



**УНИВЕРСИТЕТ**  
искусственного  
интеллекта

# Синтаксис Python. Базовые конструкции языка

**#1**



# Синтаксис Python.

## Базовые конструкции языка

### Занятие № 1

#### Знакомство с ресурсом Google Colaboratory.

Язык программирования Python является, пожалуй, одним из самых простых и понятных в плане освоения языков программирования. Наличие огромного количества дополнительных модулей и библиотек делает процесс написания кода максимально легким и быстрым.



Компания Google предоставляет бесплатный сервис для написания программ на языке Python. Данный сервис называется Google Colaboratory. Для того, чтобы воспользоваться этим ресурсом, необходимо иметь действующий аккаунт Google, пройти авторизацию и перейти на сам веб-ресурс <https://colab.research.google.com/>

Интерфейс достаточно прост и легок в освоении. Для создания нового рабочего документа (ноутбука) необходимо выбрать пункт меню File/New notebook.



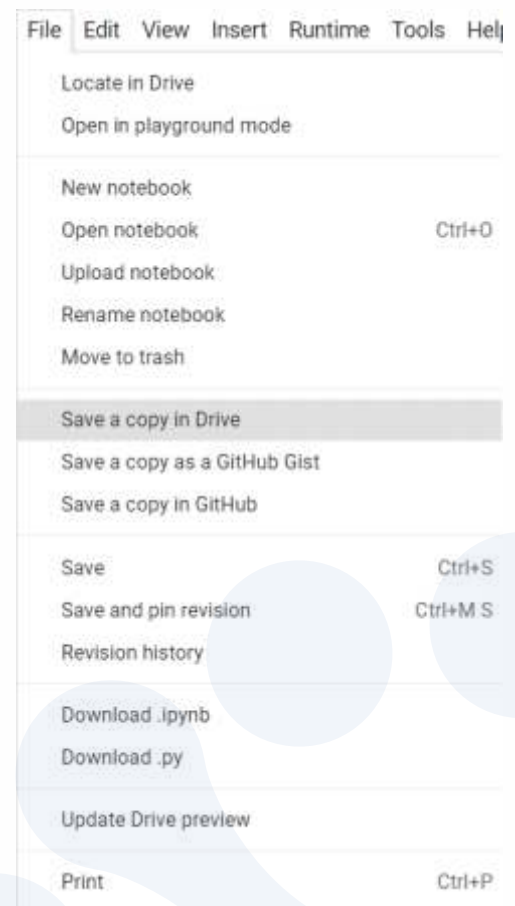
# Знакомство с ресурсом Google Colaboratory

Рабочее пространство ноутбука состоит из отдельных ячеек, которые могут быть двух типов:

- Текстовые ячейки. Создаются с помощью кнопки  + Text. Данные ячейки предназначены для отображения всевозможной текстовой информации: заголовков блока, комментариев к коду и т.п.
- Ячейки кода. Создаются с помощью кнопки  + Code. Основные ячейки, в которых выполняется кодирование и написание программ. Каждая отдельная ячейка представляет собой полноценный блок кода на языке Python, который может быть запущен с помощью кнопки “Run cell” , находящейся в левом верхнем углу ячейки.

К каждому занятию подготавливается отдельный учебный ноутбук, ссылка на который предоставляется участникам. После перехода по ссылке открывается готовый ноутбук, доступный только для чтения (только для просмотра). Чтобы иметь возможность редактировать и запускать ячейки учебного ноутбука, необходимо создать копию этого ноутбука на своем гугл-диске. Выполняется это с помощью пункта меню File/Save a copy in Drive. После этого в новой вкладке браузера откроется Ваша личная копия учебного ноутбука, доступная для редактирования.

По умолчанию при создании ноутбука тип Runtime не определен (None), в таком случае все расчеты будут вестись на CPU. Для того чтобы сменить тип Runtime, необходимо зайти в раздел Runtime



