

Урок 2. Встроенные типы и операции с ними

В уроке приводится 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. В рамках урока мы познакомимся с понятиями тернарного оператора и оператора `is`. Коснёмся и некоторых операций, выполняемых с данными базовых типов. Узнаем о трюках, которые может использовать разработчик для повышения лаконичности кода.

Оглавление

[Тип данных: число.](#)

[Целые \(int\)](#)

[Вещественные \(float\)](#)

[Комплексные \(complex\)](#)

[Тип данных: строка](#)

[Конкатенация \(сцепление\)](#)

[Взятие элемента по индексу](#)

[Извлечение среза](#)

[Механизмы реверса строк](#)

[Таблица методов строк](#)

[Тип данных: список](#)

[Тип данных: кортеж](#)

[Тип данных: множество](#)

[Тип данных: словарь](#)

[Тип данных: bool](#)

[Тип данных: bytes и bytearray](#)

[Тип данных: NoneType](#)

[Тип данных: исключение](#)

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

[Понятие тернарного оператора](#)

[Оператор is](#)

[Десятка лучших трюков в Python](#)

[Объединение списков без цикла](#)

[Удаление дубликатов в списке](#)

[Обмен значениями через кортежи](#)

[Вывод значения несуществующего ключа в словаре](#)

[Поиск самых часто встречающихся элементов списка](#)

[Распаковка последовательностей при неизвестном количестве элементов](#)

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

[Сортировка словаря по значениям](#)

[Нумерованные списки](#)

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 разные 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	<code>print(abs(-6)) -> 6</code>
Побитовое И	<code>print(4 & 6) -> 4</code>
Побитовое ИЛИ	<code>print(4 6) -> 6</code>
Побитовое исключающее ИЛИ	<code>print(4 ^ 6) -> 2</code>
Битовый сдвиг влево	<code>print(4 << 6) -> 256</code>
Битовый сдвиг вправо	<code>print(4 >> 6) -> 0</code>

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)	<pre>print(int(17.5)) -> 17 print(int('10001', 2)) -> 17</pre>
bin()	Преобразовать к двоичному формату	<pre>print(bin(17)) -> 0b10001</pre>
oct()	Преобразовать к восьмеричному формату	<pre>print(oct(17)) -> 0o21</pre>
hex()	Преобразовать к шестнадцатеричному формату	<pre>print(hex(17)) -> 0x11</pre>

Вещественные (float)

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

Комплексные (complex)

Под комплексным числом понимается выражение вида $a + ib$, где a и b — любые действительные числа, 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:

```
string = "abrakadbra"
str_reverse = string[::-1]
print(str_reverse)
```

Результат:

```
arbdakarba
```

2. Обратная итерация.

Пример:

```
for el in reversed("abrakadbra"):  
    print(el)
```

Результат:

```
a  
r  
b  
d  
a  
k  
a  
r  
b  
a
```

3. Реверс на месте.

Пример:

```
string = "abrakadabra"  
str_reverse = ''  
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  
str_reverse = ''.join(symbols)  
print(str_reverse)
```

Результат:

Таблица методов строк

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

Функция	Описание	Пример
<code>len(строка)</code>	Возвращает длину строки	<pre>print(len("my_string")) -> 9</pre>
<code>строка.split(<разделитель>)</code>	Разбить строку по разделителю	<pre>print("раз два три".split()) -> ['раз', 'два', 'три'] print("четыре_пять_шесть".split('_')) -> ['четыре', 'пять', 'шесть']</pre>
<code><разделитель>.join(список)</code>	Собрать строку из списка с указанным разделителем	<pre>print('_'.join(['раз', 'два', 'три'])) -> раз_два_три print('').join(['раз', 'два', 'три']) -> раздватри</pre>
<code>строка.title()</code>	Перевести первую букву каждого слова в верхний регистр, остальные - в нижний	<pre>print("ехал грека через реку".title()) -> Ехал Грека Через Реку</pre>
<code>строка.upper()</code>	Преобразовать строку к верхнему регистру	<pre>print('простая строка'.upper()) -> ПРОСТАЯ СТРОКА</pre>
<code>строка.lower()</code>	Преобразовать строку к нижнему регистру	<pre>print('ПРОСТАЯ СТРОКА'.lower()) -> простая строка</pre>
<code>строка.istitle()</code>	Проверить, начинаются ли слова строки с буквы в верхнем регистре	<pre>print('Ехал Грека Через Реку'.istitle()) -> True print('Ехал Грека Через реку'.istitle()) -> False</pre>

