Урок 2. Встроенные типы и операции с ними

В уроке приведено описание ключевых встроенных типов данных, реализованных в Python. Разработчик может выполнять операции с данными традиционных типов, таких как строки, числа, логический тип. Рассматриваются списки, множества, кортежи, словари. Также в рамках урока мы познакомимся с понятиями тернарного оператора и оператора is. Обязательно коснёмся и некоторых операций, выполняемых с данными базовых типов, узнаем о трюках, которые может использовать разработчик для повышения лаконичности кода.

## 

## 

## 

## 

## 

## 

Оглавление

[Тип данных: число.](#_l1dkm4a8sr5b)

[Целые (int)](#_thab5b4q5i1y)

[Вещественные (float)](#_w1zrwyhgoa22)

[Комплексные (complex)](#_1zwzklybtkuj)

[Тип данных: строка](#_x3nh4gi7ceye)

[Конкатенация (сцепление)](#_z5z3u6hw49fw)

[Взятие элемента по индексу](#_4ja8nthexx1k)

[Извлечение среза](#_4n0hccuz5978)

[Механизмы реверса строк](#_m180q8n8k1m3)

[Таблица методов строк](#_n05qkbqowq75)

[Тип данных: список](#_o5l66csrcfbu)

[Тип данных: кортеж](#_g0lgc5t0tzme)

[Тип данных: множество](#_2x3rpf5etzud)

[Тип данных: словарь](#_kp65dmu0s261)

[Тип данных: bool](#_d7vfdhs4ym7o)

[Тип данных: bytes и bytearray](#_vysv5glo9lri)

[Тип данных: NoneType](#_ig49qj9buc5f)

[Тип данных: исключение](#_ft31uf1jso9t)

[О цикле for in для обхода последовательностей](#_r57u3uv7acwz)

[Понятие тернарного оператора](#_v4htlimm7qnv)

[Оператор is](#_yhudxu86tw2x)

[Десятка лучших трюков в Python](#_4yrpat2c51wl)

[Объединение списков без цикла](#_3tcled8eyy3n)

[Удаление дубликатов в списке](#_vhucmx5gr8kn)

[Обмен значениями через кортежи](#_i2g5f6brzad3)

[Вывод значения несуществующего ключа в словаре](#_a17ioigz6kd7)

[Поиск самых часто встречающихся элементов списка](#_iekfu8hvkgtg)

[Распаковка последовательностей при неизвестном количестве элементов](#_yatu3hki0r8t)

[Вывод с помощью функции print() без перевода строки](#_m5izp8mhuhz)

[Сортировка словаря по значениям](#_350iqa18iteo)

[Нумерованные списки](#_ebdsf1stx3n3)

[Транспонирование матрицы](#_b6lecbky2exl)

[Практическое задание](#_fv3bylrz63ae)

[Дополнительные материалы](#_vqapgkhnyfqp)

[Используемая литература](#_tnflastqfeho)

На этом уроке студент:

1. Узнает о встроенных в Python типах данных: числа, строки, байты, списки, кортежи, словари и т.д.
2. Научится работать с циклами для обхода типов данных — последовательностей.
3. Познакомится с тернарным оператором и оператором is.
4. Научится выполнять различные операции со встроенными типами данных и использовать их в своих программах.

# Тип данных: число.

В Python доступны следующие виды чисел: целые (тип **int**), вещественные (тип **float**), комплексные (тип **complex**). Более подробно каждый из них рассматривается в следующем разделе урока.

## Целые (int)

В Python 3 числа соответствуют обычным числам. Стандартные операции с целыми числами были рассмотрены на уроке 1. Дополнительно над целыми числами можно производить такие операции, как взятие числа по модулю и битовые операции.

|  |  |
| --- | --- |
| **Операция** | **Пример** |
| Взятие по модулю | print(abs(-6)) -> 6 |
| Побитовое И | print(4 & 6) -> 4 |
| Побитовое ИЛИ | print(4 | 6) -> 6 |
| Побитовое исключающее ИЛИ | print(4 ^ 6) -> 2 |
| Битовый сдвиг влево | print(4 << 6) -> 256 |
| Битовый сдвиг вправо | print(4 >> 6) -> 0 |

Числа в Python могут быть представлены не только в десятичной, но и в других системах счисления. Для перевода между системами счисления применяются специальные функции.

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Пример** |
| int() | Преобразовать к целому числу в десятичном формате (по умолчанию). Также допускается выбор другой системы счисления с помощью дополнительного параметра (от 2 до 36) | print(int(17.5)) -> 17  print(int(**'10001'**, 2)) -> 17 |
| bin() | Преобразовать к двоичному формату | print(bin(17))-> 0b10001 |
| oct() | Преобразовать к восьмеричному формату | print(oct(17)) -> 0o21 |
| hex() | Преобразовать к шестнадцатеричному формату | print(hex(17)) -> 0x11 |

## Вещественные (float)

Поддерживают операции, аналогичные операциям, выполняемым с целыми числами. Более подробно рассмотрены в первом уроке.

## Комплексные (complex)

Под комплексным числом понимается выражение вида **a + ib**, где **a** и **b** являются любыми действительными числами, **i** - мнимая единица.

|  |
| --- |
| n\_1 = complex(5, 6) print(n\_1) n\_2 = complex(7, 8) print(n\_2) |

Результат:

|  |
| --- |
| (5+6j) (7+8j) |

## 

# Тип данных: строка

Строка в Python — упорядоченный набор символов для хранения и представления текстовой информации.

