KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 1 урок > Основы программирования Python

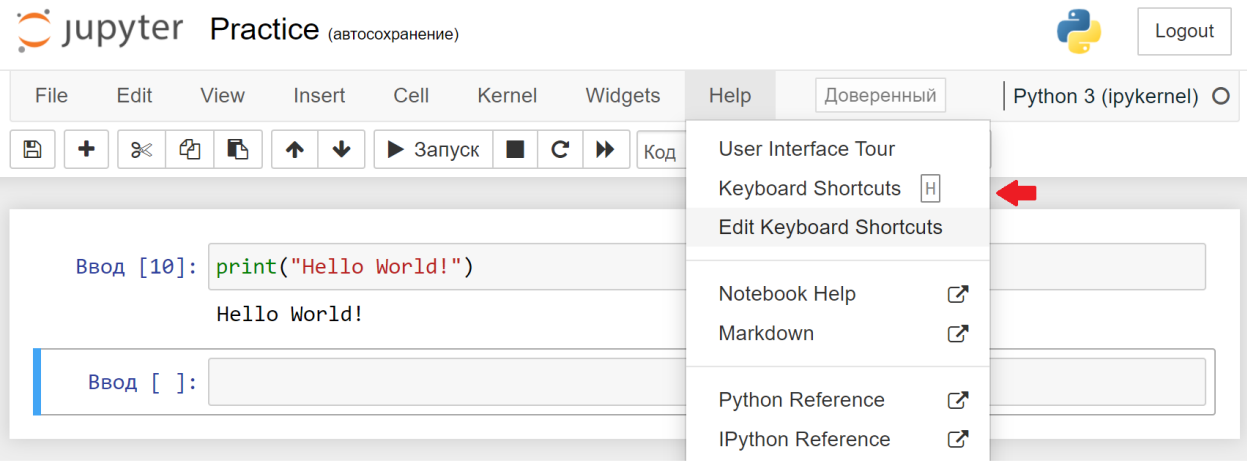
> Оглавление

> Горячие клавиши Jupyter Notebook

Jupyter Notebook — среда, в которой мы будем выполнять написанный код, не устанавливая Python к себе на компьютер. Для начала давайте ознакомимся с основными концепциями данного инструмента.

Важно знать, что код в Jupyter-ноутбуке разбивается по **ячейкам** — неделимым кусочкам, которые будут выполняться целиком. Ячейки дают возможность сразу увидеть результат выполнения — этим они и удобны.

Горячие клавиши призваны значительно упростить жизнь и ускорить работу.



В самом ноутбуке список горячих клавиш можно найти в Help → Keyboard Shortcuts

Действия с ячейками

Для того, чтобы выполнить приведенные ниже сочетания, необходимо сначала "выделить ячейку", то есть кликнуть по ней вне поля для ввода, например, чуть левее. Если заметите, что индикатор подсветки у ячейки изменился, значит, вы все сделали правильно.

Создать ячейку ниже: `b`

Создать ячейку выше: `a`

Вырезать ячейки: `x`

Копировать ячейки: `c`

Вставить копированные ячейки: `v`

Выделить несколько ячеек подряд: `Shift + вверх` или `Shift + вниз`

Объединить выделенные ячейки: `Shift + m`

Выполнить ячейки:

`Ctrl + Enter` — выполнение ячейки с сохранением фокуса

`Shift + Enter` — выполнение ячейки с переводом фокуса на следующую

Для того, чтобы выполнить одну конкретную ячейку, в которой вы работаете сейчас, выделять ее отдельно необязательно. Попробуйте :)

Другие полезные сочетания

Показ документации для объекта: `Shift + Tab`

Комментирование строки: `Ctrl + /`

Больше информации тут

> Переменные

Первое, с чем мы познакомимся, как ни странно, это комментарии.

Комментарий — это строка, которая служит только для информативных целей, ее выполнение Python осуществлять не будет. Комментарии в Python очень полезны, с их помощью программисты добавляют пояснения к своим программам, облегчая читаемость кода. Мы часто будем использовать их в конспектах.

Для того, чтобы объявить комментарий, нужно использовать знак решетки — `#`.

```
# Это комментарий. Питон игнорирует все, что написано после симовола # до конца строки
# А редактор это подсвечивает серым
```

Переменная в Python — это именованная область в памяти компьютера, куда кладутся значения, которые затем можно будет доставать, изменять и проводить другие манипуляции. Грубо говоря, это ярлык с именем, который привязан к значению.