<code>строка.isupper()</code>	Проверить, состоит ли строка из символов в верхнем регистре	<pre>print('ПРОСТАЯ СТРОКА'.isupper()) -> True print('простая строка'.isupper()) -> False</pre>
<code>строка.islower()</code>	Проверить, состоит ли строка из символов в нижнем регистре	<pre>print('простая строка'.islower()) -> True print('ПРОСТАЯ СТРОКА'.islower()) -> False</pre>
<code>ord(символ)</code>	Получить ASCII-код для символа	<pre>print(ord('b')) -> 98</pre>
<code>chr(код)</code>	Получить символ по ASCII-коду	<pre>print(chr(98)) -> 'b'</pre>
<code>строка.count(подстрока, [начало], [конец])</code>	Вернуть количество вхождений подстроки в строку	<pre>print('парара'.count('pa')) -> 3 print('парара'.count('pa', 2, 4)) -> 1</pre>
<code>строка.capitalize()</code>	Перевести первый символ строки в верхний регистр, остальные - в нижний	<pre>print('СТРОКА'.capitalize()) -> Строка</pre>
<code>строка.startswith(шаблон)</code>	Проверить, начинается ли строка с шаблона	<pre>print('парара'.startswith('pa')) -> True print('парара'.startswith('не')) -> False</pre>
<code>строка.endswith(шаблон)</code>	Проверить, заканчивается ли строка шаблоном	<pre>print('парара'.endswith('pa')) -> True print('парара'.endswith('не')) -> False</pre>
<code>строка.replace(шаблон,</code>	Заменить в строке	<pre>print('парара'.replace('pa',</pre>

замена)	шаблон на указанную подстроку	'не')) -> 'ненене'
строка.index(подстрока, [начало], [конец])	Найти подстроку в строке. Получить позицию первого вхождения или получить ValueError	<pre>print('парарара'.index('ра')) -> 0 print('парарара'.index('ра', 4, 6)) -> 4 print('парарара'.index('ра', 10, 20)) -> ValueError: substring not found</pre>
строка.find(подстрока, [начало], [конец])	Найти подстроку в строке. Получить позицию первого вхождения или получить -1	<pre>print('парарара'.find('ра')) -> 0 print('парарара'.find('ра', 4, 6)) -> 4 print('парарара'.find('ра', 10, 20)) -> -1</pre>

С другими методами строк вы можете ознакомиться по [ссылке](#).

Тип данных: список

В Python массивов как таковых нет. 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 приведён в таблице ниже:

Метод	Назначение
<code>result_list.append(el)</code>	Добавить элемент <code>el</code> в конец списка <code>result_list</code>
<code>result_list.extend(my_list)</code>	Расширить список <code>result_list</code> — добавить в конец элементы списка <code>my_list</code>
<code>result_list.insert(pos, el)</code>	Разместить на позиции <code>pos</code> (индекс элемента списка) элемент <code>el</code>
<code>result_list.remove(el)</code>	Удалить из списка первый элемент со значением <code>el</code>
<code>result_list.pop(pos)</code>	Удалить элемент с индексом <code>pos</code>
<code>result_list.index(el)</code>	Получить позицию (индекс) первого элемента со значением <code>el</code>
<code>result_list.count(el)</code>	Возвращает количество элементов списка со значением <code>el</code>
<code>result_list.sort([key=функция])</code>	Выполнить сортировку списка на основе указанной функции
<code>result_list.reverse()</code>	Выполнить реверс списка (развернуть список)
<code>result_list.copy()</code>	Вернуть копию списка
<code>result_list.clear()</code>	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
```

Здесь 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 примере используется оператор идентичности (**is**). В первом случае результат — значение **False**, так как переменные **list_1** и **list_2** ссылаются на разные объекты. Во втором случае получается значение **True**, так как 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 кортежей:

- защищают от неверных действий пользователя. Кортеж — неизменяемый список, защищён 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

Это контейнер с неповторяющимися 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 приведён в таблице ниже:

Метод	Назначение
<code>.add(el)</code>	Добавить элемент в множество
<code>.remove(el)</code>	Удалить элемент из множества. Если элемент отсутствует — ошибка <code>KeyError</code>
<code>.discard(el)</code>	Удалить элемент из множества
<code>.pop()</code>	Удалить первый элемент из множества. Множества не упорядочены, поэтому первый элемент множества заранее не определён

<code>.copy()</code>	Создать копию множества
<code>.clear()</code>	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()**) — аналогия 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 приводится в таблице ниже:

Метод	Назначение
<code>.keys()</code>	Возвращает список ключей словаря
<code>.values()</code>	Возвращает список значений
<code>.items()</code>	Возвращает список кортежей (ключ, значение)
<code>.get(key)</code>	Возвращает значение, соответствующее ключу <code>key</code> . Если ключ отсутствует, возвращает значение <code>None</code>

<code>.popitem()</code>	Удаляет элемент словаря и возвращает пару (ключ, значение). Если элементы отсутствуют, возникает исключение <code>KeyError</code>
<code>.setdefault(key)</code>	Возвращает значение, соответствующее ключу. Если ключ отсутствует, создаётся элемент с указанным ключом и значением <code>None</code>
<code>.pop(key)</code>	Удаляет ключ и возвращает значение, соответствующее ключу
<code>.update(new_dict)</code>	Добавляет пары (ключ, значение) в текущий словарь из словаря <code>new_dict</code> . Имеющиеся ключи перезаписываются
<code>.copy()</code>	Возвращает копию словаря
<code>.clear()</code>	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 — единица 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 изменяемый массив байтов.

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. Оно обозначает «здесь нет значения». Присвоение переменной такого значения — один из вариантов её сброса в пустое состояние. Python — язык объектно-ориентированный. **None** также принадлежит к объектам и обладает своим типом.