Пример:

|  |
| --- |
| my\_str = "простая строка" print(my\_str) print(type(my\_str)) |

Результат:

|  |
| --- |
| простая строка <class 'str'> |

Простейший тип данных — упорядоченная неизменяемая коллекция элементов. Со строками в Python можно выполнять множество операций, например:

## Конкатенация (сцепление)

Пример:

|  |
| --- |
| s1 = 'abra' s2 = 'kadabra' print(s1 + s2) |

Результат:

|  |
| --- |
| abrakadabra |

## Взятие элемента по индексу

Пример:

|  |
| --- |
| s = 'abrakadabra' print(s[1]) |

Результат:

|  |
| --- |
| b |

## 

## Извлечение среза

Синтаксис: **[s:f:step]**, где **s** — начало среза, **f** — окончание, **step** — шаг (опционально).

Пример:

|  |
| --- |
| s = 'abrakadabra' print(s[4:7]) print(s[3:-3]) print(s[:5]) print(s[3:]) print(s[:]) print(s[::-1]) print(s[1:7:2]) |

Результат:

|  |
| --- |
| kad akada abrak akadabra abrakadabra arbadakarba baa |

## 

## Механизмы реверса строк

Рассмотрим следующие механизмы реверса строк: срез, обратная итерация, алгоритм реверса на месте.

1. Срез.

Пример:

|  |
| --- |
| def my\_str\_reverse(string):  return string[::-1]  print(my\_str\_reverse("abrakadbra")) |

Результат:

|  |
| --- |
| arbdakarba |

1. Обратная итерация.

Пример:

|  |
| --- |
| for el in reversed("abrakadbra"):  print(el) |

Результат:

|  |
| --- |
| a r b d a k a r b a |

1. Реверс на месте.

Пример:

|  |
| --- |
| def my\_str\_reverse(string):  symbols = list(string)  for el in range(len(string) // 2):  tmp = symbols[el]  symbols[el] = symbols[len(string) - el - 1]  symbols[len(string) - el - 1] = tmp  return ''.join(symbols)  data = my\_str\_reverse("abrakadabra") print(data) |

Результат:

|  |
| --- |
| arbadakarba |

### Таблица методов строк

Рассмотрим методы, применяемые в приложениях для операций со строками и примеры их использования.

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Пример** |
| len(строка) | Возвращает длину строки | print(len(**"my\_string"**))  -> 9 |
| строка.split(<разделитель>) | Разбить строку по разделителю | print(**"раз два три"**.split())  -> ['раз', 'два', 'три']  print(**"четыре\_пять\_шесть"**.split(**'\_'**)) -> ['четыре', 'пять', 'шесть'] |
| <разделитель>.join(список) | Собрать строку из списка с указанным разделителем | print(**'\_'**.join([**'раз'**, **'два'**, **'три'**]))  -> раз\_два\_три  print(**''**.join([**'раз'**, **'два'**, **'три'**])) -> раздватри |
| строка.title() | Перевести первую букву каждого слова в верхний регистр, остальные - в нижний | print(**"ехал грека через реку"**.title())  -> Ехал Грека Через Реку |
| строка.upper() | Преобразовать строку к верхнему регистру | print(**'простая строка'**.upper())  -> ПРОСТАЯ СТРОКА |
| строка.lower() | Преобразовать строку к нижнему регистру | print(**'ПРОСТАЯ СТРОКА'**.lower())  -> простая строка |
| строка.istitle() | Проверить, начинаются ли слова строки с буквы в верхнем регистре | print(**'Ехал Грека Через Реку'**.istitle()) -> True  print(**'Ехал Грека Через реку'**.istitle()) -> False |
| строка.isupper() | Проверить, состоит ли строка из символов в верхнем регистре | print(**'ПРОСТАЯ СТРОКА'**.isupper())  -> True  print(**'простая строка'**.isupper())  -> False |
| строка.islower() | Проверить, состоит ли строка из символов в нижнем регистре | print(**'простая строка'**.islower())  -> True  print(**'ПРОСТАЯ СТРОКА'**.islower())  -> False |
| ord(символ) | Получить ASCII-код для символа | print(ord(**'b'**)) -> 98 |
| chr(код) | Получить символ по ASCII-коду | print(chr(98)) -> **'b'** |
| строка.count(подстрока, [начало], [конец]) | Вернуть количество вхождений подстроки в строку | print(**'рарара'**.count(**'ра'**)) -> 3  print(**'рарара'**.count(**'ра'**, 2, 4)) -> 1 |
| строка.capitalize() | Перевести первый символ строки в верхний регистр, остальные - в нижний | print(**'cТРОКА'**.capitalize()) -> Cтрока |
| строка.startswith(шаблон) | Проверить, начинается ли строка с шаблона | print(**'рарара'**.startswith(**'ра'**))  -> True  print(**'рарара'**.startswith(**'не'**))  -> False |
| строка.endswith(шаблон) | Проверить, заканчивается ли строка шаблоном | print(**'рарара'**.endswith(**'ра'**))  -> True  print(**'рарара'**.endswith(**'не'**))  -> False |
| строка.replace(шаблон, замена) | Заменить в строке шаблон на указанную подстроку | print(**'рарара'**.replace(**'ра'**, **'не'**)) -> **'ненене'** |
| строка.index(подстрока, [начало], [конец]) | Найти подстроку в строке. Получить позицию первого вхождения или получить ValueError | print(**'рарарара'**.index(**'ра'**))  -> 0  print(**'рарарара'**.index(**'ра'**, 4, 6))  -> 4  print(**'рарарара'**.index(**'ра'**, 10, 20)) -> ValueError: substring not found |
| строка.find(подстрока, [начало], [конец]) | Найти подстроку в строке. Получить позицию первого вхождения или получить -1 | print(**'рарарара'**.find(**'ра'**))  -> 0  print(**'рарарара'**.find(**'ра'**, 4, 6))  -> 4  print(**'рарарара'**.find(**'ра'**, 10, 20)) -> -1 |

## 

С другими методами строк вы можете ознакомиться по [ссылке](https://pythonworld.ru/tipy-dannyx-v-python/stroki-funkcii-i-metody-strok.html).