В левой части указывается название переменной, а в правой части значение, которое было ей присвоено.

```
var = 5
p = 3.1415926
h = "Hello"
```

Важная особенность переменных, которая оправдывает название термина — они могут менять свое значение.

```
# Записываем в переменную var число 5
var = 5
print(var)
```

```
Output:
5

# Меняем значение переменной на новое
var = 3.1415926
print(var)

Output:
3.1415926
```

Python также позволяет присваивать несколько значений нескольким переменным в рамках одной строки. Это позволяет сократить код и не потерять в его читаемости.

```
var, p, h = 5, 3.1415926, "Hello"
print(var)
print(p)
print(h)

Output:
5
3.1415926
Hello
```

В Python существует универсальный способ обмена переменных значениями.

```
p = 3.1415926
h = "Hello"

(p, h) = (h, p)
print(p)
print(h)

Output:
Hello
3.1415926
```

Зачем нужны переменные?

У переменных есть два основных преимущества:

1. **Переменные делают логику независимой от чисел.**

Мы можем описать какую-либо логику преобразования данных над переменными, а затем подставить в них значения и получить готовый результат. Этим приемом часто пользуются математики, когда выводят формулы.

2. **Переменные делают преобразования данных читаемыми.**

Приведем пример. Представим, что мы хотим подсчитать доход через 5 лет по банковскому вкладу в разных банках. У нас есть 100 рублей и 2 банка с разными процентами:

1. Банк "Весна" дает 3% годовых.
2. Банк "Осень" дает 2% годовых, но более надежен.

Посчитаем, сколько будет денег через 5 лет:

```
initial_sum = 100
interest_rate = 3
years = 5

# Из кода сразу ясно, как считается сумма в конце для любых входных данных
end_sum = initial_sum * ((100 + interest_rate) / 100) ** years
end_sum

Output:
115.92740743
```

Можно легко пересчитать все под другой процент, поменяв только вторую строчку — переменную, которая задает `interest_rate`.

Правила использования

- При присвоении название переменной всегда находится слева, а данные справа
- Переменные в Python принято задавать маленькими буквами
- В названии недопустимо использование пробелов, вместо них — нижние подчёркивания
- Имя переменной не может начинаться с цифры
- Лучше называть переменные так, чтобы по названию было понятно что в них лежит

- Имена переменных могут состоять только из букв, цифр и нижнего подчёркивания

Выбирая имя для переменной, следует также учитывать регистр: `PYTHON_language`, `PYTHON_LANGUAGE` и `Python_Language` – не одна, а три разные переменные.

Для примера рассмотрим правильные и неправильные имена переменных:

Неправильные	**Правильные**
python language	python_language
8pep	pep8
kdxffd	int
My_Var	my_var
!&python&	python

Другие рекомендации по оформлению кода в Python

> Базовые операции Python

Присваивание значений

Самый главный оператор в Python — оператор **присваивания**, он обозначается знаком `=`. Данный оператор связывает значение с некоторым именем *переменной.*

```
# Сейчас мы попросим записать в переменную с именем "a" значение 5
a = 5
```

Сравнение значений

Python имеет ряд привычных операторов **сравнения**:

- Равно — `==` *(обратите внимание на двойной знак)* и Не равно — `!=`
- Меньше — `<` и Меньше или равно — `<=`
- Больше — `>` и Больше или равно — `>=`

Арифметические операции

Реализован также и ряд привычных **арифметических операторов**:

- Сложение — `+`
- Вычитание — `-`
- Умножение — `*`
- Возведение в степень — `**`
- Деление — `/`
- Целочисленное деление — `//`
- Остаток от деления — `%`

Арифметические операции можно производить как с числами:

```
5 + 7

Output:
12
```

Так и с участием переменных (переменную "a" мы задали выше с помощью оператора присваивания):

```
11 * a

Output:
55
```

Программируя на Python, вы обязательно почувствуете преимущества **сокращенной записи** арифметических операций с присваиванием. Подобные конструкции используются очень часто и делают процесс написания кода более приятным, за что получили название "синтаксический сахар".

Умножение с присваиванием:

```
# Эта запись идентична записи a = a * 5
a *= 5
```

Деление с присваиванием:

```
# Эта запись идентична записи a = a / 5
a /= 5
```

Сложение с присваиванием:

```
# Эта запись идентична записи a = a + 5
a += 5
```

Вычитание с присваиванием:

```
# Эта запись идентична записи a = a - 5
a -= 5
```

Более продвинутые операторы Python мы рассмотрим в этом уроке, когда познакомимся с типами данных.