```
print(type(None))
```

Результат:

```
<class 'NoneType'>
```

Рассмотрим ещё 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 определён параметр: {el}")
```

Результат:

```
Для сотрудника пока не определён параметр: position
```

Здесь выполняется перебор ключей словаря и проверка, есть ли в словаре значения типа **None**.

Тип данных: исключение

Exceptions представляют собой ещё 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 этом 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 рассматриваться далее.

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**. Эта 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
```

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

Есть и другой вариант использования этого 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
```

Это работоспособный механизм в Python, 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

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 он не идентичен оператору **==**.

== — проверка равенства значений двух объектов.

is — проверка идентичности объектов, то есть 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 использовать оператор **is**.

Пример:

```
obj_1 = None
print(obj_1 is None)
```

Результат:

```
True
```

Десятка лучших трюков в Python

В завершение урока познакомимся с набором интересных приёмов, 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]
```

Поиск уникальных элементов в списке

Это очень популярный трюк, предполагающий трансформацию списка во множество и поиск уникальных элементов.

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'
```

Чтобы избежать такую ситуацию, 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()**. Она 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. Она содержит все не присвоенные 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 можно реализовать сортировку 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 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 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. Значениями обмениваются элементы с индексами 0 и 1, 2 и 3 и т. д. При нечётном количестве элементов последний сохранить на своём месте. Для заполнения списка элементов нужно 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 dict.
4. Пользователь вводит строку из нескольких слов, разделённых пробелами. Вывести каждое слово с новой строки. Строки нужно пронумеровать. Если слово длинное, выводить только первые 10 букв в слове.
5. Реализовать структуру «Рейтинг», представляющую собой набор натуральных чисел, который не возрастает. У пользователя нужно 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 ввёл число 3. Результат: 7, 5, 3, 3, **3**, 2.

Пользователь ввёл число 8. Результат: **8**, 7, 5, 3, 3, 2.

Пользователь ввёл число 1. Результат: 7, 5, 3, 3, 2, **1**.

Набор натуральных чисел можно задать сразу в коде, например, `my_list = [7, 5, 3, 3, 2]`.

6. *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, то есть характеристиками товара:

название, цена, количество, единица измерения. Структуру нужно сформировать программно, запросив все данные у пользователя.

Пример готовой структуры:

```
[
    (1, {"название": "компьютер", "цена": 20000, "количество": 5, "ед": "шт."}),
    (2, {"название": "принтер", "цена": 6000, "количество": 2, "ед": "шт."}),
    (3, {"название": "сканер", "цена": 2000, "количество": 7, "ед": "шт."})
]
```

Нужно собрать аналитику о товарах. Реализовать словарь, в котором каждый ключ — характеристика товара, например, название. Тогда 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.](#)
2. [Переменные и типы данных.](#)
3. [Типы данных в Python 3.](#)

Используемая литература

Для подготовки методического пособия были использованы следующие ресурсы:

1. [Язык программирования Python 3 для начинающих и чайников.](#)
2. [Программирование в Python.](#)
3. [Учим Python качественно \(habr\).](#)
4. [Самоучитель по Python.](#)
5. [Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 \(4-е издание\).](#)