File Edit View Insert **Runtime** Tools Help

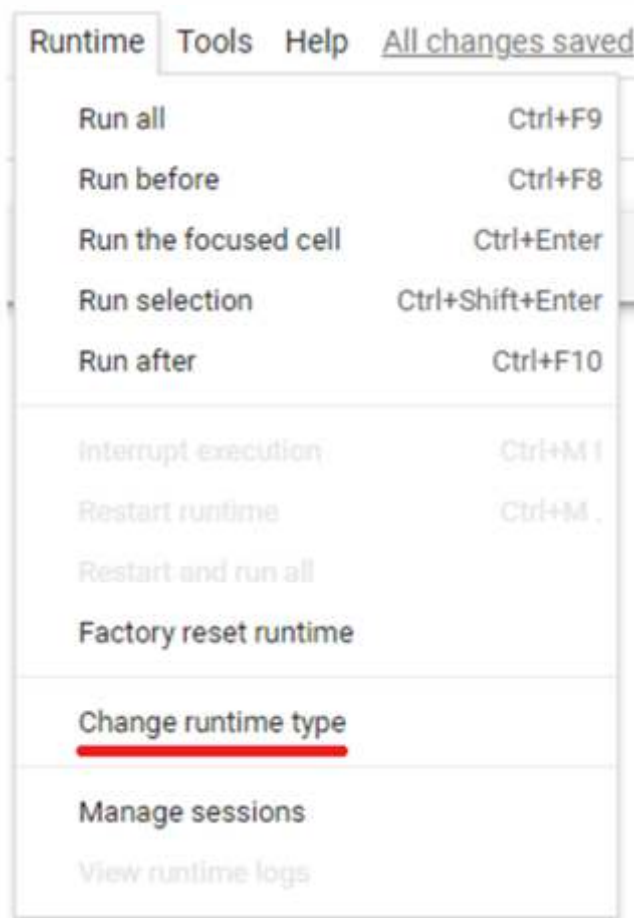
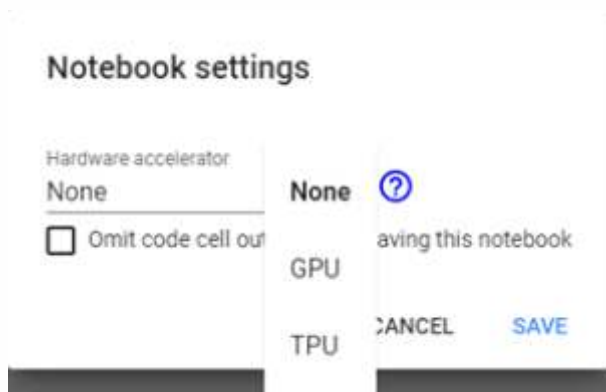
# Знакомство с ресурсом Google Colaboratory

Выбрать Change runtime type

В открывшемся окне выбрать тип runtime GPU или TPU и сохранить SAVE. После этого ноутбук перезагрузится.

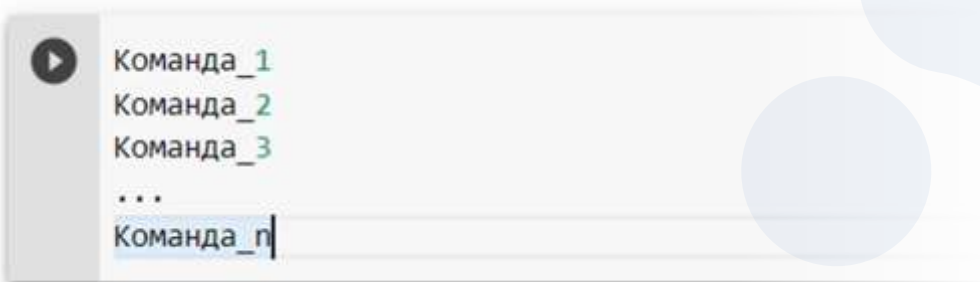
Тип GPU. Выделяется графический адаптер Tesla K80 для расчетов.

Тип TPU. Адаптер с тензорными ядрами.



## Синтаксис языка Python

Программа (на любом языке программирования) – это последовательность действий (команд), заданных разработчиком. Каждая отдельная строка в ноутбуке является командой. Таким образом, работу программы можно представить в следующем виде:



# Синтаксис языка Python

Если запустить такую ячейку (нажать кнопку “Run cell”), то поочередно будут выполняться команды: Команда\_1, Команда\_2, Команда\_3, ..., Команда\_n (в данном случае “Команда\_” – это условное обозначение команды на языке Python).

В самом простом случае программа может состоять всего из одной команды.

Язык Python предоставляет разработчиками набор готовых мини-программ, которые выполняют какое-то определенное действие (считают среднее значение нескольких чисел, печатают произвольную информацию, округляют число и др.). Такие мини-программы называются функциями языка.

