0
while count < 3:
    print(f"Итерация {count}")
    count += 1
```



```
яблоко
банан
апельсин
Итерация 0
Итерация 1
Итерация 2
```

# СТРУКТУРЫ ДАННЫХ И ПОСЛЕДОВАТЕЛЬНОСТИ

Структуры данных в Python просты, но эффективны. Чтобы стать хорошим программистом на Python, необходимо овладеть ими в совершенстве.

**Кортеж, список и словарь** – наиболее часто используемые типы последовательностей.

# КОРТЕЖИ

Структуры данных в Python просты, но эффективны. Чтобы стать хорошим программистом на Python, необходимо овладеть ими в совершенстве. **Кортеж, список и словарь** – наиболее часто используемые типы последовательностей.

**Кортеж** – это одномерная неизменяемая последовательность объектов Python фиксированной длины, которую нельзя изменить после первоначального присваивания. Проще всего создать кортеж, записав последовательность значений через

```
In [2]: tup = (4, 5, 6)
```

```
In [3]: tup  
Out[3]: (4, 5, 6)
```

```
In [4]: tup = 4, 5, 6
```

```
In [5]: tup  
Out[5]: (4, 5, 6)
```

Можно без  
скобок↑

Любую последовательность или итератор можно преобразовать в кортеж с помощью функции **tuple** :

```
In [6]: tuple([4, 0, 2])  
Out[6]: (4, 0, 2)
```

```
In [7]: tup = tuple('string')
```

```
In [8]: tup  
Out[8]: ('s', 't', 'r', 'i', 'n', 'g')
```

Обращение к  
элементам

```
In [9]: tup[0]  
Out[9]: 's'
```

Кортеж кортежей:

```
In [10]: nested_tup = (4, 5, 6), (7, 8)
```

```
In [11]: nested_tup  
Out[11]: ((4, 5, 6), (7, 8))
```

```
In [12]: nested_tup[0]  
Out[12]: (4, 5, 6)
```

```
In [13]: nested_tup[1]  
Out[13]: (7, 8)
```

# КОРТЕЖИ

ВНИМАНИЕ! Кортеж после создания изменить (т. е. записать что-то другое в существующую позицию) невозможно!

```
In [14]: tup = tuple(['foo', [1, 2], True])
```

```
In [15]: tup[2] = False
```

Кортежи можно

В составе кортежа могут быть ИЗМЕНЯЕМЫЕ ОБЪЕКТЫ, например, если является списком, то его можно модифицировать на месте

Конкатенировать с помощью оператора +, получая в результате более длинный кортеж

```
In [16]: tup[1].append(3)
```

```
In [17]: tup  
Out[17]: ('foo', [1, 2, 3], True)
```

```
In [18]: (4, None, 'foo') + (6, 0) + ('bar',)  
Out[18]: (4, None, 'foo', 6, 0, 'bar')
```

Умножение кортежа на целое число, как и в случае списка, приводит к конкатенации нескольких копий кортежа, при этом копируются не сами объекты, а только ссылки на них.

```
In [19]: ('foo', 'bar') * 4  
Out[19]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```



# КОРТЕЖИ

**Распаковать значение** - присвоить значение выражению, которое похоже на кортеж и состоит из нескольких переменных. При этом интерпретатор пытается распаковать значение в правой части оператора присвоения.

Распаковка  
вложенного кортежа

```
In [23]: tup = 4, 5, (6, 7)
```

```
In [24]: a, b, (c, d) = tup
```

```
In [25]: d  
Out[25]: 7
```

Распаковка позволяет  
обмен значений

```
In [26]: a, b = 1, 2
```

```
In [27]: a  
Out[27]: 1
```

```
In [28]: b  
Out[28]: 2
```

```
In [29]: b, a = a, b
```

```
In [30]: a  
Out[30]: 2
```

```
In [31]: b  
Out[31]: 1
```

```
In [20]: tup = (4, 5, 6)
```

```
In [21]: a, b, c = tup
```

```
In [22]: b  
Out[22]: 5
```

Обход последовательности  
кортежей или списков:

```
In [32]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
In [33]: for a, b, c in seq:  
.....:     print(f'a={a}, b={b}, c={c}')  
a=1, b=2, c=3  
a=4, b=5, c=6  
a=7, b=8, c=9
```

# КОРТЕЖИ

**Возврат нескольких значений из функции** - когда требуется «отщепить» несколько элементов из начала кортежа.

Для этого применяется специальный синтаксис **\*rest**, используемый также в сигнатурах функций, чтобы обозначить сколь угодно длинный список

```
In [34]: values = 1, 2, 3, 4, 5
```

```
In [35]: a, b, *rest = values
```

```
In [36]: a
```

```
Out[36]: 1
```

```
In [37]: b
```

```
Out[37]: 2
```

```
In [38]: rest
```

```
Out[38]: [3, 4, 5]
```

ПРИМЕР:

Часть **rest** иногда требуется отбросить; в самом имени **rest** нет ничего специального, оно может быть любым.