> Типы данных

В языке Python есть несколько стандартных типов данных. Они классифицируются по важному признаку — их **изменяемости**. Необходимо понимать особенности каждого типа данных, чтобы грамотно использовать их для решения тех или иных задач.

Для того, чтобы определить, какой тип данных у переменной, можно вызвать функцию `type()`.

Неизменяемые типы данных:

- **Строки** (*англ. "string"*, обозначаются как `str`)

```
my_string = "Hello"
type(my_string)

Output:
str
```

- **Целые числа** (*англ. "integer"*, обозначаются как `int`)

```
number = 3
type(number)

Output:
int
```

- **Числа с плавающей точкой** (*англ. "float"*, обозначаются как `float`)

```
float_number = 3.141592
type(float_number)

Output:
float
```

- **Кортежи** (*англ. "tuple"*, обозначаются как `tuple`)

```
my_tuple = ("Strings", "Numbers", "Tuples", "Boolean")
type(my_tuple)

Output:
tuple
```

- **Логический тип данных** (*англ. "boolean"*, обозначаются как `bool`)

```
logical_val = True
type(logical_val)

Output:
bool

logical_val = False
type(logical_val)

Output:
bool
```

Изменяемые типы данных:

- **Списки** (*англ. "list"*, обозначаются как `list`)

```
my_list = ["Strings", "Numbers", "Tuples", "Boolean"]
type(my_list)

Output:
list
```

- **Словари** (*англ. "dictionary"*, обозначаются как `dict`)

```
my_dict = {
"Lists": "mutable",
"Strings": "immutable",
"Numbers": "immutable"
}

type(my_dict)

Output:
dict
```

- **Множества** (*англ. "set"*, обозначаются как `set`)

```
my_set = {"Strings", "Numbers", "Tuples", "Boolean"}
type(my_set)

Output:
set
```

> Операторы: продолжение

Помимо базовых, рассмотрим также и другие важные операторы Python.

Логические операторы

В языке Python есть три логических оператора, которые объединяют два или несколько логических выражения и возвращают `True` или `False`. Они интуитивно понятны, потому что используются здесь так же, как и в естественном языке.

- `and`

- `or`
- `not`

Оператор and

Оператор `and` будет возвращать `True`, если в выражении **И** условие слева **И** условие справа истинно. Рассмотрим на примерах:

```
# И слева и справа условие истинно
2 * 2 == 4 and 5 > 4:

Output:
True

# Условие слева ложно, справа истинно
2 * 2 == 5 and 5 > 4:

Output:
False

# Слева истинно, справа ложно
True and False

Output:
False
```

Оператор or

Этот оператор будет возвращать `True`, если **хотя бы одно** из условий слева или справа истинное, и возвращать `False`, если оба условия ложные:

```
# Условие слева истинно, условие справа ложно
4 == 4 or type(3.14) == int

Output:
True

# Оба ложны
False or False

Output:
False

# Одно ложно, другое истинно
False or True
```

```
Output:
True
```

Оператор not

Этот унарный *(т. е. применяется только к одному выражению, стоящему справа от него)* оператор изменяет логическое значение на противоположное, **инвертирует** его. Например:

```
not True

Output:
False

not False

Output:
True

# Чтобы разобраться почему так, важно знать приоритетность операторов
False or not False

Output:
True
```

Приоритетность логических операторов

Рассмотренные логические операторы выполняются в Python со следующей приоритетностью в порядке от высокой до низкой:

1. Скобки `()`
2. `not`
3. `and`
4. `or`

Разберем, что вернет сложное выражение:

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

# Раскручиваем по порядку приоритетности:
1. Самый высокий приоритет имеют скобки, поэтому разбираем выражение not False and False
2. not False вернет True
3. Выражение в скобках упростилось до True and False и вернет False
```

```
4. Далее оператор not, теперь мы имеем выражение not False, которое вернет True
5. Осталось простое True or True, оно вернет True

Output:
True
```

Операторы принадлежности

Операторы принадлежности существуют для проверки, является ли значение частью определенной последовательности. В Python есть два таких оператора:

- `in`
- `not in`

Оператор in

Данный оператор возвращает логическое значение `True` или `False` в зависимости от того присутствует ли значение в последовательности. Приведем пример:

```
# У нас есть список следующего вида:
types = ["Strings", "Numbers", "Tuples", "Boolean"]

# Проверяем, есть ли значение "Text" в списке types
"Text" in types

Output:
False

# Проверяем, есть ли значение "Numbers" в списке types
"Numbers" in types

Output:
True

# Проверяем, если ли "um" в строке "Numbers"
"um" in "Numbers"

Output:
True
```

Оператор not in

Этот оператор, напротив, возвращает `True`, если значения в последовательности нет. Если же значение присутствует в последовательности — возвращается `False`.

