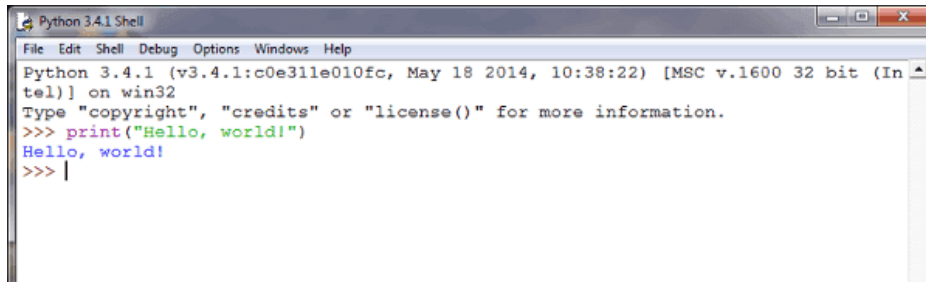


Традиционно, первой программой у нас будет "hello world".

Чтобы написать "hello world" на python, достаточно всего одной строки:

```
print("Hello world!")
```

Вводим этот код в IDLE и нажимаем Enter. Результат виден на картинке:



Поздравляю! Вы написали свою **первую программу на python!** (если что-то не работает).

С интерактивным режимом мы немного познакомились, можете с ним ещё поиграться, например, написать

```
print(3 + 4)

print(3 * 5)

print(3 ** 2):

name = input("Как Вас зовут? ")

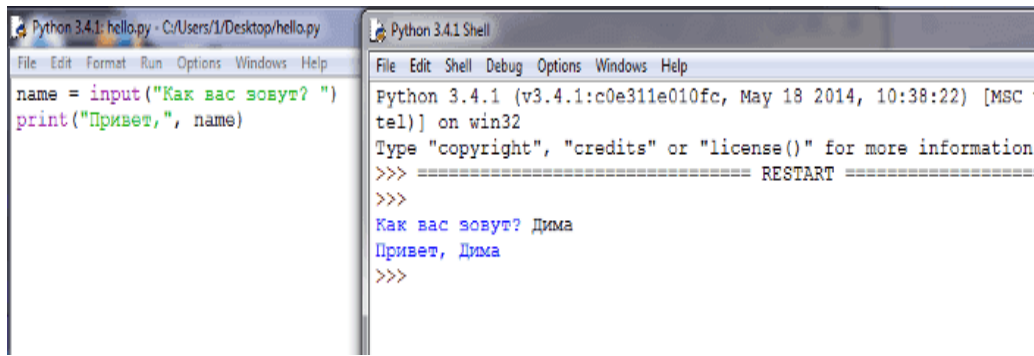
print("Привет, ", name)
```

Первая строка печатает вопрос ("Как Вас зовут? "), ожидает, пока вы не напечатаете что-нибудь и не нажмёте Enter и сохраняет введённое значение в переменной name.

Во второй строке мы используем функцию print для вывода текста на экран, в данном случае для вывода "Привет, " и того, что хранится в переменной "name".

Теперь нажмём F5 (или выберем в меню IDLE Run → Run Module) и убедимся, что то, что мы написали, работает. Перед запуском IDLE предложит нам сохранить файл. Сохраним туда, куда вам будет удобно, после чего программа запустится.

Вы должны увидеть что-то наподобие этого (на скриншоте слева - файл с написанной вами программой, справа - результат её работы):



The image shows two windows from a Python 3.4.1 IDE. The left window, titled 'Python 3.4.1: hello.py - C:/Users/1/Desktop/hello.py', contains the following code:

```
name = input("Как вас зовут? ")
print("Привет, ", name)
```

The right window, titled 'Python 3.4.1 Shell', shows the execution of this code. It displays the prompt 'Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) [MSC tel) on win32', followed by instructions to type 'copyright', 'credits' or 'license()' for more information. After a restart, it shows the input 'Как вас зовут? Дима' and the output 'Привет, Дима'.

## Синтаксис

- Конец строки является концом инструкции (точка с запятой не требуется).
- Вложенные инструкции объединяются в блоки по величине отступов. Отступ может быть любым, главное, чтобы в пределах одного вложенного блока отступ был одинаков. И про читаемость кода не забывайте. Отступ в 1 пробел, к примеру, не лучшее решение. Используйте 4 пробела (или знак табуляции, на худой конец).
- Вложенные инструкции в Python записываются в соответствии с одним и тем же шаблоном, когда основная инструкция завершается двоеточием, вслед за которым располагается вложенный блок кода, обычно с отступом под строкой основной инструкции.