# Тип данных: список

В Python массивов как таковых не существует. Их роль выполняют списки. Под списками понимаются упорядоченные изменяемые наборы объектов произвольного типа. Самый простой способ создать список — применить функцию **list()** к итерируемому объекту, например, к строке:

Пример:

|  |
| --- |
| print(list('обычная строка')) |

## 

Результат:

|  |
| --- |
| ['о', 'б', 'ы', 'ч', 'н', 'а', 'я', ' ', 'с', 'т', 'р', 'о', 'к', 'а'] |

Список основных методов работы со списками приведен в таблице ниже:

|  |  |
| --- | --- |
| **Метод** | **Назначение** |
| result\_list.append(el) | Добавить элемент el в конец списка result\_list |
| result\_list.extend(my\_list) | Расширить список result\_list —  добавить в конец элементы списка my\_list |
| result\_list.insert(pos, el) | Разместить на позиции pos (индекс элемента списка) элемент el |
| result\_list.remove(el) | Удалить из списка первый элемент со значением el |
| result\_list.pop(pos) | Удалить элемент с индексом pos |
| result\_list.index(el) | Получить позицию (индекс) первого элемента со значением el |
| result\_list.count(el) | Возвращает количество элементов списка со значением el |
| result\_list.sort([key=функция]) | Выполнить сортировку списка на основе указанной функции |
| result\_list.reverse() | Выполнить реверс списка (развернуть список) |
| result\_list.copy() | Создать поверхностную копию списка |
| result\_list.clear() | Очистить список |

Пример:

|  |
| --- |
| result\_list = [2, 'text', 456, 45.3, None] print(result\_list) |

Результат:

|  |
| --- |
| [2, 'text', 456, 45.3, None] |

Пример:

|  |
| --- |
| result\_list = [2, 'text', 456, 45.3, None]  *# append* result\_list.append("new\_el") print(result\_list) |

Результат:

|  |
| --- |
| [2, 'text', 456, 45.3, None, 'new\_el'] |

Пример:

|  |
| --- |
| result\_list = [2, 'text', 456, 45.3, None]  *# extend* result\_list.extend([8, 9, 10]) print(result\_list) |

Результат:

|  |
| --- |
| [2, 'text', 456, 45.3, None, 8, 9, 10] |

Пример:

|  |
| --- |
| result\_list = [2, 'text', 456, 45.3, None]  *# insert* result\_list.insert(1, "ins\_el") print(result\_list) |

Результат:

|  |
| --- |
| [2, 'ins\_el', 'text', 456, 45.3, None] |

Пример:

|  |
| --- |
| result\_list = [2, 'text', 456, 45.3, None, "ins\_el"]  *# remove* result\_list.remove("ins\_el") print(result\_list) |

Результат:

|  |
| --- |
| [2, 'text', 456, 45.3, None] |

Пример:

|  |
| --- |
| result\_list = [2, 'text', 456, 45.3, None]  *# pop* result\_list.pop(1) print(result\_list) |

Результат:

|  |
| --- |
| [2, 456, 45.3, None] |

Пример:

|  |
| --- |
| result\_list = [2, 'text', 456, 45.3, None]  *# index* print(result\_list.index(None)) |

Результат:

|  |
| --- |
| 4 |

Пример:

|  |
| --- |
| result\_list = [2, 'text', 456, 45.3, None]  *# count* print(result\_list.count(2)) |

Результат:

|  |
| --- |
| 1 |

Пример:

|  |
| --- |
| result\_list = [2, 'text', 456, 45.3, None]  *# reverse* result\_list.reverse() print(result\_list) |

Результат:

|  |
| --- |
| [None, 45.3, 456, 'text', 2] |

Пример:

|  |
| --- |
| result\_list = [2, 'text', 456, 45.3, None]  *# copy* copy\_list = result\_list.copy() print(copy\_list) |

Результат:

|  |
| --- |
| [2, 'text', 456, 45.3, None] |

Пример:

|  |
| --- |
| result\_list = [2, 'text', 456, 45.3, None]  *# clear* result\_list.clear() print(result\_list) |

Результат:

|  |
| --- |
| [] |

На примере списков рассмотрим использование Python-операторов **is** и **in**.

Пример:

|  |
| --- |
| my\_list = [10, 20, 30] print((40 or 50) in my\_list) |

Результат:

|  |
| --- |
| False |

В данном примере используются возможности операторов **or** и **in**. Проверяется, входит ли хотя бы одно из указанных в скобках чисел в исходный список.

Пример:

|  |
| --- |
| list\_1 = [30, 'string', None, False] list\_2 = [30, 'string', None, False]  print(list\_1 is list\_2)  list\_2 = list\_1  print(list\_1 is list\_2) |

Результат:

|  |
| --- |
| False True |

В приведенном примере используется оператор идентичности (**is**). В первом случае результатом является значение **False**, т. к. переменные **list\_1** и **list\_2** ссылаются на разные объекты. Во втором случае получается значение **True**, т. к. ссылка слева (**list\_2**) указывает на тот же объект, что и ссылка справа (**list\_1**).