```
types = ["Strings", "Numbers", "Tuples", "Boolean"]

"Text" not in types

Output:
True

"Tuples" not in types

Output:
False
```

Больше про операторы принадлежности

> Коллекции: Строки

Строка (string) — это неизменяемый упорядоченный тип данных, представляющий собой последовательность символов, заключенную в кавычки `""`.

Кавычки могут быть как одинарными, так и двойными. Выбор за вами, однако при написании программ принято выбирать что-то одно, чтобы соблюдать единство стиля.

```
# Строки заключаются в кавычки
language = "Python" # Двойные
data_type = 'String' # Или одинарные — разницы никакой
print(language)
print(data_type)

Output:
Python
String
```

Можно суммировать несколько строк, тогда они объединяются в одну строку:

```
language = "Python" + "Language"
print(language)

Output:
PythonLanguage

# Можно заключать пробельный символ в отдельные кавычки
```

```
my_phrase = "Enjoy" + " " + "learning"
print(my_phrase)

Output:
Enjoy learning
```

Строку также можно умножать на число. В этом случае строка повторяется указанное количество раз:

```
language = "Python" * 3
print(language)

Output:
PythonPythonPython
```

Длину строки можно узнать с помощью функции `len()`. Она возвращает количество символов в строке:

```
language = "Python" + "Language"
print(len(language))

Output:
14
```

Форматирование строк

Метод .format()

Бывают ситуации, когда нам было бы крайне полезно иметь некоторую матрицу (template) строки, в которую можно подставлять произвольную подстроку.

Кавычки `{}` внутри строки — места, в которые можно вставить любую подстроку.

```
template = "Wake up, {}. The {} has you"
```

Подставляем значения в наш template с помощью метода `.format()`. Переданные в него аргументы пойдут в соответствующие скобки.

```
template.format('Neo', 'Matrix')

Output:
'Wake up, Neo. The Matrix has you'
```

f-строки

f-строки — буквально означают «formatted string» и являются еще одним удобным способом форматирования строк, очень похожим на применение предыдущего метода. Они задаются с помощью литерала «f» перед кавычками и работают по следующему принципу: берут значения переменных, которые есть в текущей области видимости, и подставляют их в строку.

```
name = "Neo"
reality = "Matrix"

f"Wake up, {name}. The {reality} has you"

Output:
'Wake up, Neo. The Matrix has you'
```

Больше способов форматирования строк

Больше о типе данных string

> Коллекции: Списки

Список (list) — это упорядоченная и изменяемая последовательность, позволяющая хранить внутри себя значения различных типов. Для создания списка в Python необходимо перечислить значения в квадратных скобках `[]`.

Создание списка:

```
learning_methods = ["Supervised", "Unsupervised", "Semi-Supervised", "Reinforcement"]
```

Индексирование

К элементам списка можно обращаться с помощью **индексов**.

Например, для того, чтобы достать определенное значение из списка, необходимо указать в квадратных скобках номер нужного элемента. При этом крайне важно знать и всегда помнить о том, что нумерация в Python начинается с нуля.

Выведем различные элементы списка по индексам:

```
learning_methods = ["Supervised", "Unsupervised", "Semi-Supervised", "Reinforcement"]

# Первый элемент списка имеет нулевой индекс
learning_methods[0]

Output:
'Supervised'

# Третий элемент списка имеет индекс 2
learning_methods[2]

Output:
'Semi-Supervised'

# Последний элемент списка - он же первый с конца, всегда имеет индекс -1
learning_methods[-1]

Output:
'Reinforcement'
```

Срезы / слайсы

С помощью индексов, можно делать **срезы** списков. Синтаксис среза следующий: `[start : stop : step]`

`start` - по умолчанию равен 0

`stop` - по умолчанию равен длине списка

`step` - по умолчанию равен 1

```
# От элемента с индексом 2 до конца
learning_methods[2:]

Output:
['Semi-Supervised', 'Reinforcement']

# От начала до конца с шагом 2
learning_methods[::2]
```

```
Output:
['Supervised', 'Semi-Supervised']
```

Методы для списков

У Python есть набор встроенных методов, которые вы можете использовать при работе со списками. Некоторые из них:

- `append()` — добавляет элемент в конец списка
- `insert()` — добавляет элемент по индексу
- `pop()` — удаляет элемент по индексу или последний
- `remove()` — удаляет элементы по значению
- `reverse()` — разворачивает список
- `sort()` — сортирует список
- `copy()` — возвращает копию списка

Рассмотрим примеры применения методов.

Изменим список, добавив элементы в конец, используя метод `.append()`:

```
ml_techniques = ["Regression", "Classification"]

ml_techniques.append("Clustering")
print(ml_techniques)

Output:
['Regression', 'Classification', 'Clustering']
```

Существует несколько методов удаления элементов списка.
Метод `.remove()` удаляет переданный ему элемент из списка. При этом удаляется только первый обнаруженный в списке элемент.