- Основная инструкция:

Вложенный блок инструкций

## Несколько специальных случаев

- Иногда возможно записать несколько инструкций в одной строке, разделяя их точкой с запятой:

```
a = 1; b = 2; print(a, b)
```

Но не делайте это слишком часто! Помните об удобочитаемости. А лучше вообще так не делайте.

- Допустимо записывать одну инструкцию в нескольких строках. Достаточно ее заключить в пару круглых, квадратных или фигурных скобок:

- `if (a == 1 and b == 2 and`
- `c == 3 and d == 4):` *# Не забываем про двоеточие*

```
print('spam' * 3)
```

- Тело составной инструкции может располагаться в той же строке, что и тело основной, если тело составной инструкции не содержит составных инструкций. Ну я думаю, вы поняли :). Давайте лучше пример приведу:

```
if x > y: print(x)
```

Полное понимание синтаксиса, конечно, приходит с опытом, поэтому я советую вам заглянуть в рубрику ["Примеры программ"](#).

## Синтаксис инструкции if

Сначала записывается часть if с условным выражением, далее могут следовать одна или более необязательных частей elif, и, наконец, необязательная часть else. Общая форма записи условной инструкции if выглядит следующим образом:

```
if test1:

    state1

elif test2:

    state2

else:

    state3
```

Простой пример (напечатает 'true', так как 1 - истина):

```
>>> if 1:

...     print('true')

... else:

...     print('false')

... 
```

```
true
```

Чуть более сложный пример (его результат будет зависеть от того, что ввёл пользователь):

```
a = int(input())

if a < -5:

    print('Low')

elif -5 <= a <= 5:

    print('Mid')

else:

    print('High')
```

Конструкция с несколькими `elif` может также служить отличной заменой конструкции `switch - case` в других языках программирования.

## Проверка истинности в Python

- Любое число, не равное 0, или непустой объект - истина.
- Числа, равные 0, пустые объекты и значение `None` - ложь
- Операции сравнения применяются к структурам данных рекурсивно
- Операции сравнения возвращают `True` или `False`
- Логические операторы `and` и `or` возвращают истинный или ложный объект-операнд

Логические операторы:

```
X and Y
```

Истина, если оба значения `X` и `Y` истинны.

```
X or Y
```

Истина, если хотя бы одно из значений `X` или `Y` истинно.

```
not X
```

Истина, если X ложно.

## Трехместное выражение if/else

Следующая инструкция:

```
if X:

    A = Y

else:

    A = Z
```

довольно короткая, но, тем не менее, занимает целых 4 строки. Специально для таких случаев и было придумано выражение if/else:

```
A = Y if X else Z
```

В данной инструкции интерпретатор выполнит выражение Y, если X истинно, в противном случае выполнится выражение Z.

```
>>>
```

```
>>> A = 't' if 'spam' else 'f'

>>> A

't'
```

## Цикл while

While - один из самых универсальных циклов в Python, поэтому довольно медленный. Выполняет тело цикла до тех пор, пока условие цикла истинно.

```
>>>
```

```
>>> i = 5

>>> while i < 15:
```

```
...     print(i)

...     i = i + 2

...

5

7

9

11

13
```

## Цикл for

Цикл for уже чуточку сложнее, чуть менее универсальный, но выполняется гораздо быстрее цикла while. Этот цикл проходится по любому итерируемому объекту (например строке или списку), и во время каждого прохода выполняет тело цикла.

```
>>>
```

```
>>> for i in 'hello world':

...     print(i * 2, end='')

...

hheellllloo  wwoorrlldd
```

## Оператор continue

Оператор continue начинает следующий проход цикла, минуя оставшееся тело цикла (for или while)

```
>>>
```

```
>>> for i in 'hello world':
```

```
...     if i == 'o':

...         continue

...     print(i * 2, end='')

...

hheelllll  wwrrlldd
```

## Оператор break

Оператор break досрочно прерывает цикл.

```
>>>
```

```
>>> for i in 'hello world':

...     if i == 'o':

...         break

...     print(i * 2, end='')

...

hheelllll
```

## Волшебное слово else

Слово else, примененное в цикле for или while, проверяет, был ли произведен выход из цикла инструкцией break, или же "естественным" образом. Блок инструкций внутри else выполнится только в том случае, если выход из цикла произошел без помощи break.

```
>>>
```

```
>>> for i in 'hello world':

...     if i == 'a':

...         break
```

```
... else:

    print('Буквы а в строке нет')

...
```

