Язык программирования Python

(версия 3)

(в данном пособие сразу после кода на языке программирования python приведен результат его выполнения)

Лирическое отступление - вкратце о ЯП, о интерпретируемости python, о существовании python2 и python3. Данное руководство будет доступно позднее в почти не измененном виде. Python чуствителен к регистру. Основная команда, которая понадобится новичку в любом языке программирования - вывод информации на экран. В Python3 это можно сделать следующим образом:

In [5]:

```
print('Hello world')
print("""ahsdgas
fjh
sdfkjg""")
```

```
Hello world
ahsdgas
fjh
sdfkjg
```

Разберем приведенную программу:

- Строки заключаются в кавычки, либо "", либо ''. Многострочные строки: """""""
- Встроенная функция print выводит на экран строку, указанную в кавычках
- Обратите внимание на круглые скобки - в них пишется аргумент

Переменные:

Познакомимся с базовыми типами в языке python. Поскольку python обладает динамической типизацией, работать с разными типами легко (нет). Основные типы:

- `int` - числовой тип. Его размер неограничен
- `str` - (string) - строковый тип.
- `bool` - boolean - логический тип (True or False)
- `list` - неизменяемые списки (массивы)
- `tuple` - неизменяемые списки (массивы)
- и многие другие

В python не нужно указывать типы переменных, он подберет их сам. Чтобы узнать тип переменной, можно использовать функцию type()

In [6]:

```
#С помощью решетки можно оставлять комментарии

print(type(123)) # число
print(type("Hello world")) #  строка
```

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

In [5]:

```
print(type("123")) # А это?
print(type(Hello)) # А это?
```

```
<class 'str'>

---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-5-1b85e10dc160> in <module>()
      1 print(type("123")) # А это?
----> 2 print(type(Hello)) # А это?

NameError: name 'Hello' is not defined
```

Также команда `print` умеет выводить несколько объектов сразу, разделяя их **пробелом**

In [2]:

```
print('hello,', 'world')
print('ten is', 10)
a = 1
b = 2
print(a, '+', b, '=', a + b)
```

```
hello, world
ten is 10
1 + 2 = 3
```

Как вы заметили, `print` всегда разделяет пробелами объекты, который он выводит. Также `print` добавляет перенос строки после того, как выводит все объекты. Давайте научимся управлять этим:

- Параметр `sep`: Им задается то, что выводиться между объектами. По умолчанию `sep=" "`. Чтобы изменить его нужно написать после всех аргументов `sep="new separator"` (конечно же нужно указать тот разделитель, который вам нужен

In [2]:

```
a = 10
b = "hello"
c = 1.5
print(a, b, c, sep=", ")
print(a, *b, c, sep="-") # * записывает элементы списка(строки) через запятую
print(a, b, c, sep="")
```

```
10, hello, 1.5
10-h-e-l-l-o-1.5
10hello1.5
```

- Параметр end: Им задается то, что выводит `print` после вывода объектов. По умолчанию `end="\n"`. `\n` - это спецсимвол, обозначающий перевод строки

In [5]:

```
s = "my big string"
print(s, end = "\n") # по умолчанию
print(s, end = " ") # без новой строки
print(s, end = " end of line\n")
print("hello\nworld\nand\nyou") # напечатает целых 4 строки
```

```
my big string
my big string my big string end of line
hello
world
and
you
```

Научимся работать с числами. В python доступны большинство арифметических операций:

+ - * //

In [4]:

```
a = 10
b = 20
c = a + b # c = 30
d = c * 2 # d = 60
e = d - b # e = 40
f = d // a # f = 6
```

Обратите внимание, что деление записывается двумя слешами. Бонус: в python есть встроенная операция возведения в степень, записывается

In [5]:

```
print(2 ** 5)
```

```
32
```

В python есть операция взятия остатка от деления. Будьте осторожны, она работает по математическому определению.