```
ml_techniques = ["Regression", "Classification", "Clustering", "Decision trees"]

ml_techniques.remove("Classification")
print(ml_techniques)

Output:
['Regression', 'Clustering', 'Decision trees']
```

Метод `.pop()` удаляет элемент по его индексу (или последний элемент, если индекс не указан) и возвращает его.

```
ml_techniques = ["Regression", "Classification", "Clustering", "Decision trees"]

ml_techniques.pop(0)

Output:
'Regression'

# Элемент удалился из исходного списка
ml_techniques

Output:
['Classification', 'Clustering', 'Decision trees']
```

Ключевое слово `del` удаляет определенный индекс (также `del` может полностью удалить список).

```
ml_techniques = ["Regression", "Classification", "Clustering", "Decision trees"]

del ml_techniques[3]
print(ml_techniques)

Output:
['Regression', 'Classification', 'Clustering']
```

Чтобы определить сколько элементов списка у вас есть, можно воспользоваться уже знакомой функцией `len()` . Выведем число элементов в списке:

```
ml_techniques = ["Regression", "Classification", "Clustering", "Decision trees"]

len(ml_techniques)

Output:
4
```

Когда применять списки

- Списки стоит применять, когда мы хотим класть много данных в одно логическое место. Например, список заказов одного клиента.

- Если хотим добавлять элементы, следить за их взаимным порядком и удалять — самое то.

Больше информации здесь и в этом источнике

> Коллекции: Кортежи и Множества

Кортежи

Кортеж (tuple) — это упорядоченная неизменяемая коллекция. Кортеж обозначается в Python круглыми скобками `()`.

Эта структура данных похожа на список, однако у нее уже нет присущих спискам методов `.append()` и `.remove()`, т. е. в существующий кортеж не получится положить новый элемент или же удалить имеющийся. Все потому, что объект `tuple`, будучи созданным, не может быть изменен.

Создание кортежа:

```
supervised_learning = ("Classification", "Regression")

# Кортеж из одного элемента можно создать следующим образом:
# Внимание на запятую, не поставив которую, вы получите не кортеж, а строку!
my_tuple = ('Python',)
```

Кортежи можно складывать, как и списки. Но, в отличии от списков, при сложении создается **новый** кортеж, куда копируется сначала первый, а потом второй кортеж.

```
my_tuple = (1, 2, 3)

(5, 5) + my_tuple

Output:
(5, 5, 1, 2, 3)
```

Когда применять кортежи

Кортежи применяют когда требуются удобства списков, но в то же время важна неизменяемость данных. Подробнее об этом поговорим во втором уроке.

Множества

Множество (set) — неупорядоченная и не индексируемая коллекция. Она особенна тем, что хранит только уникальные элементы. Множества обозначаются фигурными скобками `{}`.

Так как эти коллекции не упорядочены, элементы множеств будут отображаться в произвольном порядке.

Создание множества (перечисляем элементы в фигурных скобках):

```
types = {"String", "Numbers", "Tuples", "Boolean"}
types

Output:
{'Boolean', 'Numbers', 'String', 'Tuples'} # Порядок не соответствует изначальному
```

По этой же причине не получится обратиться к элементу множества по индексу:

```
types[0]

Output:
TypeError: 'set' object is not subscriptable
```

Но что насчет создания пустого множества? Кажется логичным попробовать создать его следующим образом:

```
my_non_set = {}
```

Однако не все так просто. Если мы узнаем тип данных у созданной переменной, то получим ответ — словарь.

```
type(my_non_set)

Output:
dict
```

Поэтому правильно создать пустое множество можно с помощью функции `set()`:

```
my_set = set()
type(my_set)

Output:
set
```

Частый кейс использования множеств: так как множества могут хранить только уникальные элементы, мы можем легко исключить все дубли из списка элементов, преобразовав его в `set`.