# Тип данных: кортеж

Кортеж представляет собой аналогичную списку структуру с одним отличием. Кортеж — неизменяемая структура. Самый простой способ создать кортеж — применить функцию **tuple()** к итерируемому объекту.

Пример:

|  |
| --- |
| print(tuple('обычная строка')) |

Результат:

|  |
| --- |
| ('о', 'б', 'ы', 'ч', 'н', 'а', 'я', ' ', 'с', 'т', 'р', 'о', 'к', 'а') |

## 

Преимущества кортежи:

* Защищают от неверных действий пользователя. Кортеж — неизменяемый список, защищен от случайных и намеренных изменений.
* Меньший размер по сравнению со списками.

Список:

|  |
| --- |
| my\_l = [4, 234, 45.8, "text", "word", "el", True, None] print(my\_l.\_\_sizeof\_\_()) |

Результат:

|  |
| --- |
| 104 |

Кортеж:

|  |
| --- |
| my\_t = (4, 234, 45.8, "text", "word", "el", True, None) print(my\_t.\_\_sizeof\_\_()) |

Результат:

|  |
| --- |
| 88 |

В этих примерах сравниваются список и кортеж с одинаковыми данными. Но в итоге кортеж — более экономичная структура хранения данных. Список занимает 104 байта, а кортеж — 88.

Кортежи, как коллекции, поддерживают те же операции, что и списки. Операции не должны изменять саму коллекцию (например, **index()**, **count()**).

# Тип данных: множество

Множество в Python — это контейнер с не повторяющимися элементами, расположенными в случайном порядке. Множество, создаваемое с помощью функции **set()**, представляет собой изменяемый тип данных, **frozenset()** — неизменяемый.

Пример:

|  |
| --- |
| perem\_1 = set('abrakadabra') perem\_2 = frozenset('abrakadabra') print(perem\_1) print(perem\_2) perem\_1.add('!') print(perem\_1) perem\_2.add('!') print(perem\_2) |

Результат:

|  |
| --- |
| {'a', 'k', 'b', 'd', 'r'} frozenset({'a', 'k', 'b', 'd', 'r'}) {'k', '!', 'r', 'b', 'a', 'd'} Traceback (most recent call last): \*\* IDLE Internal Exception:  File "/usr/lib/python3.5/idlelib/run.py", line 351, in runcode  exec(code, self.locals)  File "run.py", line 7, in <module>  perem\_2.add('!') AttributeError: 'frozenset' object has no attribute 'add' |

Список основных методов работы с изменяемыми множествами приведен в таблице ниже:

|  |  |
| --- | --- |
| **Метод** | **Назначение** |
| .add(el) | Добавить элемент в множество |
| .remove(el) | Удалить элемент из множества. Если элемент отсутствует — ошибка KeyError |
| .discard(el) | Удалить элемент из множества |
| .pop() | Удалить первый элемент из множества. Множества не упорядочены, поэтому первый элемент множества заранее не определен |
| .copy() | Создать копию множества |
| .clear() | Очистить множество |

Пример:

|  |
| --- |
| my\_set = {400, None, "text", True} print(my\_set) |

Результат:

|  |
| --- |
| {400, True, None, 'text'} |

Пример:

|  |
| --- |
| my\_set = {400, None, "text", True}  *# add* my\_set.add("another\_el") print(my\_set) |

Результат:

|  |
| --- |
| {True, 'another\_el', 'text', 400, None} |

## 

Пример:

|  |
| --- |
| my\_set = {400, None, "text", True}  *# remove* my\_set.remove("text") print(my\_set) |

Результат:

|  |
| --- |
| {400, True, None} |

Пример:

|  |
| --- |
| my\_set = {400, None, "text", True}  *# discard* my\_set.discard(400) print(my\_set) |

Результат:

|  |
| --- |
| {True, 'text', None} |

## 

Пример:

|  |
| --- |
| my\_set = {400, None, "text", True}  *# pop* my\_set.pop() print(my\_set) |

Результат:

|  |
| --- |
| {True, 'text', None} |

## 

Пример:

|  |
| --- |
| my\_set = {400, None, "text", True}  *# copy* print(my\_set.copy()) |

Результат:

|  |
| --- |
| {400, 'text', None, True} |

## 

Пример:

|  |
| --- |
| my\_set = {400, None, "text", True}  *# clear* my\_set.clear() print(my\_set) |

Результат:

|  |
| --- |
| set() |

Изменяемые множества (**set()**) и неизменяемые (**frozenset()**) являются аналогией списков и кортежей.

Пример:

|  |
| --- |
| my\_s = set('abrakadabra') my\_fs = frozenset('abrakadabra') print(my\_s == my\_fs) |

Результат:

|  |
| --- |
| True |

## 

Пример:

|  |
| --- |
| *# вычитание* my\_s = set('kadabra') print(my\_s) my\_fs = frozenset('abra') print(my\_fs) print(my\_s - my\_fs) |

Результат:

|  |
| --- |
| {'r', 'a', 'b', 'd', 'k'} frozenset({'r', 'a', 'b'}) {'k', 'd'} |

## 

Пример:

|  |
| --- |
| *# объединение* my\_s = set('kadabra') print(my\_s) my\_fs = frozenset('abra') print(my\_fs) print(my\_s | my\_fs) |

Результат:

|  |
| --- |
| {'b', 'r', 'a', 'd', 'k'} frozenset({'b', 'a', 'r'}) {'b', 'd', 'r', 'k', 'a'} |

# 

# Тип данных: словарь

Словарь — неупорядоченный набор произвольных объектов с доступом по ключу. Один из вариантов создания словаря — с помощью функции **dict()**.