Многие программисты используют для обозначения ненужных переменных знак подчеркивания (**\_**)

```
In [39]: a, b, *_ = values
```

Методы кортежа: размер, содержимое кортежа нельзя модифицировать.  
Наиболее полезен метод **count** (имеется также у списков), возвращающий количество вхождений значения:

```
In [40]: a = (1, 2, 2, 2, 3, 4, 2)
```

```
In [41]: a.count(2)
```

```
Out[41]: 4
```

# КОРТЕЖИ



```
# Создание:
coordinates = (10, 20) # Неизменяемый тип
single_element = (42,) # Кортеж из одного элемента (запятая обязательна!)
colors = tuple(["красный", "зелёный", "синий"])

# Распаковка
x, y = coordinates # x = 10, y = 20

# Неизменяемость
try:
    coordinates[0] = 5 # Ошибка: TypeError
except TypeError as e:
    print("Кортежи нельзя изменять!")

# Возврат нескольких значений из функции:
def get_stats(data):
    return min(data), max(data), sum(data)/len(data)
```



Кортежи нельзя изменять!

# СПИСКИ

В отличие от кортежей, списки имеют переменную длину, а их содержимое можно модифицировать. Список определяется с помощью квадратных скобок [] или конструктора типа **list**:

```
In [42]: a_list = [2, 3, 7, None]

In [43]: tup = ("foo", "bar", "baz")

In [44]: b_list = list(tup)
```

```
In [45]: b_list
Out[45]: ['foo', 'bar', 'baz']

In [46]: b_list[1] = "peekaboo"

In [47]: b_list
Out[47]: ['foo', 'peekaboo', 'baz']
```

Функция `list` часто используется при обработке данных, чтобы материализовать итератор или генераторное выражение:

```
In [48]: gen = range(10)

In [49]: gen
Out[49]: range(0, 10)

In [50]: list(gen)
Out[50]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# СПИСКИ - ДОБАВЛЕНИЕ И УДАЛЕНИЕ ЭЛЕМЕНТОВ

Для добавления элемента в конец списка служит метод **append**:

```
In [51]: b_list.append("dwarf")  
  
In [52]: b_list  
Out[52]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

Метод **insert** - вставить элемент в указанную позицию списка:

```
In [53]: b_list.insert(1, "red")  
  
In [54]: b_list  
Out[54]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

**ВНИМАНИЕ!** Индекс позиции вставки должен принадлежать диапазону от 0 до длины списка включительно.

Метод **pop** удаляет из списка элемент находившийся в указанной позиции, и возвращает его:

```
In [55]: b_list.pop(2)  
Out[55]: 'peekaboo'  
  
In [56]: b_list  
Out[56]: ['foo', 'red', 'baz', 'dwarf']
```

Метод **remove** находит и удаляет из списка первый элемент с указанным значением:

```
In [57]: b_list.append("foo")  
  
In [58]: b_list  
Out[58]: ['foo', 'red', 'baz', 'dwarf', 'foo']  
  
In [59]: b_list.remove("foo")  
  
In [60]: b_list  
Out[60]: ['red', 'baz', 'dwarf', 'foo']
```

Проверить, содержит ли список некоторое значение - используем

ключевое слово **in** :

```
In [61]: "dwarf" in b_list  
Out[61]: True
```

# СПИСКИ - КОНКАТЕНАЦИЯ И КОМБИНИРОВАНИЕ СПИСКОВ

Операция сложения конкатенирует списки:

```
In [63]: [4, None, "foo"] + [7, 8, (2, 3)]  
Out[63]: [4, None, 'foo', 7, 8, (2, 3)]
```

Добавить в его конец несколько элементов – метод **extend**:

```
In [64]: x = [4, None, "foo"]  
  
In [65]: x.extend([7, 8, (2, 3)])  
  
In [66]: x  
Out[66]: [4, None, 'foo', 7, 8, (2, 3)]
```

**ВНИМАНИЕ!** конкатенация – сравнительно дорогая операция, потому что нужно создать новый список и скопировать в него все объекты.

Обычно предпочтительнее использовать **extend** для добавления элементов в существующий список, особенно если строится длинный список.