Буквы а в строке нет

## Ключевые слова

**False** - ложь.

**True** - правда.

**None** - "пустой" объект.

**and** - логическое И.

**with / as** - [менеджер контекста](#).

**assert** условие - возбуждает исключение, если условие ложно.

**break** - выход из цикла.

**class** - [пользовательский тип](#), состоящий из методов и атрибутов.

**continue** - переход на следующую итерацию цикла.

**def** - определение [функции](#).

**del** - удаление объекта.

**elif** - в противном случае, если.

**else** - см. [for/else](#) или [if/else](#).

**except** - перехватить исключение.

**finally** - вкупе с инструкцией try, выполняет инструкции независимо от того, было ли исключение или нет.

**for** - цикл for.

**from** - импорт нескольких функций из модуля.

**global** - позволяет сделать значение переменной, присвоенное ей внутри функции, доступным и за пределами этой функции.

**if** - [если](#).

**import** - импорт модуля.

**in** - проверка на вхождение.

**is** - ссылаются ли 2 объекта на одно и то же место в памяти.

**lambda** - определение анонимной функции.

**nonlocal** - позволяет сделать значение переменной, присвоенное ей внутри функции, доступным в объемлющей инструкции.

**not** - логическое НЕ.



**or** - логическое ИЛИ.

**pass** - ничего не делающая конструкция.

**raise** - возбудить исключение.

**return** - вернуть результат.

**try** - выполнить инструкции, перехватывая [исключения](#).

**while** - [цикл](#) while.

**yield** - определение функции-генератора.

## Целые числа (int)

Числа в Python 3 ничем не отличаются от обычных чисел. Они поддерживают набор самых обычных математических операций:

$x + y$	Сложение
$x - y$	Вычитание
$x * y$	Умножение
$x / y$	Деление
$x // y$	Получение целой части от деления
$x \% y$	Остаток от деления
$-x$	Смена знака числа
$\text{abs}(x)$	Модуль числа

<code>divmod(x, y)</code>	Пара ( <code>x // y</code> , <code>x % y</code> )
<code>x ** y</code>	Возведение в степень
<code>pow(x, y[, z])</code>	$x^y$ по модулю (если модуль задан)

Также нужно отметить, что целые числа в python 3, в отличие от многих других языков, поддерживают длинную арифметику (однако, это требует больше памяти).

```
>>>
```

```
>>> 255 + 34
```

```
289
```

```
>>> 5 * 2
```

```
10
```

```
>>> 20 / 3
```

```
6.666666666666667
```

```
>>> 20 // 3
```

```
6
```

```
>>> 20 % 3
```

```
2
```

```
>>> 3 ** 4
```

```
81
```

```
>>> pow(3, 4)
```

```
81
```

```
>>> pow(3, 4, 27)
```

```
0
```

```
>>> 3 ** 150
```

```
3699884850351269729247007824516966441864731003897229738151844053017  
48249
```

## Битовые операции

Над целыми числами также можно производить битовые операции

$x   y$	Побитовое <i>или</i>
$x \wedge y$	Побитовое <i>исключающее или</i>
$x \& y$	Побитовое <i>и</i>
$x \ll n$	Битовый сдвиг влево
$x \gg y$	Битовый сдвиг вправо
$\sim x$	Инверсия битов

## Дополнительные методы

`int.bit_length()` - количество бит, необходимых для представления числа в двоичном виде, без учёта знака и лидирующих нулей.

```
>>>
```

```
>>> n = -37

>>> bin(n)

'-0b100101'

>>> n.bit_length()

6
```

**int.to\_bytes**(length, byteorder, \*, signed=False) - возвращает [строку байтов](#), представляющих это число.

```
>>>
```

```
>>> (1024).to_bytes(2, byteorder='big')

b'\x04\x00'

>>> (1024).to_bytes(10, byteorder='big')

b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'

>>> (-1024).to_bytes(10, byteorder='big', signed=True)

b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'

>>> x = 1000

>>> x.to_bytes((x.bit_length() // 8) + 1, byteorder='little')

b'\xe8\x03'
```

classmethod **int.from\_bytes**(bytes, byteorder, \*, signed=False) - возвращает число из данной строки байтов.

```
>>>
```