Пример:

|  |
| --- |
| my\_dict = dict(key\_1='val\_1', key\_2='val\_2') print(my\_dict) |

Результат:

|  |
| --- |
| {'key\_1': 'val\_1', 'key\_2': 'val\_2'} |

Список основных методов работы со словарями приведен в таблице ниже:

|  |  |
| --- | --- |
| **Метод** | **Назначение** |
| .keys() | Возвращает список ключей словаря |
| .values() | Возвращает список значений |
| .items() | Возвращает список кортежей (ключ, значение) |
| .get(key) | Возвращает значение, соответствующее ключу key. Если ключ отсутствует, возвращает значение None |
| .popitem() | Удаляет элемент словаря и возвращает пару (ключ, значение). Если элементы отсутствуют, возникает исключение KeyError |
| .setdefault(key) | Возвращает значение, соответствующее ключу. Если ключ отсутствует, создается элемент с указанным ключом и значением None |
| .pop(key) | Удаляет ключ и возвращает значение, соответствующее ключу |
| .update(new\_dict) | Добавляет пары (ключ, значение) в текущий словарь из словаря new\_dict. Имеющиеся ключи перезаписываются |
| .copy() | Возвращает копию словаря |
| .clear() | Очищает словарь |

Пример:

|  |
| --- |
| my\_dict = {"key\_1": 500, 2: 400, "key\_3": True, 4: None}  *# keys* print(my\_dict.keys()) |

Результат:

|  |
| --- |
| dict\_keys(['key\_1', 2, 'key\_3', 4]) |

Пример:

|  |
| --- |
| my\_dict = {"key\_1": 500, 2: 400, "key\_3": True, 4: None}  *# values* print(my\_dict.values()) |

Результат:

|  |
| --- |
| dict\_values([500, 400, True, None]) |

Пример:

|  |
| --- |
| my\_dict = {"key\_1": 500, 2: 400, "key\_3": True, 4: None}  *# items* print(my\_dict.items()) |

Результат:

|  |
| --- |
| dict\_items([('key\_1', 500), (2, 400), ('key\_3', True), (4, None)]) |

Пример:

|  |
| --- |
| my\_dict = {"key\_1": 500, 2: 400, "key\_3": True, 4: None}  *# get* print(my\_dict.get(2)) |

Результат:

|  |
| --- |
| 400 |

Пример:

|  |
| --- |
| my\_dict = {"key\_1": 500, 2: 400, "key\_3": True, 4: None}  *# popitem* print(my\_dict.popitem()) print(my\_dict.popitem()) print(my\_dict.popitem()) print(my\_dict.popitem()) |

Результат:

|  |
| --- |
| (4, None) ('key\_3', True) (2, 400) ('key\_1', 500) |

Пример:

|  |
| --- |
| my\_dict = {"key\_1": 500, 2: 400, "key\_3": True, 4: None}  *# setdefault* print(my\_dict.setdefault(5)) print(my\_dict.items()) |

Результат:

|  |
| --- |
| None dict\_items([('key\_1', 500), (2, 400), ('key\_3', True), (4, None), (5, None)]) |

Пример:

|  |
| --- |
| my\_dict = {"key\_1": 500, 2: 400, "key\_3": True, 4: None}  *# pop* print(my\_dict.pop(2)) print(my\_dict.items()) |

Результат:

|  |
| --- |
| 400 dict\_items([('key\_1', 500), ('key\_3', True), (4, None)]) |

Пример:

|  |
| --- |
| my\_dict = {"key\_1": 500, 2: 400, "key\_3": True, 4: None}  *# update* my\_dict.update({8: 8, 9: 9, 10: 10}) print(my\_dict.items()) |

Результат:

|  |
| --- |
| dict\_items([('key\_1', 500), (2, 400), ('key\_3', True), (4, None), (8, 8), (9, 9), (10, 10)]) |

Пример:

|  |
| --- |
| my\_dict = {"key\_1": 500, 2: 400, "key\_3": True, 4: None}  *# copy* print(my\_dict.copy()) |

Результат:

|  |
| --- |
| {'key\_1': 500, 2: 400, 'key\_3': True, 4: None} |

Пример:

|  |
| --- |
| my\_dict = {"key\_1": 500, 2: 400, "key\_3": True, 4: None}  *# clear* my\_dict.clear() print(my\_dict.items()) |

Результат:

|  |
| --- |
| dict\_items([]) |

# 

# Тип данных: bool

Логический тип, применяется в представлениях истинности.

Пример:

|  |
| --- |
| print(True) print(bool(20)) print(bool('text'))  print(False) print(bool(0)) print(bool('')) print(bool()) |

Результат:

|  |
| --- |
| True True True False False False False |

Функция **bool()** позволяет привести любое значение к логическому типу (если это значение может быть интерпретировано в качестве логического типа).

# Тип данных: bytes и bytearray

Байты являются единицей хранения информации (текстовой, графической, звуковой). Байтовое представление похоже на обычное строковое, но с рядом отличий.

Пример:

|  |
| --- |
| print(b'text') print('текст'.encode('utf-8')) print(bytes('text', encoding = 'utf-8')) print(bytes([10, 20, 30, 40])) |

## 

Результат:

|  |
| --- |
| b'text' b'\xd1\x82\xd0\xb5\xd0\xba\xd1\x81\xd1\x82' b'text' b'\n\x14\x1e(' |

Тип данных **bytearray** представляет собой массив байт. В отличие от **bytes** является изменяемым.