```
everything = []  
for chunk in list_of_lists:  
    everything.extend(chunk)
```

**Быстрее,  
чем**

```
everything = []  
for chunk in list_of_lists:  
    everything = everything + chunk
```



# СПИСКИ - СОРТИРОВКА

отсортировать на месте  
(без создания нового объекта),  
вызвав его метод **sort** :

```
In [67]: a = [7, 2, 5, 1, 3]
```

```
In [68]: a.sort()
```

```
In [69]: a
```

```
Out[69]: [1, 2, 3, 5, 7]
```

Возможность передать ключ сортировки,  
т. е. функцию, порождающую значение,  
по которому должны сортироваться объекты:

```
In [70]: b = ["saw", "small", "He", "foxes", "six"]
```

```
In [71]: b.sort(key=len)
```

```
In [72]: b
```

```
Out[72]: ['He', 'saw', 'six', 'small', 'foxes']
```



# СПИСКИ - ВЫРЕЗАНИЕ

Из большинства последовательностей можно вырезать участки с помощью нотации среза, которая в простейшей форме сводится к передаче пары **start:stop** оператору доступа по индексу []:

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]
```

```
In [74]: seq[1:5]  
Out[74]: [2, 3, 7, 5]
```

Срезу также можно присваивать последовательность

```
In [75]: seq[3:5] = [6, 3]
```

```
In [76]: seq  
Out[76]: [7, 2, 3, 6, 3, 6, 0, 1]
```

Если индекс в срезе отрицателен, то отсчет ведется от конца последовательности:

```
In [79]: seq[-4:]  
Out[79]: [3, 6, 0, 1]
```

```
In [80]: seq[-6:-2]  
Out[80]: [3, 6, 3, 6]
```

Элемент с индексом **start** включается в срез, элемент с индексом **stop** не включается, поэтому количество элементов в результате равно **stop - start**.

Любой член пары – как start, так и stop – можно опустить, тогда по умолчанию подразумевается начало и конец последовательности соответственно.

```
In [77]: seq[:5]  
Out[77]: [7, 2, 3, 6, 3]
```

```
In [78]: seq[3:]  
Out[78]: [6, 3, 6, 0, 1]
```

Допускается и вторая запятая, после которой можно указать шаг, например взять каждый второй элемент

```
In [81]: seq[::2]  
Out[81]: [7, 3, 3, 0]
```

Если задать шаг -1, то список или кортеж будет инвертирован:

```
In [82]: seq[::-1]  
Out[82]: [1, 0, 6, 3, 6, 3, 2, 7]
```

# СПИСКИ



```
numbers = [1, 2, 3, 4, 5]
```

```
# Основные операции
```

```
numbers.append(6)      # Добавить элемент
```

```
numbers.remove(3)      # Удалить элемент
```

```
length = len(numbers)  # Длина списка
```

```
# Срезы (slice)
```

```
first_two = numbers[:2] # [1, 2]
```

```
last_three = numbers[-3:] # [4, 5, 6]
```

```
# Создание нового списка на основе старого, возведя числа в квадрат
```

```
squares = [x**2 for x in numbers]
```

```
print(squares)
```



```
[1, 4, 16, 25, 36]
```

# ПОЛЕЗНЫЕ ВСТРОЕННЫЕ ФУНКЦИИ СПИСКОВ



```
numbers = [4, 2, 9, 7]
```

```
print(len(numbers))    # 4 (длина)
print(sum(numbers))    # 22 (сумма)
print(min(numbers))    # 2 (минимум)
print(max(numbers))    # 9 (максимум)
print(sorted(numbers)) # [2, 4, 7, 9] (сортировка)
```

```
names = ["Анна", "Пётр", "Мария"]
print(", ".join(names)) # Объединение строк
```



```
4
22
2
9
[2, 4, 7, 9]
Анна, Пётр, Мария
```

# СЛОВАРИ

Словарь, или **dict** - самая важная из встроенных в Python структур данных. Он представляет собой коллекцию пар ключ-значение, в которой и ключ, и значение – объекты Python. С каждым ключом ассоциировано значение, так что значение можно извлекать, вставлять, изменять или удалять, если известен ключ.