Первая функция, с которой мы начнем знакомство с языком Python – функция print.

## Функция print()

Функция print() выводит на печать указанные разработчиком данные.

Создадим новый ноутбук (File/New notebook) и добавим ячейку кода (“+ Code”).

В появившейся ячейке введем print(5) и запустим ячейку (“Run cell”).



В результате появится строка вывода ячейки, в которой будет напечатано число 5. Простыми словами, функция print() выводит на экран данные, которые переданы ей в качестве аргументов (указаны в скобках).

Мы можем передать в функцию (указать в скобках) сразу несколько значений. Добавим еще одну ячейку кода (“+ Code”), в появившейся ячейке введем print(5,6,7,8) и запустим ячейку (“Run cell”).



# Синтаксис языка Python

В результате на экран будут выведены все 4 значения (5,6,7,8).

Большинство функций языка Python содержат справочную информацию, которую можно получить, запустив ячейку с именем функции и вопросительным знаком в конце. Добавим ячейку кода (“+ Code”), в появившейся ячейке введем `print?` и запустим ячейку (“Run cell”). Откроется дополнительное окно с подробным описанием функции `print`.

Полный синтаксис (описание) функции `print`:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Подобная запись означает, что в качестве параметров функции `print()` (в качестве значений, которые указываются в скобках) могут быть следующие:

`*objects` – одно или несколько значений, которые требуется напечатать (\* означает в данном случае один или несколько).

`sep` – разделитель между элементами. По умолчанию этот параметр равен пробелу, поэтому в примере выше между числами 5,6,7,8 стоит пробел. При необходимости мы можем указать произвольный разделитель (может быть как отдельный символ, так и целое слово). В примере выше изменим код на `print(5,6,7,8, sep = '->>')` и запустим ячейку.

В результате между элементами появится указанный нами разделитель: `->>`.

`end` – символ конца строки, который будет добавлен после всех выведенных элементов. По умолчанию этот символ равен `'\n'` (перевод строки – аналог клавиши Enter). Благодаря этому каждое использование функции `print()` будет выводить информацию с новой строки.

До сих пор мы выводили на экран только числовые данные. Функция `print()` может печатать также и символьную информацию. Для этого необходимо символьные данные поместить в кавычки (одинарные или двойные).

Создадим новую ячейку (“+ Code”), в появившейся ячейке введем `print('Тестовая запись')` и запустим ячейку (“Run cell”):

```
[6] 1 print("Тестовая запись")
```

Тестовая запись

В результате в строке вывода будет напечатана следующая фраза:  
Тестовая запись.



# Синтаксис языка Python

## Арифметические операции

Язык Python позволяет выполнять стандартный набор арифметических операций: сложение (+), вычитание (-), умножение (\*), деление (/), возведение в степень (\*\*), целочисленное деление (/ /), получение остатка от деления (%).

Произвести вычисление можно прямо в ячейке кода, просто используя соответствующий символ арифметической операции. Создадим новую ячейку (“+ Code”), в появившейся ячейке введем  $7+5$  и запустим ячейку (“Run cell”).

В результате в строке вывода будет отображено число 12 – результат сложения чисел 7 и 5. Аналогично производятся вычисления для остальных арифметических операций. В рассмотренном примере мы вывели информацию без использования функции `print()`. Такой подход допустим, но не рекомендуем. Создадим новую ячейку (“+ Code”), в появившейся ячейке введем две команды  $7+5$  (на первой строке) и  $7 * 5$  (на второй строке), запустим ячейку (“Run cell”):

```
[1] 1 7 + 5  
    2 7 * 5
```

35

Как видим, в строке вывода отобразилось только число 35 (то есть результат последнего выполненного действия). Для того чтобы получить результат обоих вычислений, необходимо воспользоваться функцией `print`:

```
[2] 1 print(7 + 5)  
    2 print(7 * 5)
```

12  
35

Теперь мы видим оба результата.

Ряд арифметических операций может вызвать определенные сложности для понимания. Рассмотрим более подробно каждую из них.

# Синтаксис языка Python

- Целочисленное деление

Результат целочисленного деления показывает, сколько целых частей знаменателя входит в числитель:  $7 // 3 = 2$ . Простыми словами, число семь содержит две тройки.

$3 // 6 = 0$  (число 3 не содержит ни одного целого числа шесть).

