



> Конспект > 1 урок > PYTHON

> Оглавление 1 урока

1. Хоткеи юпитер ноутбука
2. Арифметика
3. Типы данных
4. Переменные
5. Сравнения
6. Операции над строками
7. Списки
8. Оформление код и комментарии
9. Использованные на уроке функции
10. Условия
11. Логические операторы
12. Циклы
13. Вложенные конструкции
14. Форматирование строк
15. Словари

> Горячие клавиши в ноутбуке

Выполнение ячейки

Ctrl + Enter

или

Shift + Enter

Комментирование строки

Ctrl + /

Создать ячейку выше

Обратите внимание, эта команда и команды ниже будут работать, если у вас выделена ячейка (она подсвечена голубым, при этом курсор не мелькает внутри ячейки)

a

Создать ячейку ниже

b

Объединить выделенные ячейки

Shift + m

Вырезать ячейки

x

Копировать ячейки

c

Вставить скопированные/вырезанные ячейки

v

[Больше информации](#)

> Арифметика

Сложение

```
2 + 2
4
```

Вычитание

```
-3 - 5
-8
```

Умножение

```
3 * 4
12
```

Возведение в степень

```
2 ** 3  
8
```

Деление

```
10 / 2  
5.0
```

Деление нацело - в одиннадцати содержится пять целых двоек

```
11 // 2  
5
```

Остаток от деления - при делении десяти нацело на два не остаётся остатка

```
10 % 2  
0
```

Скобки как в математике меняют приоритет операций

```
(12 + 4) * 9  
144
```

[Больше информации](#)

Операторы присваивания

Помимо простых арифметических операторов (+, -, *, / и т.д.) в питоне существуют операторы присваивания

Они заменяют текущее значение переменной новым значением с помощью задаваемого выражения.

Простой оператор присваивания

```
a = 5
```

Присвоит переменной `a` значение `5`

Сложение И

```
a = 4
b = 3
a += b # аналогично a = a + b
```

Присвоит переменной `a` сумму двух переменных (`a` и `b`)

```
a = 7
```

Вычитание И

```
b = 7
c = 4
b -= c # аналогично b = b - c
```

Присвоит переменной `b` разницу двух переменных (`b` и `c`)

```
b = 3
```

[Подробнее об операторах](#)

> Типы данных

Самые важные – числа и строки

- Числа записываются как почти везде – просто последовательность цифр: `0`, `5`, `100`
- Дробные числа представляются как десятичные дроби, дробная часть обособляется точкой: `10.5`, `0.0`, `300.5034`
- Строки заключены в кавычки – `'Ivan'`. Кавычки могут быть одинарными, двойными, или их утроенной версией: `''`, `""`, `'''`, `"""`

[Больше информации](#)

> Переменные

```
variable = 3
```

Слева – название переменной, затем следует знак равно, а справа – значение, которое было присвоено переменной.

Нужны для хранения значений, при вычислении вместо переменной подставляется её значение.

[Больше информации](#)

> Сравнения

Числа можно сравнивать, в результате сравнений получается логическое значение – True или False

Больше

```
5 > 10  
False
```

Меньше

```
3 < 6  
True
```

Больше либо равно

```
4 >=4  
True
```

Меньше либо равно

```
10 <= 9  
False
```

Равно – используется 2 знака равенства, потому что 1 знак является оператором присвоения для создания переменных

```
5 == 5  
True
```

Не равно

```
6 != 3  
True
```

Больше информации

> Операции над строками

Строки можно складывать, получая их соединение (конкатенат)

```
'Vasya' + 'Petrov'  
'VasyaPetrov'
```

[Больше информации](#)

> Списки (листы, массивы или эрреи)

Более сложный тип данных, позволяющий хранить много значений различных типов. Для его создания необходимо перечислить внутри квадратных скобок `[]` значения через запятую:

```
employees = ['Anatoly', 'Alexander', 'Lavrentii']
```

Список можно изменять (например, добавлять новые значения):

```
employees.append('Rostislav')  
['Anatoly', 'Alexander', 'Lavrentii', 'Rostislav']
```

В данном случае мы использовали метод `.append()`, добавляющий значение в список, у которого мы задействовали метод

Методы похожи на функции, только вызываются у конкретного объекта (в данном случае списка).