Создать словарь - с помощью фигурных скобок {}, отделяя ключи от значений двоеточием:

```
In [83]: empty_dict = {}
```

```
In [84]: d1 = {"a": "some value", "b": [1, 2, 3, 4]}
```

```
In [85]: d1
```

```
Out[85]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

Для доступа к элементам, вставки и присваивания применяется синтаксис:

```
In [86]: d1[7] = "an integer"
```

```
In [87]: d1
```

```
Out[87]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

```
In [88]: d1["b"]
```

```
Out[88]: [1, 2, 3, 4]
```

Проверка наличия ключа в словаре :

```
In [89]: "b" in d1
```

```
Out[89]: True
```

# СЛОВАРИ

Для удаления ключа можно использовать либо ключевое слово **del**, либо метод **pop** (который не только удаляет ключ, но и возвращает ассоциированное с ним значение):

```
In [90]: d1[5] = "some value"
```

```
In [91]: d1
```

```
Out[91]:
```

```
{'a': 'some value',  
 'b': [1, 2, 3, 4],  
 7: 'an integer',  
 5: 'some value'}
```

```
In [92]: d1["dummy"] = "another value"
```

```
In [93]: d1
```

```
Out[93]:
```

```
{'a': 'some value',  
 'b': [1, 2, 3, 4],  
 7: 'an integer',  
 5: 'some value',  
 'dummy': 'another value'}
```

```
In [94]: del d1[5]
```

```
In [95]: d1
```

```
Out[95]:
```

```
{'a': 'some value',  
 'b': [1, 2, 3, 4],  
 7: 'an integer',  
 'dummy': 'another value'}
```

```
In [96]: ret = d1.pop("dummy")
```

```
In [97]: ret
```

```
Out[97]: 'another value'
```

```
In [98]: d1
```

```
Out[98]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

# СЛОВАРИ

Методы **keys** и **values** возвращают соответственно список ключей и список значений. Хотя точный порядок пар ключ-значение не определен, эти методы возвращают ключи и значения в одном и том же порядке:

```
In [99]: list(d1.keys())
Out[99]: ['a', 'b', 7]

In [100]: list(d1.values())
Out[100]: ['some value', [1, 2, 3, 4], 'an integer']
```

Одновременно обойти ключи и значения - метод **items**, который отдает 2-кортежи, состоящие из ключа и значения:

```
In [101]: list(d1.items())
Out[101]: [('a', 'some value'), ('b', [1, 2, 3, 4]), (7, 'an integer')]
```

Два словаря можно объединить в один методом **update**:

```
In [102]: d1.update({"b": "foo", "c": 12})

In [103]: d1
Out[103]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

## ВАЖНО!

Метод **update** модифицирует словарь на месте, т. е. старые значения существующих ключей, переданных update, **стираются**.



# СЛОВАРИ - СОЗДАНИЕ СЛОВАРЯ ИЗ ПОСЛЕДОВАТЕЛЬНОСТЕЙ, ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

Нередко бывает, что имеются две последовательности, которые естественно рассматривать как ключи и соответствующие им значения, а значит, требуется построить из них словарь.

Первый способ :

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

Методы словаря **get** и **pop** могут принимать значение, возвращаемое по умолчанию :

```
value = some_dict.get(key, default_value)
```

!!! Метод **get** по умолчанию возвращает None, если ключ не найден, тогда как **pop** в этом случае возбуждает исключение.

Второй: словарь – это коллекция 2-кортежей, тогда функция **dict** принимает список 2-кортежей:

```
In [104]: tuples = zip(range(5), reversed(range(5)))

In [105]: tuples
Out[105]: <zip at 0x7fefe4553a00>

In [106]: mapping = dict(tuples)

In [107]: mapping
Out[107]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Метод **setdefault**

```
In [112]: by_letter = {}

In [113]: for word in words:
.....:     letter = word[0]
.....:     by_letter.setdefault(letter, []).append(word)
.....:

In [114]: by_letter
Out[114]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

В стандартном модуле `collections` есть полезный класс `defaultdict`. Его конструктору передается тип или функция, генерирующая значение по умолчанию для каждой записи в словаре:

```
In [115]: from collections import defaultdict

In [116]: by_letter = defaultdict(list)

In [117]: for word in words:
.....:     by_letter[word[0]].append(word)
```



# СЛОВАРИ - ДОПУСТИМЫЕ ТИПЫ КЛЮЧЕЙ СЛОВАРЯ

Значениями словаря могут быть произвольные объекты Python, но ключами должны быть неизменяемые объекты, например скалярные типы (int, float, строка) или кортежи (причем все объекты кортежа тоже должны быть неизменяемыми). Технически это свойство называется хешируемостью.