- Остаток от деления

Точнее будет сказать: остаток от целочисленного деления.  $7 \% 3 = 1$ . В примере выше мы рассмотрели, что  $7 // 3 = 2$  – число семь содержит две тройки ( $2 * 3 = 6$ ), и в итоге остается единица ( $7 - 6 = 1$ ). Единица в данном случае и будет результатом операции: остаток от деления.

$9 \% 5 = 4$  ( $9 // 5 = 1$ ,  $9 - 5 * 1 = 4$ )

$6 \% 3 = 0$  ( $6 // 3 = 2$ ,  $6 - 3 * 2 = 0$ )

- Возведение в степень

При использовании данной операции можно получить не только целочисленную степень числа, например:  $5 ** 2 = 25$  или  $3 ** 3 = 27$ , но и получить значение корня произвольной степени. Так, для получения квадратного корня, необходимо выполнить следующее:  $16 ** 0.5 = 4$ ; для получения корня третьей степени:  $27 ** (1/3) = 3$ .

Для закрепления материала рассмотрим решение следующей задачи: найти площадь круга с радиусом  $R=5$ . Подсказка: площадь круга вычисляется по формуле  $S=\pi*R**2$ ,  $\pi \sim 3.14$ .

Создадим новую ячейку (“+ Code”), в появившейся ячейке введем `print(3.14 * 5 ** 2)` и запустим ячейку (“Run cell”):

```
[3] 1 print(3.14 * 5 ** 2)
```

78.5

В результате в строке вывода мы получим значение 78.5, соответствующее площади круга с радиусом 5.

В данном примере мы применили сразу две арифметические операции в одном примере (умножение и возведение в степень). При этом Python все сделал правильно: вначале было подсчитано значение  $5 ** 2$ , а затем произведено умножение  $3.14 * 25$ .



# Синтаксис языка Python

В данном случае мы имеем дело с понятием “приоритет операций”. Самый высокий приоритет среди арифметических операций у операции “возведение в степень” (это означает, что эта операция всегда будет выполняться самой первой). Далее идут операции умножения, деления, целочисленного деления и получение остатка. И самый низкий приоритет у операций сложения и вычитания.

Мы можем управлять приоритетом операций с помощью использования скобок. Например, результатом выражения  $(3.14 * 5) ** 2$  будет число 246.49 (поскольку вначале будет подсчитано значение  $3.14 * 5$  и только затем будет произведено возведение в степень).

## Переменные

До сих пор мы напрямую использовали числовые или текстовые значения, например, для передачи в функцию `print()`:

```
print(2)
print('Язык Питон')
```

В языках программирования, в том числе Python, применяется способ хранения различных значений с помощью контейнеров, которые называются переменными. У каждой переменной есть имя и значение.

Имя переменной не должно начинаться с цифры и не должно включать специальные символы (№, #, ? и т.п.). В остальном имя переменной может быть любым и выбирается разработчиком (оно может включать даже кириллические символы, но это не рекомендуется).

Процесс работы с переменными можно представить на примере работы с таблицей, состоящей из двух колонок (первая – имя переменной, вторая – значение переменной).

Когда мы выполняем следующий код:

```
my_variable = 123
```

Python создает новую переменную с именем `my_variable` и значением 123. Или, возвращаясь к аналогии с таблицей, в таблице появляется новая строка:

# Синтаксис языка Python

Имя переменной	Значение
my_variable	123

И теперь, когда в любом месте программы мы воспользуемся именем переменной (my\_variable), Python автоматически подставит вместо него значение 123. Например, вызвав команду:

```
print(my_variable)
```

на экран выведется значение: 123.

Мы можем поменять значение переменной. Например:

```
my_variable = 4 + 5
```

В результате значение переменной изменится на 9. А наша таблица примет вид:

Имя переменной	Значение
my_variable	9

## Типы данных

Мы уже частично касались этого термина, когда выводили с помощью функции print числовые и символьные данные. В зависимости от типа информации в языке Python выделяют 4 типа данных:

- int – целые числа (1, -45, 16786 и т.п.);
- float – дробные числа или числа с плавающей точкой (4.34, 0.1234, -123.234 и т.п.);
- str – строковый или символьный тип ('Человек', 'Язык Python', 'Google' и т.п.);
- bool – логический тип данных (имеет всего два значения True и False).