Пример:

|  |
| --- |
| my\_var = bytearray(b"some text") print(my\_var) print(my\_var[0])  *#my\_var[0] = b'h' -> TypeError: an integer is required* my\_var[0] = 105 print(my\_var)  my\_var = bytearray(b"some text") for i in range(len(my\_var)):  my\_var[i] += i  print(my\_var) |

Результат:

|  |
| --- |
| bytearray(b'some text') 115 bytearray(b'iome text') bytearray(b'spoh$yk\x7f|') |

# Тип данных: NoneType

Значение **None** переменной сигнализирует о присвоении пустого значения этой переменной, т. е., оно обозначает «здесь нет значения». Присвоение переменной такого значения является одним из вариантов ее сброса в пустое состояние. Python — язык объектно-ориентированный. **None** также принадлежит к объектам и обладает своим типом.

|  |
| --- |
| print(type(None)) |

Результат:

|  |
| --- |
| <class 'NoneType'> |

Рассмотрим еще один пример:

|  |
| --- |
| my\_dict = {'name': 'Ivan', 'surname': 'Ivanov', 'age': 40, 'position': None} for el in my\_dict:  if my\_dict[el] == None:  print(f"Для сотрудника пока не определен параметр: {el}") |

Результат:

|  |
| --- |
| Для сотрудника пока не определен параметр: position |

В этом примере выполняется перебор ключей словаря и проверка, есть ли в словаре значения типа None.

# Тип данных: исключение

Exceptions представляют собой еще один тип данных и предназначены для вывода сообщений об ошибках.

Пример:

|  |
| --- |
| print(500 / 0) |

Результат:

|  |
| --- |
| Traceback (most recent call last):  File "my\_file.py", line 1, in <module>  print(500 / 0) ZeroDivisionError: division by zero |

В данном случае интерпретатор вывел информацию о наличии исключения (**ZeroDivisionError**), связанного с делением на 0 (division by zero). Это только один из типов исключений. В Python предусмотрены и другие, которые будут рассмотрены далее (как и механизмы обработки исключений).

# О цикле for in для обхода последовательностей

В Python списки, кортежи, строки относятся к последовательностям. Для выполнения однотипных операций с каждым элементом последовательностей в Python применяются циклы **for**. Данная функция отвечает за генерацию набора чисел в пределах указанного диапазона.

Общий синтаксис:

|  |
| --- |
| for [переменная-итератор] in [последовательность]:  [действия, выполняемые для каждой переменной] |

Пример:

|  |
| --- |
| for el in "my\_string":  print(el) |

Результат:

|  |
| --- |
| m  y  \_  s  t  r  i  n  g |

В этом примере **el** — переменная-итератор, последовательно принимающая значения — элементы строки.

Кортежи относятся к неизменяемым последовательностям, но допускают перебор элементов и выполнение операций с ними.

Пример:

|  |
| --- |
| my\_tuple = (1, 2, 3, 4, 5) my\_list = [] for el in my\_tuple:  my\_list.append(el \* 2) print(my\_list) |

Результат:

|  |
| --- |
| [2, 4, 6, 8, 10] |

Проверим работу цикла **for** на примере списка.

Пример:

|  |
| --- |
| orig\_list = [1, 2, 3, 4, 5] new\_list = [] for el in orig\_list:  new\_list.append(el / 2) print(new\_list) |

Результат:

|  |
| --- |
| [0.5, 1.0, 1.5, 2.0, 2.5] |

И на примере множества:

|  |
| --- |
| orig\_set = {1, 2, 3, 4, 5} new\_set = set() for el in orig\_set:  new\_set.add(el / 2) print(new\_set) |

Результат:

|  |
| --- |
| {0.5, 1.0, 2.0, 2.5, 1.5} |

Возможности цикла **for** применяются и к словарям:

Пример:

|  |
| --- |
| my\_dict = {'title': 'Samsung Galaxy', 'price': 20000, 'country': 'China', 'year': '2016'} for key, value in my\_dict.items():  print(f"{key} - {value}") |

Результат:

|  |
| --- |
| title - Samsung Galaxy price - 20000 country - China year - 2016 |

# Понятие тернарного оператора

Понятие тернарного оператора в Python очень близко к понятию условного выражения. Тернарные операторы позволяют вернуть некоторый результат в зависимости от истинности или ложности некоторого условия.

Шаблон тернарного оператора:

|  |
| --- |
| condition\_if\_true if condition else condition\_if\_false |

Пример:

|  |
| --- |
| is\_checked = True mode = "checked" if is\_checked else "not checked" print(mode) |

Результат:

|  |
| --- |
| checked |

Использование представленного подхода позволяет выполнить быструю проверку условия вместо использования нескольких ветвей с **if**. Код получается более компактным и читабельным.

Существует и другой вариант использования данного подхода (с кортежами):

Шаблон:

|  |
| --- |
| (if\_check\_is\_false, if\_check\_is\_true)[param\_to\_check] |

Пример:

|  |
| --- |
| checked = True personality = ("проверено", "не проверено")[checked] print(personality) |

Результат:

|  |
| --- |
| не проверено |

Данный механизм в Python является работоспособным, поскольку значение **True** соответствует единице, а **False** — нулю. Кроме кортежей допускается использование списков.

В Python также предусмотрена возможность использования более лаконичной версии тернарного оператора.