Проверить, является ли объект хешируемым (и, стало быть, может быть ключом словаря), позволяет функция **hash**:  
Хеш-значения, которые возвращает функция **hash**, вообще говоря, зависят от версии Python.

Чтобы использовать список в качестве ключа, достаточно преобразовать его в кортеж, который допускает хеширование, если это верно для его элементов:

```
In [118]: hash("string")
Out[118]: 3634226001988967898

In [119]: hash((1, 2, (2, 3)))
Out[119]: -9209053662355515447

In [120]: hash((1, 2, [2, 3])) # ошибка, потому что списки изменяемы
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-120-473c35a62c0b> in <module>
----> 1 hash((1, 2, [2, 3])) # ошибка, потому что списки изменяемы
TypeError: unhashable type: 'list'
```

```
In [121]: d = {}

In [122]: d[tuple([1, 2, 3])] = 5

In [123]: d
Out[123]: {(1, 2, 3): 5}
```

# СЛОВАРИ

```
person = {  
    "name": "Иван",  
    "age": 30,  
    "city": "Москва"  
}  
  
# Доступ к элементам  
print(person["name"]) # Иван  
print(person.get("job", "не указано")) # Безопасный доступ  
print("") # Пустая строка  
  
# Изменение  
person["age"] = 31  
person["job"] = "инженер" # Добавление  
  
# Перебор  
for key, value in person.items():  
    print(f"{key}: {value}")
```

```
Иван  
не указано  
  
name: Иван  
age: 31  
city: Москва  
job: инженер
```

# МНОЖЕСТВА

Множество – это неупорядоченная коллекция уникальных элементов.

Создание множества с помощью функции `set` :

```
In [124]: set([2, 2, 2, 1, 3, 3])  
Out[124]: {1, 2, 3}
```

Создание множества - задав множество-литерал в фигурных скобках:

```
In [125]: {2, 2, 2, 1, 3, 3}  
Out[125]: {1, 2, 3}
```

Множества поддерживают теоретико-множественные операции: объединение, пересечение, разность и симметрическую разность.

Их объединение – это множество, содержащее неповторяющиеся элементы, встречающиеся хотя бы в одном множестве. Вычислить его можно с помощью метода **union** или бинарного оператора `|` :

```
In [128]: a.union(b)  
Out[128]: {1, 2, 3, 4, 5, 6, 7, 8}  
  
In [129]: a | b  
Out[129]: {1, 2, 3, 4, 5, 6, 7, 8}
```

Пересечение множеств содержит элементы, встречающиеся в обоих множествах. Вычислить его можно с помощью метода **intersection** или

```
In [130]: a.intersection(b)  
Out[130]: {3, 4, 5}  
  
In [131]: a & b  
Out[131]: {3, 4, 5}
```

# МНОЖЕСТВА

## ОПЕРАЦИИ НАД МНОЖЕСТВАМИ В PYTHON

Функция	Альтернативный синтаксис	Описание
<code>a.add(x)</code>	Нет	Добавить элемент <code>x</code> в множество <code>a</code>
<code>a.clear()</code>	Нет	Опустошить множество, удалив из него все элементы
<code>a.remove(x)</code>	Нет	Удалить элемент <code>x</code> из множества <code>a</code>
<code>a.pop()</code>	Нет	Удалить какой-то элемент <code>x</code> из множества <code>a</code> и возбудить исключение <code>KeyError</code> , если множество пусто
<code>a.union(b)</code>	<code>a   b</code>	Найти все уникальные элементы, входящие либо в <code>a</code> , либо в <code>b</code>
<code>a.update(b)</code>	<code>a  = b</code>	Присвоить <code>a</code> объединение элементов <code>a</code> и <code>b</code>
<code>a.intersection(b)</code>	<code>a &amp; b</code>	Найти все элементы, входящие и в <code>a</code> , и в <code>b</code>
<code>a.intersection_update(b)</code>	<code>a &amp;= b</code>	Присвоить <code>a</code> пересечение элементов <code>a</code> и <code>b</code>
<code>a.difference(b)</code>	<code>a - b</code>	Найти элементы, входящие в <code>a</code> , но не входящие в <code>b</code>
<code>a.difference_update(b)</code>	<code>a -= b</code>	Записать в <code>a</code> элементы, которые входят в <code>a</code> , но не входят в <code>b</code>
<code>a.symmetric_difference(b)</code>	<code>a ^ b</code>	Найти элементы, входящие либо в <code>a</code> , либо в <code>b</code> , но не в <code>a</code> и <code>b</code> одновременно
<code>a.symmetric_difference_update(b)</code>	<code>a ^= b</code>	Записать в <code>a</code> элементы, которые входят либо в <code>a</code> , либо в <code>b</code> , но не в <code>a</code> и <code>b</code> одновременно
<code>a.issubset(b)</code>	Нет	<code>True</code> , если все элементы <code>a</code> входят также и в <code>b</code>
<code>a.issuperset(b)</code>	Нет	<code>True</code> , если все элементы <code>b</code> входят также и в <code>a</code>
<code>a.isdisjoint(b)</code>	Нет	<code>True</code> , если у <code>a</code> и <code>b</code> нет ни одного общего элемента