Тип данных определяется автоматически в момент создания переменной. Например, когда мы создаем переменную:

```
my_var = 45
```

Python автоматически назначит этой переменной тип int

# Синтаксис языка Python

Еще один пример:

```
my_var2 = 'Python'
```

Тип данной переменной – str.

Таким образом, в нашу таблицу можно добавить еще одну колонку, которая указывает тип переменной:

Имя переменной	Значение	Тип
my_variable	123	int
my_var	45	int
my_var2	Python	str

Особое внимание следует уделить строковому типу str. Значения переменных данного типа должны быть помещены в кавычки (одинарные или двойные). Это обязательное условие и его необходимо запомнить.

Такой код будет ошибочным:

```
my_var3 = Переменная
```

Правильный код такой:

```
my_var3 = 'Переменная'
```

Рассмотрим более подробно логический тип данных bool. Если, например, переменная числового типа может принимать абсолютно любое числовое значение (–123443546, 0, 3, 3454364563465 и т.п.), то переменная логического типа может принимать только два значения: True или False.

```
bool_var1 = True
```

```
bool_var2 = False
```

Для закрепления материала решим задачу подсчета площади круга с использованием переменных. Найти площадь круга с радиусом  $R=5$ . Подсказка: площадь круга вычисляется по формуле  $S=\pi \cdot R^2$ ,  $\pi \sim 3.14$

- Создайте переменную `r` со значением 5
- Создайте переменную `pi` со значением 3.14
- Сохраните результат вычисления в переменную `S`
- Выведите на печать результаты

# Синтаксис языка Python

Создаем переменную `r` и присваиваем ей значение 5

```
r = 5
```

Создаем переменную `pi_` и присваиваем ей значение 3.14

```
pi_ = 3.14
```

Создаем переменную `S` и присваиваем ей значение результата вычисления площади круга:

```
S = pi_ * r ** 2
```

Выводим результат на экран:

```
print('Площадь круга равна:', S)
```

Мы получаем тот же самый результат, что и посчитанный ранее.

## Преобразование типов

Python позволяет преобразовывать один тип данных в другой. Делается это с помощью функций, соответствующих типу данных, к которому преобразовываем значение переменной. Например, для того чтобы преобразовать значение переменной 3.14 к целочисленному типу, необходимо выполнить следующий код:

```
var1 = int(3.14)
```

В результате такого преобразования в переменную `var1` запишется значение 3 (дробная часть будет отброшена и запишется только целая часть).

Строковые данные также могут быть преобразованы к числовому:

```
var2 = int('123')
```

Однако следует понимать, что следующее преобразование приведет к ошибке:

```
var3 = int('число')
```

Поскольку невозможно преобразовать значение 'число' к числовому представлению.

Для того чтобы посмотреть какой тип данных, в Python есть функция `type()`, которая на вход принимает переменную и возвращает ее тип.

# Синтаксис языка Python

Например,

```
print(type(var1))  
print(type(var2))
```

выведет

```
<class 'int'>  
<class 'int'>
```

## Функция input()

Вы уже знакомы с функцией print(), которая выводит информацию на экран. Функция input() позволяет вводить данные с клавиатуры. Функция применяется следующим образом:

```
new_variable = input()
```

После запуска ячейки с этим кодом появится строка ввода, в которую можно ввести произвольную информацию. Введенная информация будет сохранена в переменной new\_variable в строковом типе.

Таким образом, для того чтобы получить числовое представление введенного пользователем числа, необходимо преобразовать введенные данные к типу int

```
new_variable = int(input())
```

Если в скобках функции print указать какой-либо текст, то этот текст будет отображен перед строкой ввода:

```
input('Введите число: ')
```

## Базовые конструкции

### Логические выражения

Для сравнения выражений в языке есть логические выражения.

< меньше

> больше

<= меньше или равно

>= больше или равно

== равно

!= не равно

# Базовые конструкции

Простое логическое выражение имеет вид

<арифметическое выражение> <знак сравнения> <арифметическое выражение>.

Результатом логического выражения будет булево значение True либо False. В целочисленных представлениях True равно 1, а False равно 0.