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
```

```
16
```

```
>>> int.from_bytes(b'\x00\x10', byteorder='little')
```

```
4096
```

```
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
```

```
-1024
```

```
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
```

```
64512
```

```
>>> int.from_bytes([255, 0, 0], byteorder='big')
```

```
16711680
```

## Системы счисления

Те, у кого в школе была информатика, знают, что числа могут быть представлены не только в десятичной системе счисления. К примеру, в компьютере используется двоичный код, и, к примеру, число 19 в двоичной системе счисления будет выглядеть как 10011. Также иногда нужно переводить числа из одной системы счисления в другую. Python для этого предоставляет несколько функций:

- **int([object], [основание системы счисления])** - преобразование к целому числу в десятичной системе счисления. По умолчанию система счисления десятичная, но можно задать любое основание от 2 до 36 включительно.
- **bin(x)** - преобразование целого числа в двоичную строку.
- **hex(x)** - преобразование целого числа в шестнадцатеричную строку.
- **oct(x)** - преобразование целого числа в восьмеричную строку.

## Литералы строк

Работа со строками в Python очень удобна. Существует несколько литералов строк, которые мы сейчас и рассмотрим.

### Строки в апострофах и в кавычках

```
S = 'spam"s'
```

```
S = "spam's"
```

Строки в апострофах и в кавычках - одно и то же. Причина наличия двух вариантов в том, чтобы позволить вставлять в литералы строк символы кавычек или апострофов, не используя экранирование.

## Экранированные последовательности - служебные символы

Экранированные последовательности позволяют вставить символы, которые сложно ввести с клавиатуры.

Экранированная последовательность	Назначение
\n	Перевод строки
\a	Звонок
\b	Забой
\f	Перевод страницы
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\N{id}	Идентификатор ID базы данных Юникода

Экранированная последовательность	Назначение
\uhhhh	16-битовый символ Юникода в 16-ричном представлении
\Uhhhh...	32-битовый символ Юникода в 32-ричном представлении
\xhh	16-ричное значение символа
\ooo	8-ричное значение символа
\0	Символ Null (не является признаком конца строки)

## "Сырые" строки - подавляют экранирование

Если перед открывающей кавычкой стоит символ 'r' (в любом регистре), то механизм экранирования отключается.

```
S = r'C:\newt.txt'
```

Но, несмотря на назначение, "сырая" строка не может заканчиваться символом обратного слэша. Пути решения:

```
S = r'\n\n\\'[:-1]
```

```
S = r'\n\n' + '\\'
```

```
S = '\\n\\n'
```

## Строки в тройных апострофах или кавычках

Главное достоинство строк в тройных кавычках в том, что их можно использовать для записи многострочных блоков текста. Внутри такой строки возможно присутствие кавычек и апострофов, главное, чтобы не было трех кавычек подряд.

```
>>> c = '''это очень большая

... строка, многострочный

... блок текста'''

>>> c

'это очень большая\nстрока, многострочный\nблок текста'

>>> print(c)

это очень большая

строка, многострочный

блок текста
```

## Что такое списки?

Списки в Python - упорядоченные изменяемые коллекции объектов произвольных типов (почти как массив, но типы могут отличаться).

Чтобы использовать списки, их нужно создать. Создать список можно несколькими способами. Например, можно обработать любой итерируемый объект (например, [строку](#)) встроенной функцией **list**:

```
>>>
```

```
>>> list('список')

['с', 'п', 'и', 'с', 'о', 'к']
```

Список можно создать и при помощи литерала:

```
>>>
```



```
>>> s = [] # Пустой список

>>> l = ['s', 'p', ['isok'], 2]

>>> s

[]

>>> l

['s', 'p', ['isok'], 2]
```

Как видно из примера, список может содержать любое количество любых объектов (в том числе и вложенные списки), или не содержать ничего.

И еще один способ создать список - это **генераторы списков**. Генератор списков - способ построить новый список, применяя выражение к каждому элементу последовательности. Генераторы списков очень похожи на цикл [for](#).

```
>>>
```

```
>>> c = [c * 3 for c in 'list']

>>> c

['lll', 'iii', 'sss', 'ttt']
```

Возможна и более сложная конструкция генератора списков:

```
>>>
```

```
>>> c = [c * 3 for c in 'list' if c != 'i']

>>> c

['lll', 'sss', 'ttt']

>>> c = [c + d for c in 'list' if c != 'i' for d in 'spam' if d != 'a']