```
types = ["String", "Numbers", "Tuples", "Boolean", "Boolean"]

set(types)

Output:
{'Boolean', 'Numbers', 'String', 'Tuples'}
```

В множество можно добавлять только неизменяемые объекты! Это связано с его внутренним строением. Так, создать множество вида `{2, "hello", []}` не получится, т. к. третий элемент имеет тип `list`, а он изменяемый.

Объекты типа `set` близки к "множествам" из математики. Можно брать пересечения множеств, объединение, симметричную разность:

- `a.intersection(b)` — возвращает множество, являющееся пересечением `a` и `b`
- `a.union(b)` — объединяет множества `a` и `b`
- `a.symmetric_difference(b)` — возвращает множество элементов, которые не пересекаются в `a` и `b`

```
a = {'a', (1, 2), 3}
b = {3, (1, 2), 'unique'}

a.intersection(b)
Output:
{(1, 2), 3}

a.union(b)
Output:
{(1, 2), 3, 'a', 'unique'}

a.symmetric_difference(b)
```

```
Output:
{'a', 'unique'}

a - b
Output:
{'a'}

b - a
Output:
{'unique'}
```

Когда применять множества

- Если нужно получить только уникальные элементы и порядок элементов не важен. Самый частый на практике кейс.
- Если нужно узнать пересечение и прочие операции над множествами.

Помимо обычных множеств существует также тип **frozenset**. Объекты этого типа не могут быть изменены после создания, поэтому они могут использоваться как ключ словаря или как элементы другого множества.

Больше информации про множества и про тип frozenset

> Коллекции: Словари

Словарь (dictionary) — ассоциативный тип данных, где каждый элемент является парой ключ-значение. Для создания словаря необходимо указать элементы внутри фигурных скобок `{}`.

Создадим словарь:

```
# Пустой словарь
name_to_number = {}
```

Ключами в словаре могут быть любые неизменяемые объекты (строка, число, кортеж).

```
# Синтаксис элементов в словаре – {key: value}
name_to_number = {
    'Алексей': '+7 123 123-12-34',
```

```
    'Никита': '+43 321-32-10'
}
```

Вы можете получить доступ к значению словаря по ключу:

```
name_to_number['Алексей']

Output:
'+7 123 123-12-34'

# При этом, если ключа нет, мы получим ошибку
name_to_number['Николай']

Output:
KeyError: 'Николай'
```

Существует также метод под названием `get()`, который даст вам тот же результат:

```
name_to_number.get('Алексей')

Output:
'+7 123 123-12-34'

# Можно попросить возвращать некое значение по умолчанию, если ключ не найден
# Выведет "no info", т.к. ключа "Мария" нет в словаре
name_to_number.get('Мария', 'no info')

Output:
no info
```

В словарь легко добавлять элементы через оператор присваивания:

```
name_to_number['Владимир'] = '+1 111 111-11-11'
name_to_number

Output:
{'Алексей': '+7 123 123-12-34',
 'Никита': '+43 321-32-10',
 'Владимир': '+1 111 111-11-11'}
```

Через оператор `in` можно узнать, есть ли ключ в словаре:

```
'Владимир' in name_to_number

Output:
True
```

В Python существует набор встроенных методов, с помощью которых вы можете работать со словарём:

- `clear()` — удаляет все элементы из словаря
- `get()` — возвращает значение по ключу
- `keys()` — возвращает список, содержащий ключи словаря
- `pop()` — удаляет элементы по ключу
- `values()` — возвращает список всех значений в словаре

Больше информации

> Циклы

Отлично, мы можем хранить много объектов в списках, хранить соответствия между объектами в словаре. Но вместе с этим приходит вопрос: как их обрабатывать? Писать код на обработку каждого элемента — занятие неблагодарное. К счастью, в Python есть **циклы**, которые позволяют применять набор операций к каждому элементу из списка.

В Python есть 2 вида циклов:

- Циклы, повторяющиеся определенное количество раз (цикл `for`)
- Циклы, повторяющиеся до наступления определенного события (цикл `while`).

Цикл for

for — пробегает по всем элементам и выполняет набор операций. При помощи этого цикла можно пробегаться *(итерироваться)* по любому объекту-коллекции.

Цикл пишется по простому синтаксису:

for один_элемент in список_или_tuple_или_set:
    операция_1

операция_2

В `один_элемент` на каждом прогоне (итерации) один за другим будут записываться объекты из списка.

Обратите внимание на двоеточие в конце строки (`:`) и отбивку **табуляцией** (или 4-мя пробелами) — это требования синтаксиса.

Через отбивку пробелами Python понимает, какие команды нужно повторять в цикле, а какие нет. При этом команды, которые цикл будет повторять раз за разом, называются телом цикла. Двоеточие в конце строки с `for` сообщает Python, что дальше находится тело цикла.

Рассмотрим пример:

```
for a in [9, 16, 25]:
    print('считаем в цикле')
    print(a ** 0.5 + 1)
print('Пишем один раз')

Output:
считаем в цикле
4.0
считаем в цикле
5.0
считаем в цикле
6.0
Пишем один раз
```

В примере выше первые два `print` идут в цикле, а последний — нет, поэтому он исполняется только один раз. Попробуйте добавить в список `[9, 16, 25]` еще несколько значений и посмотрите, как поменяется вывод.

Давайте разберем примеры с конструкцией `range()`:

```
# range(n) сгенерирует все числа от 0 до n-1 (последний не входит!)
# range очень часто встречается на практике, на нем остановимся детальнее
for i in range(5):
    print(i)

Output:
0
1
2
3
4
```

```
# range(start, stop) сгенерирует все числа от start до stop - 1
for i in range(1, 7):
    # f"что-то" называется f-string
    # через него можно подставить значение Python-выражения прямо в строку
    print(f'Квадрат числа {i} есть {i * i}')

Output:
Квадрат числа 1 есть 1
Квадрат числа 2 есть 4
Квадрат числа 3 есть 9
Квадрат числа 4 есть 16
Квадрат числа 5 есть 25
Квадрат числа 6 есть 36
```

Используем цикл `for`, чтобы сгенерировать последовательность четных чисел:

```
# range(start, stop, step) генерирует от start до (stop - 1) с шагом step
even_numbers = []
for i in range(0, 14, 2):
    even_numbers.append(i)
print(even_numbers)

Output:
[0, 2, 4, 6, 8, 10, 12]
```

Цикл while

Если цикл `for` бегал по элементам из набора, то циклу `while` не нужны никакие списки. Он просто прогоняет код в своем теле цикла до тех пор, пока верно некое утверждение.

```
# Тело цикла (do things) выполняется, пока условие (predicate)
# после ключевого слова while — истина (True)
while predicate:
    do things
```

Пример работающего цикла:

```
i = 0
while i < 5:
```

```
    print(i)
    i = i + 1

Output:
0
1
2
3
4
```

Будьте осторожны, если вы удалите ту часть, в которой мы увеличиваем значение `i`, то мы получим бесконечный цикл:

```
# Здесь i никогда не бывает больше 5
i = 0
while i < 5:
    print(i)

Output:
0
0
0
0
...
```

Рассмотрим цикл посложнее. Представим, что нас попросили подсчитать, сколько раз число делится на 2.

С помощью оператора `==` происходит проверка на равенство: `number % 2 == 0`. Если остаток от деления на два равен нулю (проверка на четность), то вся конструкция обратится в `True` — это увидит `while` и войдет внутрь цикла. Как только `number % 2 == 0` станет `False` (т. е. остаток перестанет быть нулем), цикл `while` увидит это и воспримет как сигнал завершения — больше внутрь цикла заходить не будет. После этого выполнится `print(number)`.

```
number = 34624

# Мы заранее не можем знать, сколько раз число поделится на 2 - тут и выручает while
while number % 2 == 0:  # кстати, number % 2 == 0 - имеет булевый тип
    print('делится на 2, делим')
    number //= 2 # то же самое, что number = number // 2
print(number)

Output:
делится на 2, делим
```

```
делится на 2, делим
делится на 2, делим
делится на 2, делим
делится на 2, делим
делится на 2, делим
541
```

О функции range с примерами

> Условия

Представим ситуацию, когда на входе нам дан список балансов на счетах клиентов, и нам нужно вывести все счета с отрицательным балансом. В голову приходит сравнивать баланс с нулем, и если он меньше нуля, то записывать в ответ. Но как объяснить Python это **"если"**?

На помощь приходит оператор `if`, который позволяет выполнить определенный набор инструкций в зависимости от некоторого условия.

Синтаксис оператора `if` выглядит так:

```
if predicate:
    doing_1
    ...
    doing_n
```

- `if` — ключевое слово, даёт понять, что дальше следует условие (ветвь)
- `predicate` — выражение, которое сводится к логическому (True/False)
- `:` — необходимый элемент синтаксиса
- Отступ — 4 пробела или tab
- `doing` — команды, которые вы хотите выполнить, если попали в эту ветку (то есть `predicate True`). Все строки в одном условии должны быть с одинаковым уровнем отступа, чтобы выполняться в нём.

Посмотрим на конструкцию `if` внимательно. Ничего не напоминает? Синтаксис точно такой же, как у `while` — и это не спроста: в Python `for`, `while` и `if` являются блочными конструкциями: они формируют блоки и каким-то образом их контролируют (`for` и `while` формируют блок

повторяющихся операций, а `if` формирует блок, который выполнится при определенном условии).

Простой пример для затравки: в зависимости от четности программа выводит разные сообщения.