У всех логических операций над множествами имеются варианты с обновлением на месте, которые позволяют заменить содержимое множества в левой части результатом операции. Для очень больших множеств это может оказаться эффективнее:

```
In [132]: c = a.copy()
```

```
In [133]: c |= b
```

```
In [134]: c
Out[134]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [135]: d = a.copy()
```

```
In [136]: d &= b
```

```
In [137]: d
Out[137]: {3, 4, 5}
```

# МНОЖЕСТВА

## ОПЕРАЦИИ НАД МНОЖЕСТВАМИ В PYTHON

Элементы множества должны быть неизменяемыми и хешируемыми (т. е. вызов **hash** для значения не должен возбуждать исключения).

Чтобы сохранить в множестве элементы, подобные списку, необходимо сначала преобразовать их в кортеж:

```
In [138]: my_data = [1, 2, 3, 4]
In [139]: my_set = {tuple(my_data)}
In [140]: my_set
Out[140]: {(1, 2, 3, 4)}
```

Проверить, является ли множество подмножеством (содержится в) или надмножеством (содержит) другого множества:

```
In [141]: a_set = {1, 2, 3, 4, 5}
In [142]: {1, 2, 3}.issubset(a_set)
Out[142]: True
In [143]: a_set.issuperset({1, 2, 3})
Out[143]: True
```

Множества называются равными, если состоят из одинаковых элементов.

```
In [144]: {1, 2, 3} == {3, 2, 1}
Out[144]: True
```



# ВСТРОЕННЫЕ ФУНКЦИИ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

У последовательностей в Python есть несколько полезных функций, которые следует знать и применять при любой возможности.

**enumerate** - возвращает последовательность кортежей (i, value) :

```
for index, value in enumerate(collection):  
    # что-то сделать с value
```

**sorted** - возвращает новый отсортированный список, построенный из элементов произвольной последовательности: :

```
In [145]: sorted([7, 1, 2, 6, 0, 3, 2])  
Out[145]: [0, 1, 2, 2, 3, 6, 7]  
  
In [146]: sorted("horse race")  
Out[146]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

Функция **zip** «сшивает» элементы нескольких списков, кортежей или других последовательностей в пары, создавая список кортежей:

```
In [147]: seq1 = ["foo", "bar", "baz"]  
  
In [148]: seq2 = ["one", "two", "three"]  
  
In [149]: zipped = zip(seq1, seq2)  
  
In [150]: list(zipped)  
Out[150]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

**Reversed** - перебирает элементы последовательности в обратном порядке:

```
In [154]: list(reversed(range(10)))  
Out[154]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

# МНОЖЕСТВА



# Создание:

```
unique_numbers = {1, 2, 3, 3, 2} # {1, 2, 3} (дубли удаляются)
empty_set = set()                # Не используйте {} – это словарь!
```

# Операции:

```
A = {1, 2, 3}
```

```
B = {3, 4, 5}
```

```
print(A | B) # Объединение: {1, 2, 3, 4, 5}
```

```
print(A & B) # Пересечение: {3}
```

```
print(A - B) # Разность: {1, 2}
```

```
print(A ^ B) # Симметричная разность: {1, 2, 4, 5}
```

# Методы:

```
A.add(4) # Добавление элемента
```

```
A.remove(1) # Удаление (KeyError, если элемента нет)
```

```
A.discard(10) # Безопасное удаление (нет ошибки)
```

```
A.pop() # Удаляет и возвращает случайный элемент
```



# ФУНКЦИИ

**Функции** – главный и самый важный способ организации и повторного использования кода в Python. Если вам кажется, что некоторый код может использоваться более одного раза, возможно с небольшими вариациями, то имеет смысл оформить его в виде функции. Кроме того, функции могут сделать код более понятным, поскольку дают имя группе взаимосвязанных предложений.

Начинается ключевым словом  
**def**

```
In [173]: def my_function(x, y):  
.....:     return x + y
```

**return** –результат  
возвращается

По достижении строки **return** значение или выражение, следующее после **return**, передается в контекст, из которого была вызвана функция, например:

```
In [174]: my_function(1, 2)  
Out[174]: 3  
  
In [175]: result = my_function(1, 2)  
  
In [176]: result  
Out[176]: 3
```

Ничто не мешает иметь в функции несколько предложений **return**. Если достигнут конец функции, а предложение **return** не встретилось, то возвращается None.

# ФУНКЦИИ

У функции могут быть позиционные и именованные аргументы. *Именованные аргументы* обычно используются для задания значений по умолчанию и необязательных аргументов.

Функция с факультативным аргументом `z`, который по умолчанию принимает значение 1.5

```
def my_function2(x, y, z=1.5):  
    if z > 1:  
        return z * (x + y)  
    else:  
        return z / (x + y)
```

Именованные аргументы факультативны, но все позиционные аргументы должны быть заданы при вызове функции.

Передавать значения аргумента `z` можно с указанием и без указания имени, но рекомендуется все-таки имя указывать.

```
In [181]: my_function2(5, 6, z=0.7)  
Out[181]: 0.06363636363636363
```

```
In [182]: my_function2(3.14, 7, 3.5)  
Out[182]: 35.49
```

Основное ограничение состоит в том, что именованные аргументы должны находиться после всех позиционных (если таковые имеются). Сами же именованные аргументы можно задавать в любом порядке, это освобождает программиста от необходимости запоминать, в каком порядке были указаны аргументы функции в объявлении. Важно лишь помнить их имена.

# ФУНКЦИИ

Функции могут обращаться к переменным, объявленным как внутри самой функции, так и вне нее — в объемлющих (и даже глобальной) областях видимости.

Область видимости переменной в Python называют *также пространством имен*.

Любая переменная, которой присвоено значение внутри функции, по умолчанию попадает в локальное пространство имен.

Локальное пространство имен создается при вызове функции, и в него сразу же заносятся аргументы функции. По завершении функции локальное пространство имен уничтожается (хотя бывают и исключения, см. ниже раздел о замыканиях).

Присваивать значение глобальной переменной вне области видимости функции допустимо, но такие переменные должны быть объявлены с помощью ключевого слова **global** или **nonlocal**

# ФУНКЦИИ



```
# Простая функция
def greet(name):
    return f"Привет, {name}!"

print(greet("Максим"))

# Функция с параметрами по умолчанию
def power(number, exponent=2):
    return number ** exponent

print(power(3))      # 9 (3²)
print(power(2, 3))   # 8 (2³)
```



```
Привет, Максим!
9
8
```

# РАБОТА С ФАЙЛАМИ

Чтобы открыть файл для чтения или для записи, Если для создания объекта файла использовалась функция **open**, то следует явно закрывать файл по завершении работы с ним.

```
In [233]: path = "examples/segismundo.txt"
```

```
In [234]: f = open(path, encoding="utf-8")
```

Заккрытие файла возвращает ресурсы операционной системе:

```
In [237]: f.close()
```

По умолчанию файл открывается только для чтения в режиме 'r'. Далее дескриптор файла `f` можно рассматривать как список и перебирать строки:

```
for line in f:  
    print(line)
```

Упростить эту процедуру позволяет предложение **with**:

```
In [238]: with open(path) as f:  
.....:     lines = [x.rstrip() for x in f]
```

В таком случае файл `f` автоматически закрывается по выходе из блока **with**. Если не позаботиться о закрытии файлов, то в небольших скриптах может и не произойти ничего страшного, но в программах, работающих с большим количеством файлов, возможны проблемы. Если бы мы написали **f = open(path, 'w')**, то был бы создан новый файл **examples/ segismundo.txt** (будьте осторожны!), а старый был бы перезаписан. Существует также режим 'x', в котором создается допускающий запись файл, но лишь в том случае, если его еще не существует, а в противном случае возбуждается исключение.



# РАБОТА С ФАЙЛАМИ

При работе с файлами, допускающими чтение, чаще всего употребляются методы `read`, `seek` и `tell`. Метод `read` возвращает определенное число символов из файла. Что такое «символ», определяется кодировкой файла.

**read** продвигает указатель файла вперед на количество