Например,

```
2 != 3
```

вернет

```
True
```

Также можно сравнивать не значения, а их переменные, например,

```
a = 2
```

```
b = 3
```

выражение `a != b`

вернет `True`

Чтобы записать сложное логическое выражение, часто необходимо воспользоваться логическими связками "и", "или" и "не". В Python они обозначаются как `and`, `or` и `not` соответственно. Операции `and` и `or` являются бинарными, т.е. должны быть записаны между операндами, например `x < 3 or y > 2`. Операция `not` – унарная и должна быть записана перед единственным своим операндом.

Например,

```
x = 4
```

```
y = 5
```

```
print(x > 3 and y > 3)
```

```
print(x > y or x == y)
```

```
print(not (x > y))
```

вернет,

```
True
```

```
False
```

```
True
```



# Базовые конструкции

## Условный оператор

Часто приходится проверять условия и принимать решения в зависимости от этих условий. Для этих целей в Python есть условный оператор `if`.

Синтаксис в Python в общем виде имеет вид:

```
if <условие1>:
```

```
    действие1-1
```

```
    действие1-2
```

```
    .....
```

```
    действие1-N
```

```
elif <условие2>:
```

```
    действие2-1
```

```
    действие2-2
```

```
    .....
```

```
    действие2-N
```

```
else:
```

```
    действие
```

Работа оператора `if` выглядит следующим образом:

если (оператор `if` – “если”) выполняется условие1, то выполнить действия1-1, действие1-2 и т.д. После чего будут выполняться команды после условного оператора. Если условие1 НЕ выполняется, то проверяется условие2 (`elif` – “иначе если”). Если условие2 выполняется, то выполняются действия: действие2-1, действие2-2 и т.д. В случае если не выполняется ни одно условие, то выполняются действия после оператора `else` (“иначе”). В случае, если после проверки условий ничего не нужно выполнять, оператор `else` можно не использовать.

# Базовые конструкции

Например,

```
if x > y:
    print('X больше Y')
    print('x - y = ', x - y)
elif y > x:
    print('Y больше X')
    print('y - x = ', y - x)
else:
    print('X равно Y')
```

При  $x=4$   $y=4$ , выведет фразу

'X равно Y'

При  $x=5$   $y=4$ , выведет фразу

'X больше Y'

При  $x=4$   $y=5$ , выведет фразу

'Y больше X'

Операторы `elif` и `else` являются не обязательными.

В Python для выделения блоков используются пробелы (по умолчанию принято использовать 4 пробела) либо табуляция. Рассмотрим пример:

если пропустить пробелы и записать его в виде

```
if x > y:
    print('X больше Y')
print('x - y = ', x - y)
```

то в случае если условие не выполняется (например,  $x=3$ ,  $y=4$ ), то на экран выведется:

$x - y = -1$

хотя задача, чтобы оно выполнялось, только когда выполняется условие.

В случае если забыть пробелы в конструкции,

```
if x > y:
    print('X больше Y')
print(x - y)
else:
    print('')
    print('Y не меньше, чем X')
```

то получим ошибку,

`SyntaxError: invalid syntax`

так как оператор `else` (а также и оператор `elif`) не может использоваться без оператора `if`. Т.к. мы забыли пробелы перед `print`, то `else` уже другой блок, в результате чего получили ошибку.

# Базовые конструкции

## Цикл while

Для выполнения периодических операций, пока выполняется какое-то условие в Python, есть оператор while (“пока”). Проверка условия происходит каждый раз после выполнения блока действий. Синтаксис в Python:

```
while <условие>:  
    <действие1>  
    <действие2>
```

Например,

```
value = 3  
while value > 0:  
    print('Это многослойная сеть', value)  
    value -= 1
```

Пока value больше 0, выводим на экран 'Это многослойная сеть' и значение value, а также уменьшаем value на 1. Если выполнить выражение, то на экране выведется

```
Это многослойная сеть 3  
Это многослойная сеть 2  
Это многослойная сеть 1
```

**ВАЖНО!** При использовании цикла while необходимо, чтобы параметры условия модифицировались в цикле, иначе получите бесконечный цикл, который не завершится никогда. Например,

```
value = 3  
while value > 0:  
    print('Это многослойная сеть', value)
```

Это бесконечный цикл, который будет выводить на экран бесконечно долго, пока не закончится оперативная память на ПК.

```
Это многослойная сеть 3  
так как мы убрали уменьшение value на 1.
```

## Цикл for

Часто требуется перебрать все значения в массиве данных и к каждому объекту этого массива применить действие. Для это в Python есть цикл for. Синтаксис в Python:

# Базовые конструкции

for <значение> in <массив данных>:

    <действие1>

    <действие2>

    .....