```
number = 4
if number % 2 == 0:
    print('число четное')
else:
    print('Число нечетное')

Output:
число четное
```

А теперь поищем счета с отрицательным балансом:

```
accounts_balance = [1482.0, 28182.12, -124.42, 85.3, -23.5, 82]  # входные данные

# сюда запишем ответ
negative_accounts = []
# проходимся по всем элементам во входных данных
for balance in accounts_balance:
    # если значение текущего элемента < 0, то идем во внутрь `if`
    if balance < 0:
        print(f'{balance} отрицателен, записываем в ответ')
        negative_accounts.append(balance)
    # выполнится в любом случае, т.к. не находится под властью `if` (отступы важны!)
    print(f'закончили обрабатывать баланс {balance}')

print(f'Отрицательные балансы: {negative_accounts}')

Output:
закончили обрабатывать баланс 1482.0
закончили обрабатывать баланс 28182.12
-124.42 отрицателен, записываем в ответ
закончили обрабатывать баланс -124.42
закончили обрабатывать баланс 85.3
-23.5 отрицателен, записываем в ответ
закончили обрабатывать баланс -23.5
закончили обрабатывать баланс 82
Отрицательные балансы: [-124.42, -23.5]
```

Помимо простого условия, которое либо выполнится, либо нет, есть более сложные — с двумя и более ветвями. Для их обозначения используются ключевые слова `elif` и `else`.

- `elif` — ключевое слово, означающее, что дальше идёт условие, которое будет выполняться, только если все предыдущие условия не выполнились (`predicate` в них был `False`) и предикат этой ветки `True`.

Для использования `elif` обязательно должна быть ветка `if` выше.

- `else` — ключевое слово, означающее, что дальше идёт условие, которое будет выполняться в том случае, если все предыдущие условия не выполнились.

Для использования `else` выше него обязательно должна быть ветка `if` или `elif`.

```
# input() - это запрос ввода от пользователя
# У вас появится внизу строка, куда можно ввести что угодно, и это запишется в `name`
# Попробуйте ввести в строку "Москва" и "Лондон" без кавычек (запустите ячейку два раза)
name = input()
if name == 'Москва':
    print('Столица России')
# else if
elif name == 'Лондон':
    print('Столица Великобритании')
else:
    print('Неизвестно')

Output:
Норвегия
Неизвестно
```

Работает следующим образом: если условие после `if` не выполнено, то идут проверяться условия после `elif` в том порядке, в каком они записаны в коде. Отбирается первое правильное — и его блок исполняется. Если ничего не нашлось, то выполняется код из `else`.

> Ключевые слова break, continue

Расскажем про два важных дополнения к `for` и `while`: это команды `break` и `continue`. Оба дают возможность выйти из цикла раньше задуманного.

Рассмотрим пример использования `break`:

```
words = ['see', 'this', 'long', 'sentence', 'here', 'no-show', 'no-show']

for i in range(len(words)):
```

```
    print(words[i])
    # если мы уже на третьем слове, досрочно выйдем из цикла
    if i > 2:
        break

Output:
see
this
long
sentence
```

Ключевое слово `continue` работает схожим образом, только при этом мы не выходим из цикла целиком, а пропускаем одну итерацию:

```
for feature in ['make sandwitch', 'make coffee', 'watch TV', 'wash plate']:
    if feature == 'make coffee':
        # Пропустим одну итерацию в цикле
        print('Coffee machine is broken, skipping :(')
        continue
    # else уже не нужен - если условие не истинно
    # то все равно выйдем на print ниже
    # а если истинно - continue гарантирует, что дальше цикл выполняться не будет
    print(f'starting action: {feature}')

Output:
starting action: make sandwitch
Coffee machine is broken, skipping :(
starting action: watch TV
starting action: wash plate
```

Для чего это нужно?

Иногда в логике цикла бывают отклонения: скажем, определенную комбинацию параметров нельзя обрабатывать циклом — тогда используем `continue`.

`break` чаще всего используется в циклах `while` как уточнение, когда останавливать цикл. Например: в машинном обучении есть алгоритм градиентного спуска, который "учит" модель на данных. Процесс "обучения" идет по шагам, и часто процесс нужно прервать, если новый шаг не привел к улучшению модели — в таких случаях можно применить `break` для выхода.