Индексирование

Значения (элементы) из списка можно достать с помощью индексирования – для этого укажите после списка в квадратных скобках номер нужного элемента, начиная с 0. Помимо этого можно индексироваться с конца списка, используя отрицательные числа:

```
employees[0]  
'Anatoly'
```

```
employees[2]  
'Lavrentii'
```

```
employees[-1] # negative to index from the end 'Rostislav'
```

Также можно брать срезы (слайсы) списка. То есть получать кусочек исходного списка

- От элемента с индексом 1 до конца

```
employees[1:]  
['Alexander', 'Lavrentii', 'Rostislav']
```

- От начала до элемента с индексом 2

```
employees[:2]
['Anatoly', 'Alexander']
```

- От начала до конца с шагом 2

```
employees[::2]
['Anatoly', 'Lavrentii']
```

Синтаксис среза такой

```
[start : stop : step]
```

- **start** – от какого элемента берём значения, по умолчанию равно 0
- **stop** – до какого элемента берём значения, не включая его, по умолчанию равно длине списка
- **step** – шаг, с которым берём элементы, по умолчанию равен 1

Дефолтные аргументы можно пропускать.

Метод `pop()`

Чтобы убрать элемент из списка по его порядковому номеру (индексу) используется метод `pop()`

```
employees.pop(1)
['Anatoly', 'Lavrentii', 'Rostislav']
```

При этом убираемый из списка элемент возвращается пользователю (мы можем его использовать):

```
lecturer_name = employees.pop(0)
employees
['Lavrentii', 'Rostislav']

lecturer_name
'Anatoly'
```

[Больше информации](#) и ещё [источник](#)

> Правила оформления кода

Конвенциональны и не влияют на работу программы. Их соблюдают для единого стиля программ от разных программистов. Часть из них:

- операнды и операторы (значения и знаки операций, например `+` или `*`) разделяются пробелами
- перед запятой не ставится пробел, но ставится после
- переменные следует называть осмысленно – код становится проще читать, он становится осмысленнее (это может показаться незначительным, но одна из самых важных вещей)

Комментарии

Используются для объяснения того, что происходит в коде. Очень важны, так как код чаще читают, чем пишут, и комментарии упрощают понимание происходящего.

Начинаются с решётки, дальнейшая строка игнорируется питоном

```
# эта строка нужна для объяснения, она не влияет на скрипт
```

Больше информации

> **Использованные функции**

- `len()` – находит длину коллекции (это группа типов данных, куда входят строки, списки, словари), [больше информации](#)

```
len([10, 20, 35])
3
```

- `print()` – печатает то, что указано внутри скобок, [больше информации](#)
- `str()` – превращает переданное значение в строку, [больше информации](#)

```
str(5)
'5'
```

- `int()` – превращает переданное значение в целое число, [больше информации](#)

```
int('10')
10
```

- `float()` – превращает переданное значение в дробное число, [больше информации](#)

```
float('2.5')
2.5
```

> **Условия**

Конструкция для управления ходом программы (что делать в каких-то случаях)

```
if predicate:
    what to do
```


- `if` – ключевое слово, которое даёт понять питону что дальше будет условие (ветвь)
- `predicate` – какое-то выражение, которое сводится к логическому (True/False)
- `:` – элемент синтаксиса (просто так нужно, чтобы питон понял)
- отступ – 4 пробела или `tab` (используйте в программе что-то одно из этих вариантов), используется для обозначения на каком уровне мы находимся, улучшает читаемость
- `what to do` – команды, которые вы хотите выполнить, если попали в эту ветку (то есть `predicate True`); все строки в одном условии должны быть с этим уровнем отступа, чтобы выполняться в нём

Помимо простого условия, которое или выполнится или нет, есть более сложные – с 2-мя и более ветвями. Для их обозначения используются ключевые слова `elif` и `else`. Их можно использовать только если условие началось (вплотную до этого была ветка `if` или `elif`)

`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` обязательно должна быть ветка `if` или `elif` выше

```
if 10 > 0:
    print('1st branch was executed')
elif 10 < 0:
    print('This branch is False')
else:
    print('This branch is unreachable in this program')
```

Больше информации

> Логические операторы

Для логических значений `True` и `False` есть свои специальные операторы (как `+` или `-` для чисел). Они бывают полезны при задании предикатов в условиях и для анализа таблиц, где есть логический тип. Например, таблица с клиентами, где есть колонка просрочил ли клиент выплаты по кредитам.

Итак, что же это за операторы?