Пример:

|  |
| --- |
| print(True or "Some") print(False or "Some") |

Результат:

|  |
| --- |
| True Some |

Этот механизм удобно использовать, когда требуется проверить возвращаемое функцией значение.

Пример:

|  |
| --- |
| func\_return = None message = func\_return or "Функция ничего не возвращает" print(message) |

Результат:

|  |
| --- |
| Функция ничего не возвращает |

# 

# Оператор is

Данный оператор проверяет тождественность (идентичность) двух объектов в памяти. Возвращает значение **True** (истина), если переменные ссылаются на один и тот же объект.

Пример:

|  |
| --- |
| a = 20 b = 20  if a is b:  print("Переменные идентичны") else:  print("Переменные не идентичны") |

Результат:

|  |
| --- |
| Переменные идентичны |

Важная особенность использования оператора **is** заключается в том, что идентичен оператору **==**.

**==** — проверка равенства значений двух объектов.

**is** — проверка идентичности объектов, т. е., проверка того, что переменные указывают на один и тот же объект в памяти.

Пример:

|  |
| --- |
| obj\_1 = [10, 20, 30, 40] obj\_2 = obj\_1 print(obj\_1 == obj\_2) print(obj\_1 is obj\_2)  obj\_2 = obj\_1[:] *# переменная obj\_2 ссылается на копию obj\_1* print(obj\_1 == obj\_2) print(obj\_1 is obj\_2) print(obj\_1 is not obj\_2) |

Результат:

|  |
| --- |
| True True True False True |

Для проверки соответствия объекта типу **NoneType** предпочтительно использование оператора **is**.

Пример:

|  |
| --- |
| obj\_1 = None print(obj\_1 is None) |

Результат:

|  |
| --- |
| True |

# 

# Десятка лучших трюков в Python

В завершение урока познакомимся с набором интересных приемов, которые пригодятся вам на практике.

## Объединение списков без цикла

Явный вариант решения задачи объединения списков разной длины предполагает перебор элементов в цикле. Но возможно и более лаконичное решение через функцию **sum()**.

Пример:

|  |
| --- |
| my\_list = [[10, 20, 30], [40, 50], [60], [70, 80, 90]] print(sum(my\_list, [])) |

# 

Результат:

|  |
| --- |
| [10, 20, 30, 40, 50, 60, 70, 80, 90] |

## Удаление дубликатов в списке

Это очень популярный трюк, предполагающий трансформацию списка во множество и обратно в список для удаления дублирующих элементов в списке.

Пример:

|  |
| --- |
| my\_list = [10, 10, 3, 4, 5, 9, 30, 30] print(list(set(my\_list))) |

# 

Результат:

|  |
| --- |
| [3, 4, 5, 9, 10, 30] |

## Обмен значениями через кортежи

Позволяет выполнять обмен значения без создания дополнительной переменной. Трюк допустим для любого числа переменных.

Пример:

|  |
| --- |
| var\_1, var\_2 = 20, 30 print(var\_1, var\_2) var\_1, var\_2 = var\_2, var\_1 print(var\_1, var\_2) |

Результат:

|  |
| --- |
| 20 30 30 20 |

Правая часть выражения может представлять собой любой итерируемый объект. Главное, чтобы число элементов в левой и правой частях совпадало.

## Вывод значения несуществующего ключа в словаре

Если попытаться обратиться к несуществующему ключу словаря, возникнет исключение:

Пример:

|  |
| --- |
| my\_dict = {'k\_1': 20, 'k\_2': True, 'k\_3': 'text'} print(my\_dict['k\_4']) |

Результат:

|  |
| --- |
| KeyError: 'k\_4' |

Во избежание такой ситуации можно воспользоваться методом **get()**.

Пример:

|  |
| --- |
| my\_dict = {'k\_1': 20, 'k\_2': True, 'k\_3': 'text'} print(my\_dict.get('k\_4')) |

Результат:

|  |
| --- |
| None |

## Поиск самых часто встречающихся элементов списка

Искать самый часто встречающийся элемент можно используя встроенную функцию **max()**, которая ищет наибольшее значение не только для итерируемого объекта, но и для результатов применения к этому объекту функции. Можно преобразовать список во множество и применить метод **count** для определения количества вхождений элемента в итерируемый объект.

Пример:

|  |
| --- |
| my\_list = [10, 20, 20, 20, 30, 50, 70, 30] print(max(set(my\_list), key=my\_list.count)) |

Результат:

|  |
| --- |
| 20 |

## Распаковка последовательностей при неизвестном количестве элементов

В Python оператор **\*** соответствует операции распаковки последовательности. Переменная с этим параметром связывается с частью списка, содержащей все неприсвоенные элементы, соответствующие текущей позиции.

Пример:

|  |
| --- |
| my\_list = [20, 30, 40, 50] \*el\_1, el\_2, el\_3 = my\_list print(el\_1, el\_2, el\_3) el\_1, \*el\_2, el\_3 = my\_list print(el\_1, el\_2, el\_3) el\_1, el\_2, \*el\_3 = my\_list print(el\_1, el\_2, el\_3) el\_1, el\_2, el\_3, \*el\_4 = my\_list print(el\_1, el\_2, el\_3, el\_4) el\_1, el\_2, el\_3, el\_4, \*el\_5 = my\_list print(el\_1, el\_2, el\_3, el\_4, el\_5) |

Результат:

|  |
| --- |
| [20, 30] 40 50 20 [30, 40] 50 20 30 [40, 50] 20 30 40 [50] 20 30 40 50 [] |

## Вывод с помощью функции print() без перевода строки

По умолчанию функция **print()** добавляет символ перевода строки, который можно отменить, добавив в функцию параметр **end** со значением пустой строки.