    <действиеN>

Дословно, для <значение> из <массива данных> выполнить действия <действие1>, <действие2> и т.д.

Например,

```
a = [0, 1, 2, 3]
```

```
for i in a:
```

```
    print(i)
```

Для i из массива данных a напечатать i. На экран выведется

0

1

2

3

## Функция range()

Часто требуется создать список значений. Для того чтобы его не писать вручную, в Python есть функция генерации значений range(). Функция range() принимает в качестве аргументов 3 параметра: start – начальное значение, stop – конечное значение, step – шаг. Параметры start, step являются не обязательными, stop – обязательный. По умолчанию start = 0, step=1. Диапазон значений от start до stop (не включая значение stop) с шагом step. Рассмотрим примеры:

```
for i in range(10):
```

```
    print(i)
```

на экран выведется

0

1

2

3

4

5

6

7

8

9

# Базовые конструкции

```
for i in range(3, 12):  
    print(i)
```

на экран выведется

3  
4  
5  
6  
7  
8  
9  
10  
11

```
for i in range(3, 12, 2):  
    print(i)
```

на экран выведется

3  
5  
7  
9  
11

## Базовые структуры

### Список (list)

Чтобы не создавать множество переменных для хранения данных, их можно структурировать в наборы данных, для этого в Python есть списки (list). Списки – это изменяемый набор данных, в котором можно хранить данные разного типа. Для создания списка можно элементы списка через запятую обернуть в квадратные скобки [, ].

Например,

среди наших коллег провели опрос о том, какие ассоциации вызывает каждое из времен года. Итог опроса показал среди топовых ассоциаций следующие:

# Базовые структуры

```
zima = 'снег'  
vesna = 'солнце'  
leto = 'фрукты'  
osen = 'листья'
```

хранить их в отдельной переменной и помнить название каждой неудобно, для этого создадим список

```
associations = ['снег', 'солнце', 'фрукты', 'листья']
```

в котором будем хранить все наши ассоциации, в данном случае в списке данные одного типа.

Другой пример: нам надо хранить данные о человеке, для этого можно создать список, например, person, в котором будут данные о человеке,

```
person = ['Иванов', 'Иван', 'Иванович', 1999, 170, 70]
```

содержащие данные разного типа.

Также бывает список из списков, например,

```
associations = [['Зима', 'снег'], ['Весна', 'солнце'],  
['Лето', 'фрукты'], ['Осень', 'листья']]
```

К элементам списка можно обратиться по индексу элемента. Индексы начинают с 0, а не с 1, то есть индекс первого элемента в списке равен 0, а не 1.

```
print(associations[0])
```

выведет

```
снег
```

Индексом может быть не только положительное число, но и отрицательное. В случае отрицательных индексов отсчет идет с -1.

Значение списка с индексом -1 будет последний элемент списка. Например,

```
print(associations[-1])
```

выведет

```
листья
```

Также к элементам списка можно обратиться и с помощью оператора цикла for:

```
for a in associations:  
    print(a)
```

выведет

```
снег
```

```
солнце
```

```
фрукты
```

```
листья
```



# Базовые структуры

Для операций над самим списком есть множество методов.

Методы:

`list.append(x)` добавляет элемент в конец списка

`list.extend(L)` расширяет список `list`, добавляя в конец все элементы списка `L`

`list.insert(i, x)` вставляет на `i`-ый элемент значение `x`

`list.remove(x)` удаляет первый элемент в списке, имеющий значение `x` `ValueError`, если такого элемента не существует

`list.pop([i])` удаляет `i`-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент

`list.index(x, [start [, end]])` возвращает положение первого элемента со значением `x` (при этом поиск ведется от `start` до `end`)

`list.count(x)` возвращает количество элементов со значением `x`

`list.sort([key=функция])` сортирует список на основе функции

`list.reverse()` разворачивает список

`list.copy()` поверхностная копия списка

`list.clear()` очищает список

Обращение к методу списка(`list`) происходит через объект '.', например, забыли про ассоциацию, которая связана с осенью. Для добавления в список ассоциаций необходимо:

```
associations.append('грязь')
print(associations)
```

в результате получим:

```
['снег', 'солнце', 'фрукты', 'листья', 'грязь']
```

## Словарь (dict)

В случае если нам надо хранить данные в массиве, к которым необходимо обращаться не по числовому индексу, а по имени, в Python есть структура словарь(`dict`). Другими словами, словарь – это именованный список.