```
In [239]: f1 = open(path)

In [240]: f1.read(10)
Out[240]: 'Sueca el r'

In [241]: f2 = open(path, mode="rb") # Двоичный режим

In [242]: f2.read(10)
Out[242]: b'Sue\xc3\xb1a el '
```

Закрывать файлы:

```
In [250]: f1.close()
In [251]: f2.close()
```

Режим	Описание
r	Режим чтения
w	Режим записи. Создается новый файл (а старый с тем же именем удаляется)
x	Режим записи. Создается новый файл, но возникает ошибка, если файл с таким именем уже существует
a	Дописывание в конец существующего файла (если файл не существует, он создается)
r+	Чтение и запись
b	Уточнение режима для двоичных файлов: 'rb' или 'wb'
t	Текстовый режим (байты автоматически декодируются в Unicode). Этот режим подразумевается по умолчанию, если не указано противное

**tell** сообщает текущую

```
In [243]: f1.tell()
Out[243]: 11
```

```
In [244]: f2.tell()
Out[244]: 10
```

**seek** изменяет позицию в файле на указанную:

```
In [247]: f1.seek(3)
Out[247]: 3
```

```
In [248]: f1.read(1)
Out[248]: 'ñ'
```

```
In [249]: f1.tell()
Out[249]: 5
```



# РАБОТА С ФАЙЛАМИ

Для записи текста в файл служат методы **write** или **writelines**

```
In [252]: path
Out[252]: 'examples/segismundo.txt'

In [253]: with open("tmp.txt", mode="w") as handle:
.....:     handle.writelines(x for x in open(path) if len(x) > 1)

In [254]: with open("tmp.txt") as f:
.....:     lines = f.readlines()

In [255]: lines
Out[255]:
['Sueña el rico en su riqueza,\n',
 'que más cuidados le ofrece;\n',
 'sueña el pobre que padece\n',
 'su miseria y su pobreza;\n',
 'sueña el que a medrar empieza,\n',
 'sueña el que afana y pretende,\n',
 'sueña el que agravia y ofende,\n',
 'y en el mundo, en conclusión,\n',
 'todos sueñan lo que son,\n',
 'aunque ninguno lo entiende.\n']
```

# РАБОТА С ФАЙЛАМИ

Метод	Описание
<code>read([size])</code>	Возвращает прочитанные из файла данные в виде строки. Необязательный аргумент <code>size</code> говорит, сколько байтов читать
<code>readable()</code>	Возвращает <code>True</code> , если файл поддерживает операции <code>read</code>
<code>readlines([size])</code>	Возвращает список прочитанных из файла строк. Необязательный аргумент <code>size</code> говорит, сколько строк читать
<code>write(string)</code>	Записывает переданную строку в файл
<code>writable()</code>	Возвращает <code>True</code> , если файл поддерживает операции <code>write</code>
<code>writelines(strings)</code>	Записывает переданную последовательность строк в файл
<code>close()</code>	Закрывает дескриптор файла
<code>flush()</code>	Сбрасывает внутренний буфер ввода-вывода на диск
<code>seek(pos)</code>	Перемещает указатель чтения-записи на байт файла с указанным номером
<code>seekable()</code>	Возвращает <code>True</code> , если файл поддерживает поиск, а следовательно, произвольный доступ (некоторые файлоподобные объекты этого не делают)
<code>tell()</code>	Возвращает текущую позицию в файле в виде целого числа
<code>closed</code>	<code>True</code> , если файл закрыт
<code>encoding</code>	Кодировка, используемая при интерпретации байтов файла как Unicode (обычно UTF-8)

# РАБОТА С ФАЙЛАМИ



```
# Запись в файл
with open("test.txt", "w") as file:
    file.write("Пример текста")

# Чтение из файла
with open("test.txt", "r") as file:
    content = file.read()
    print(content)
```



Пример текста

# ДЛЯ КАКИХ ДЕЙСТВИЙ ПРИ АНАЛИЗЕ ДАННЫХ ИСПОЛЬЗОВАТЬ ?

- **Списки** — для изменяемых упорядоченных коллекций
- **Кортежи** — для неизменяемых данных (например, координаты)
- **Словари** — для хранения данных в формате ключ-значение
- **Множества** — для работы с уникальными элементами