Пример:

|  |
| --- |
| for el in ["ab", "ra", "kada", "bra"]:  print(el, end='') |

Результат:

|  |
| --- |
| abrakadabra |

## Сортировка словаря по значениям

По умолчанию элементы словаря сортируются по наименованиям ключей.

Пример:

|  |
| --- |
| my\_dict = {'python': 1991, 'java': 1995, 'c++': 1983} print(sorted(my\_dict)) |

Результат:

|  |
| --- |
| ['c++', 'java', 'python'] |

Но есть возможность реализации сортировки по значениям элементов.

Пример:

|  |
| --- |
| my\_dict = {'python': 1991, 'java': 1995, 'c++': 1983} print(sorted(my\_dict, key=my\_dict.get)) |

Результат:

|  |
| --- |
| ['c++', 'python', 'java'] |

## 

## Нумерованные списки

Для реализации нумерованного списка можно воспользоваться уже знакомой нам функцией **enumerate()**.

Пример:

|  |
| --- |
| for ind, el in enumerate(['ноль', 'один', 'два', 'три']):  print(ind, el) |

Результат:

|  |
| --- |
| 0 ноль 1 один 2 два 3 три |

Пример:

|  |
| --- |
| for ind, el in enumerate(['один', 'два', 'три'], 1):  print(ind, el) |

Результат:

|  |
| --- |
| 1 один 2 два 3 три |

## Транспонирование матрицы

Под транспонированием понимается замена местами строк и столбцов матрицы (двумерного массива). Для этого можно воспользоваться уже знакомой функцией zip().

Пример:

|  |
| --- |
| old\_list = [('a', 'b'), ('c', 'd'), ('e', 'f')] new\_list = zip(\*old\_list) print(list(new\_list)) |

Результат:

|  |
| --- |
| [('a', 'c', 'e'), ('b', 'd', 'f')] |

# 

# Практическое задание

1. Создать список и заполнить его элементами различных типов данных. Реализовать скрипт проверки типа данных каждого элемента. Использовать функцию **type()** для проверки типа. Элементы списка можно не запрашивать у пользователя, а указать явно, в программе.
2. Для списка реализовать обмен значений соседних элементов, т.е. Значениями обмениваются элементы с индексами 0 и 1, 2 и 3 и т.д. При нечетном количестве элементов последний сохранить на своем месте. Для заполнения списка элементов необходимо использовать функцию **input()**.
3. Пользователь вводит месяц в виде целого числа от 1 до 12. Сообщить к какому времени года относится месяц (зима, весна, лето, осень). Напишите решения через list и через dict.
4. Пользователь вводит строку из нескольких слов, разделённых пробелами. Вывести каждое слово с новой строки. Строки необходимо пронумеровать. Если в слово длинное, выводить только первые 10 букв в слове.
5. Реализовать структуру «Рейтинг», представляющую собой не возрастающий набор натуральных чисел. У пользователя необходимо запрашивать новый элемент рейтинга. Если в рейтинге существуют элементы с одинаковыми значениями, то новый элемент с тем же значением должен разместиться после них.

Подсказка. Например, набор натуральных чисел: 7, 5, 3, 3, 2.

Пользователь ввел число 3. Результат: 7, 5, 3, 3, 3, 2.

Пользователь ввел число 8. Результат: 8, 7, 5, 3, 3, 2.

Пользователь ввел число 1. Результат: 7, 5, 3, 3, 2, 1.

Набор натуральных чисел можно задать непосредственно в коде, например, my\_list = [7, 5, 3, 3, 2].

1. \*Реализовать структуру данных «Товары». Она должна представлять собой список кортежей. Каждый кортеж хранит информацию об отдельном товаре. В кортеже должно быть два элемента — номер товара и словарь с параметрами (характеристиками товара: название, цена, количество, единица измерения). Структуру нужно сформировать программно, т.е. запрашивать все данные у пользователя.

Пример готовой структуры:

[

(1, {“название”: “компьютер”, “цена”: 20000, “количество”: 5, “eд”: “шт.”}),

(2, {“название”: “принтер”, “цена”: 6000, “количество”: 2, “eд”: “шт.”}),

(3, {“название”: “сканер”, “цена”: 2000, “количество”: 7, “eд”: “шт.”})

]

Необходимо собрать аналитику о товарах. Реализовать словарь, в котором каждый ключ — характеристика товара, например название, а значение — список значений-характеристик, например список названий товаров.

Пример:

{

“название”: [“компьютер”, “принтер”, “сканер”],

“цена”: [20000, 6000, 2000],

“количество”: [5, 2, 7],

“ед”: [“шт.”]

}

# Дополнительные материалы

1. [Числа: целые, вещественные, комплексные](http://ctrockii.blogspot.ru/2012/06/python-path-windows.html).
2. [Переменные и типы данных](https://metanit.com/python/tutorial/2.2.php).
3. [Типы данных в Python 3](https://www.8host.com/blog/tipy-dannyx-v-python-3/).

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Язык программирования Python 3 для начинающих и чайников](https://pythonworld.ru/).
2. [Программирование в Python](https://python-scripts.com/).
3. [Учим Python качественно (habr)](https://habrahabr.ru/post/150302/).
4. [Самоучитель по Python](http://pythonworld.ru/samouchitel-python).
5. [Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 (4-е издание)](http://www.proklondike.com/books/python/lutz_python_2011.html).