# Базовые структуры

Для создания словаря можно воспользоваться конструкцией <ключ>:<значение> через запятую, обернув в фигурные скобки {, }. Например,

```
my_dict = {'Зима' : 'снег',
           'Весна' : 'солнце',
           'Лето' : 'фрукты',
           'Осень' : 'листья'
          }
```

Чтобы получить значения всех ключей, у словаря есть метод .keys(). Для нашего примера:

```
my_dict.keys()
```

вернет

```
dict_keys(['Зима', 'Весна', 'Лето', 'Осень'])
```

Для того чтобы получить список значений словаря, есть метод .values(). Для нашего примера:

```
my_dict.values()
```

вернет

```
dict_values(['снег', 'солнце', 'фрукты', 'листья'])
```

Для того чтобы получить список пар ключ–значение, есть метод .items(). Для нашего примера:

```
my_dict.items()
```

вернет,

```
dict_items([('Зима', 'снег'), ('Весна', 'солнце'), ('Лето', 'фрукты'), ('Осень', 'листья')])
```

Для того чтобы получить значение, надо обратиться по ключу, например,

```
my_dict['Зима']
```

вернет,

```
'снег'
```

Словари – изменяемые объекты, для того чтобы добавить новое значение, необходимо новому ключу списка присвоить значение, например,

```
my_dict['Высокосный год'] = '29 февраля'
```

# Базовые структуры

---

в результате получим:

```
my_dict
```

вернет

```
{'Весна': 'солнце',  
  'Високосный год': '29 февраля',  
  'Зима': 'снег',  
  'Лето': 'фрукты',  
  'Осень': 'листья'}
```

Чтобы удалить элемент из словаря, существует метод `.pop()`, который принимает значение ключа, который необходимо удалить.

```
my_dict.pop('Высокосный год')
```

в результате

```
print(my_dict)
```

получим:

```
{'Зима': 'снег', 'Весна': 'солнце', 'Лето': 'фрукты', 'Осень':  
  'листья'}
```

## Глоссарий

---

### Функции (из коробки):

**print()** – вывод на экран

**input()** – ввод с клавиатуры

**len()** – длина объекта

**range()** – диапазон чисел

### Основные типы данных:

**int** – целые числа (7)

**float** – числа с плавающей точкой (7.0)

**string** – строковые данные ('строка')

**bool** – булевы данные (True/False)

# Глоссарий

---

## Базовые структуры данных:

**list** – структура, предназначенная для хранения данных разных типов.  
преобразовать объект в список `list()`

**set** – множество – похож на список с той разницей, что в нем не может быть повторяющихся элементов. Т. е., если список может быть `[1,2,2]`, то множество будет `(1,2)`.

Преобразовать объект в множество/создать пустое множество `set()`

**tuple** – кортеж – похож на список с той разницей, что в него нельзя добавить новые элементы.

Преобразовать объект в кортеж/создать новый кортеж `tuple()`

**dict** – структура, предназначенная для хранения данных разных типов, при этом организована по типу ключа–значение.

Преобразовать в словарь/создать новый словарь {КЛЮЧ:ЗНАЧЕНИЕ}

## Краткий гайд по обозначениям в Python

---

### Если это...

`[]` – список

`()` – кортеж

`{ }` – словарь

`[[], []]` – список списков

`[(), ()]` – список кортежей

`[{ }, { }]` – список словарей

`([], [])` – кортеж из списков

`((), ())` – кортеж из кортежей

`({ }, { })` – кортеж из словарей

# Краткий гайд по обозначениям в Python

---

{ключ:[значение]} – словарь и значение представлено в виде списка

{ключ: ()} – словарь и значение представлено в виде кортежа

.() – функция/метод

Пример:

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

```
a.sort() # вызываем функцию сортировки
```

```
>>>>[1, 2, 3, 4, 5]
```

.– атрибут объекта

```
array.shape
```

## А если так, то...

a[ ] – вытащить значение под конкретным индексом. a является в этом случае итерируемым объектом (объект, в котором есть другие объекты)

a( ) – вызвать функцию (если необходимы параметры, то указываются в скобках)