>>> c
```

```
['ls', 'lp', 'lm', 'ss', 'sp', 'sm', 'ts', 'tp', 'tm']
```

Но в сложных случаях лучше пользоваться обычным циклом `for` для генерации списков.

## Функции и методы списков

Создать создали, теперь нужно со списком что-то делать. Для списков доступны основные [встроенные функции](#), а также методы списков.

### Таблица "методы списков"

Метод	Что делает
<b>list.append(x)</b>	Добавляет элемент в конец списка
<b>list.extend(L)</b>	Расширяет список list, добавляя в конец все элементы списка L
<b>list.insert(i, x)</b>	Вставляет на i-ый элемент значение x
<b>list.remove(x)</b>	Удаляет первый элемент в списке, имеющий значение x. ValueError, если такого элемента не существует
<b>list.pop([i])</b>	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
<b>list.index(x, [start [, end]])</b>	Возвращает положение первого элемента со значением x (при этом поиск ведется от start до end)
<b>list.count(x)</b>	Возвращает количество элементов со значением x

Метод	Что делает
<code>list.sort([key=функция])</code>	Сортирует список на основе функции
<code>list.reverse()</code>	Разворачивает список
<code>list.copy()</code>	Поверхностная копия списка
<code>list.clear()</code>	Очищает список

Нужно отметить, что методы списков, в отличие от [строковых методов](#), изменяют сам список, а потому результат выполнения не нужно записывать в эту переменную.

```
>>>
```

```
>>> l = [1, 2, 3, 5, 7]
```

```
>>> l.sort()
```

```
>>> l
```

```
[1, 2, 3, 5, 7]
```

```
>>> l = l.sort()
```

```
>>> print(l)
```

```
None
```

И, напоследок, примеры работы со списками:

```
>>>
```

```
>>> a = [66.25, 333, 333, 1, 1234.5]

>>> print(a.count(333), a.count(66.25), a.count('x'))

2 1 0

>>> a.insert(2, -1)

>>> a.append(333)

>>> a

[66.25, 333, -1, 333, 1, 1234.5, 333]

>>> a.index(333)

1

>>> a.remove(333)

>>> a

[66.25, -1, 333, 1, 1234.5, 333]

>>> a.reverse()

>>> a

[333, 1234.5, 1, 333, -1, 66.25]

>>> a.sort()

>>> a

[-1, 1, 66.25, 333, 333, 1234.5]
```

## Взятие элемента по индексу

Как и в других языках программирования, взятие по индексу:

```
>>>
```

```
>>> a = [1, 3, 8, 7]

>>> a[0]

1

>>> a[3]

7

>>> a[4]

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

IndexError: list index out of range
```

Как и во многих других языках, нумерация элементов начинается с нуля. При попытке доступа к несуществующему индексу возникает исключение `IndexError`.

В данном примере переменная `a` являлась [СПИСКОМ](#), однако взять элемент по индексу можно и у других типов: строк, кортежей.

В Python также поддерживаются отрицательные индексы, при этом нумерация идёт с конца, например:

```
>>>
```

```
>>> a = [1, 3, 8, 7]

>>> a[-1]

7

>>> a[-4]

1

>>> a[-5]

Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
IndexError: list index out of range
```

## Срезы

В Python, кроме индексов, существуют ещё и **срезы**.

item[START:STOP:STEP] - берёт срез от номера START, до STOP (не включая его), с шагом STEP. По умолчанию START = 0, STOP = длине объекта, STEP = 1. Соответственно, какие-нибудь (а возможно, и все) параметры могут быть опущены.

```
>>>
```

```
>>> a = [1, 3, 8, 7]
```

```
>>> a[:]
```

```
[1, 3, 8, 7]
```

```
>>> a[1:]
```

```
[3, 8, 7]
```

```
>>> a[:3]
```

```
[1, 3, 8]
```

```
>>> a[::2]
```

```
[1, 8]
```

Также все эти параметры могут быть и отрицательными:

```
>>>
```

```
>>> a = [1, 3, 8, 7]
```

```
>>> a[::-1]
```

```
[7, 8, 3, 1]
```

```
>>> a[:-2]

[1, 3]

>>> a[-2::-1]

[8, 3, 1]

>>> a[1:4:-1]

[]
```

В последнем примере получился пустой список, так как `START < STOP`, а `STEP` отрицательный. То же самое произойдёт, если диапазон значений окажется за пределами объекта:

```
>>>
```

```
>>> a = [1, 3, 8, 7]

>>> a[10:20]

[]
```

Также с помощью срезов можно не только извлекать элементы, но и добавлять и удалять элементы (разумеется, только для изменяемых последовательностей).

```
>>>
```

```
>>> a = [1, 3, 8, 7]

>>> a[1:3] = [0, 0, 0]

>>> a

[1, 0, 0, 0, 7]

>>> del a[:-3]

>>> a
```

[0, 0, 7]