- `not` – инвертирует логическое значение

```
not True
```

Результат: `False`

```
not False
```

Результат: `True`

- `or` – даёт `True`, если хотя бы один из операндов `True`

```
False or True
```

Результат: `True`

```
False or False
```

Результат: `False`

- `and` – даёт `False`, если хотя бы один из операндов `False`

```
False and True
```

Результат: `False`

```
True and True
```

Результат: `True`

Логические действия можно соединять друг с другом и указывать порядок выполнения операций, прямо как с арифметическими действиями!

```
True and (False or True) # True and TrueTrue
```

Больше информации

> Циклы (петли)

Конструкция для перебора значений из списка

```
for something in collection:  
    what to do
```

- `for` – ключевое слово, которое даёт понять питону, что дальше будет цикл

- `something` — переменная, куда будет последовательно подставляться значение из списка; нужна, чтобы обращаться к значению
- `in` — ключевое слово, показывающее, что мы работаем с элементом коллекции
- `:` — как в условии
- отступ — как в условии
- `what to do` — тело цикла; то, что будет происходить для каждого из элементов

```
for x in [1, 2, 3]:
    print(x + 10)

111213
```

[Больше информации](#)

Конструкция while

Цикл `while` (“пока”) выполняет тело цикла, пока проверяемое условие истинно. Конструкция выглядит так:

```
while predicate:
    what to do
```

Тело цикла (`what to do`) выполняется, пока условие (`predicate`) после ключевого слова `while` - истина (`True`). Если условие никогда не будет ложным (`False`), тогда цикл будет выполняться бесконечное количество раз. Поэтому важно следить, какое задается условие

Пример бесконечного цикла:

```
x = 0
while x < 10:
    print(x)

000000
...
```

Здесь `x` никогда не бывает больше `10`, но стоит добавить изменение условия в тело цикла, как все заработает корректно

```
x = 0
while x < 10:
    print(x)
    x += 1
```

С каждым новым циклом к `x` прибавляется `1`

Еще один пример:

```
i = 0
a = 2

while i < 10:
    print(a)
    a += 2
    i += 1
```

Здесь `i` выступает как счетчик циклов, с каждым новым циклом значение увеличивается на `1`. Пока `i < 10`, с каждым новым циклом к переменной `a` добавляется `2`.

> Вложенные конструкции

Циклы и условия можно комбинировать и получать более сложные программы. Отступы отображают питону и программисту, какая строка к какой конструкции относится.

```
# Create list with some data
rates = [1.3, 0.95, 1.2, 1.1, 0.7, 1.35]

# Here comes the cycle
for rate in rates:
    if rate > 1: # Start condition, 1 indent for cycle, 1 indent for condition
        print('rate is greater than 1') # body of condition, 1 indent for cycle, 1 indent for condition
    else:
        print('rate is lesser than 1')

rate is greater than 1
rate is lesser than 1
rate is greater than 1
rate is greater than 1
rate is lesser than 1
rate is greater than 1
```

> Форматирование строк

Очень полезно иметь матрицу (template) строки, в которую можно подставлять произвольную подстроку.

```
intro = "My name is {}, I'm from {}"
intro.format('Sasha', 'Russia')
"My name is Sasha, I'm from Russia"

intro.format('Katya', 'Russia')
"My name is Katya, I'm from Russia"
```

`{}` внутри строки означает возможность применения метода `format`. Переданные в него аргументы будут подставлены в соответствующие скобки.

Существуют и другие варианты темплейтов

> Словари (дикты)

Ассоциативный тип данных – в нём каждый элемент является парой ключ-значение. Для создания нужно указать элементы внутри фигурных скобок – {} Синтаксис элементов в словаре –

```
{key: value}
```

```
salaries = {'Ivan': 30000} # key 'Ivan' is associated with value 30000
```

Чтобы узнать что ассоциировано с ключом 'Ivan', нужно проиндексироваться по нему:

```
salaries['Ivan']  
30000
```

Ключами словаря могут быть строки и числа, а значениями почти что угодно – числа, строки, списки, даже другие словари!

Задавание нового элемента в словаре

```
salaries['Anna'] = 50000  
  
salaries  
{'Ivan': 30000, 'Anna': 50000}
```

Итерирование по словарю

- по ключам

```
for name in salaries: # salaries.keys() is analogous  
    print(name)  
  
Ivan  
Anna
```

- по значениям

```
for salary in salaries.values():  
    print(salary)  
  
3000050000
```

[Больше информации](#)