In [3]:

```
print(12418924791824 % 100)
```

```
24
```

Теперь научимся работать со строками. Python позволяет сливать их рядом (конкатенация):

In [16]:

```
a = 'hello'
b = 'world'
c = a + ' ' + b
print(c)
```

```
hello world
```

Оператор in позволяет проверять наличие вхождение одной строки в другую:

In [20]:

```
print('-' in 'into')
```

```
False
```

Логический тип - является результатом логических выражений.

In [3]:

```
a = True # False
b = not a # and - И; or - ИЛИ
print(b)
```

```
False
```

Логические выражения. Можно использовать следующие операторы:

- <
- >
- <= - важно, что равно стоит на втором месте
- >=
- == - важно, что два символа равно (один символ - присваивание)
- != - проверка н
- in
- not
- not in (так называемый синтаксический сахар)

А теперь научимся вводить. Для этого используем функцию input() следующим образом

In [8]:

```
a = input()
print(a)
type(a) # как думаете, а какой тип здесь, если ввести 10?
```

```
7125471254
7125471254
```

Out[8]:

```
str
```

Чтобы преобразовать строку "10" к числу 10, воспользуемся функцией int()

In [22]:

```
a = '10'
print(type(a))
b = int(a)
print(type(b))
print(a, b)
```

```
<class 'str'>
<class 'int'>
10 10
```

Таким образом, самый удобный способ считать число с клавиатуры:

In [ ]:

```
a = int(input())
```

Продолжим разговор про числа. Как вы могли заметить, они не всегда являются целыми. Для представления нецелых чисел существует специальный тип, называемый "float". Чтобы обозначить питону, что число следует считать не целым, следует поставить точку в его записи:

In [23]:

```
a = 10 # целое
b = 10.0 # вещественное
print(type(a))
print(type(b))
print(a == b)
```

```
<class 'int'>
<class 'float'>
True
```

Несколько тонкостей преобразования из целых чисел в вещественные и наоборот

In [30]:

```
a = 10 # a - int
b = 1.0 * a # результат операции вещественный, если хотя бы один из элементов ве
щественной
d = float(a) # Практически аналогична команде выше
print(a, b, d)
```

```
10 10.0 10.0
```

In [28]:

```
a = 12.2
b = 12.7
print(int(a))
print(int(b))
```

```
12
12
12
```

Округление:

In [29]:

```
print(round(12.2))
print(round(12.7))
print(round(12.5))
```

```
12
13
12
```

В остальном, все операции для вещественных чисел аналогичны операциям с целыми числами. Обратите внимание:

In [3]:

```
a = 10
b = 5
#a и b - целые числа
c = a / b
print(c)
print(type(c))
#c - вещественное число!!!
```

```
2.0
<class 'float'>
```

Условный оператор

Часто возникает необходимость в зависимости от какого-то условия выполнять различные действия. В python это позволяет делать оператор *if*.

Шаблон оператора if (то, что писать не обязательно указано в квадратных скобках):

In [ ]:

```
if <условие>:
    <код 1 (с одинаковым отступом)>
[elif <условие 1>:
    <код 2 (с одинаковым отступом)>]
[elif <условие 1>:
    <код 3(с одинаковым отступом)>]
    ...
[else:
    <код 4(с одинаковым отступом)>]
```

Несколько важных моментов:

- все команды `elif` и `else` пишутся на таком же уровне, что и `if`
- Правило выполнения команд:
  - Сначала проверяется условие 1
  - Если оно верно, то выполняется код 1
  - Если оно неверно, то управление будет переходить по очереди к блокам `elif`
  - Если очередное условие в `elif` оказалось верным, то выполняется соответствующий код
  - Если ни одно условие (как в `if`, так и в `elif`) не выполняется, что запускается код 4
- В любом случае выполнится не более одного кода. Если есть ветка `else`, то обязательно выполнится ровно один код
- Ветка после `if` должна быть обязательно
- Количество веток `elif` внутри одного `if` не ограниченно
- Ветка `else` может либо быть ровно одна, либо вовсе отсутствовать

Несколько примеров использования `if`:

In [7]:

```
a = int(input())
if a % 2 == 0:
    print('четное')

if a % 2 != 0:
    print('нечетное')


## Запишем тоже самое немного короче:
if a % 2 == 0:
    print('четное')
else:
    print('нечетное')
```

```
11
нечетное
нечетное
```

In [10]:

```
x = int(input())
y = int(input())
#Определим, в какой четверти находится число
if x > 0 and y > 0:
    print("Первая четверть")
elif x > 0 and y < 0:
    print("Четвертая четверть")
elif y > 0:#Если дошли досюда, то точно x < 0, иначе бы выполнились бы предыдущие условия. Поэтому проверим только y
    print("Вторая четверть")
else:
    print("Третья четверть")
```

```
-1
-2
Третья четверть
```

Логические операторы

Кроме приведенных выше операторов сравнения (`< > <= >= == !=`) еще есть логические операторы, которые помогают задавать сложные логические выражения:

- `and` - логическое И (верно только если верны левый и правый операнд)
- `or` - логические ИЛИ (верно, если верен хотя бы один операнд)
- `not` - логическиое отрицание (верно, если единстенный операнд неверен)

`and` и `or` - принимают два операнда (слева и справа). `not` - только один (справа)

Логические И имеет больший приоритет, чем ИЛИ. К примеру, выражение `A and B or C` верно, если верно либо C, либо и A, и B. Однако, если поставить скобки, трактовка будет другая: выражение `A and (B or C)` верно, если верно A и верно либо B либо C.

Пример:

- выражение `a % 10 == 0 or b % 10 == 0` верно если хотя бы одно из чисел `a` или `b` заканчивается на 0.

Списки (list)

Как и понятно из названия, списки в python позволяют хранить упорядоченную информацию. В python списки обозначаются квадратными скобками. Переменные могут быть типа список, рассмотрим на примере:

In [13]:

```
a1 = [1, 3, 2]
a2 = []
a3 = ['hello', 102, [3, 1, 2]]
print(type(a1))
```

```
<class 'list'>
```

С помощью функции `len` можно узнавать длину списка:

In [14]:

```
print(a3)
print(len(a3))
```

```
['hello', 102, [3, 1, 2]]
3
```

Поскольку в списке элементы упорядоченны, у них есть свои номера, по которым можно к ним обращаться: Обратите внимание, элементы нумеруются с нуля!! Также можно использовать отрицательные индексы: a[-1] - последний элемент, a[-2] - предпоследний и так далее.

In [15]:

```
a = [5, 2, 3, 4, 1]
print(a[0])
print(a[3])
print(a[-1])
a[-2] = 10
print(a)
```

```
5
4
1
[5, 2, 3, 10, 1]
```

Рассмотрим некоторые операции со списками:

- Сложение - складываемые списки "сливаются друг с другом" (в конец первого дописываются все элементы второго)
- Умножение - по аналогии - список складывается сам с собой

Подробнее в примерах:

In [16]:

```
ar = [1, 2]
pr = [2, 3]
print(ar + pr)
print(ar * 4)
print([1] * 10)
```

```
[1, 2, 2, 3]
[1, 2, 1, 2, 1, 2, 1, 2]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Чтобы добавить в конец списка новый элемент можно воспользоваться методом `append`

In [18]:

```
ar = [1, 5]
ar.append(6)
ar += [4] # аналогично предыдущему
print(ar)
```

```
[1, 5, 6, 4]
```

Построение списков

В python есть функция, которая позволяет генерировать списки из целых чисел. `range(n)` создаст список из элементов `[0, 1, 2, ..., n - 1]`

In [19]:

```
print(list(range(10)))
print(list(range(20)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 1
9]
```

Чтобы вывести, я использовал функцию `list`, которая преобразовывает специальный объект генератора к удобному списку. Это нужно только для того, чтобы вывести список на экран

`range(start, stop)` сгенерирует список вида `[start, start + 1, ..., stop - 1]`

In [20]:

```
print(list(range(3, 10)))
print(list(range(70, 20)))
```

```
[3, 4, 5, 6, 7, 8, 9]
[]
```

`range(start, stop, step)` сгенерирует список вида `[start, start + step, ..., last]`, где `last` обладает следующими свойствами:

- `last` < `stop`
- `last` + `step` >= `stop`

In [22]:

```
print(list(range(3, 20, 4)))
print(list(range(30, 12, -5)))
```

```
[3, 7, 11, 15, 19]
[30, 25, 20, 15]
```

цикл for

Цикл for в python перебирает значения переданного ему списка, переменная цикла принимает каждый раз значение какого-то элемента списка

In [ ]:

```
for <переменная цикла> in <список>:
    <код с отступом>
[else:
    <ветка>]
```

Несколько примеров:

In [24]:

```
for i in [2, 34, 4]:
    print(i)
```

```
2
34
4
```

In [23]:

```
for i in range(3, 6):
    print(i * i)
```

```
9
16
25
```

Обратите внимание, как ведет себя переменная цикла. В начале каждой итерации переменной цикла присваивается новое значение, пытаться изменять его бессмысленно:

In [26]:

```
i = 10
for i in range(4):
    print(i)
    i = 'bye'
print('after:')
print(i)
```

```
0
1
2
3
after:
bye
```

Иногда возникает необходимость прервать выполнение цикла (например мы хотим найти первый нулевой элемент, а дальнейшие нам не интересны) Для этого есть специальная команда `break` Более того, теперь можно вспомнить про `else`: в ветку `else` мы попадем только в том случае, если не выполнился ни один `break`

In [7]:

```
for i in range(10):
    print(i, end=" ")
    if i > 3:
        break
else:
    print("I'm here")
```

```
0 1 2 3 4
```

In [8]:

```
for i in range(5):
    print(i, end=" ")
    if i > 11:
        break
else:
    print("I'm here")
```

```
0 1 2 3 4 I'm here
```

Цикл while

Цикл `while` выполняется, пока условие верно. Синтаксис (в квадратных скобочках написано то, что писать необязательно):

In [ ]:

```
while <условие>:
    <тело цикла с отступом>
[else:
    <тело else>]
```

Действие ветки `else` аналогично

Пример `while`:

In [9]:

```
i = 1
sum = 0
while i < 5:
    sum += i
    i += 1
print(sum)
```

```
10
```

Срезы

Срезы - особый способ индексации на списках, позволяющий выделать подсписки. На самом деле с ними очент просто работать, ведь аргументы аналогичны range

Итак, с помощью квадратных скобок обращаемся по индексу, однако нумерация с нуля:
Отрицательные индексы - с конца списка

In [11]:

```
a = [1, 5, 3, 6, 7]
print(a[0], a[3], a[-1], a[-3])
```

```
1 6 7 3
```

Также можно указывать два числа через двоеточие, получится новый подсписок по следующему правилу: ar[a:b] - список, состоящий из элементов [ar[a], ar[a + 1], ..., ar[b - 1]]

In [15]:

```
a = [1, 5, 3, 6, 7]
#    0  1  2  3  4
print(a[1:3], a[3:5], a[-4:-1])
```

```
[5, 3] [6, 7] [5, 3, 6]
```

Можно явно не писать индексы среза, в этом случае будет использоватся начало и конец списка соответственно

In [16]:

```
a = [1, 5, 3, 6, 7]
print(a[:3], a[3:], a[-2:])
```

```
[1, 5, 3] [6, 7] [6, 7]
```

Можно добвалять третий аргумент - шаг. Последовательность индексов строится по аналогии с `range`.

Проще увидеть на примере:

In [17]:

```
a = [1, 5, 3, 6, 7, 8, 9, 2]
print(a[::2], a[2:-1:3])
```

```
[1, 3, 7, 9] [3, 8]
```

Генераторы

Генераторы - это очень мощный инструмент python'a, который позволяет генериировать списки из существующих.

Синтаксис (а вот здесь квадратные скобки нужны и показывают, что мы генерируем список):

In [ ]:

```
[<результирующее выражение> for <переменная цикла> in <исходный список>]
```

In [ ]:

```
[<результирующее выражение> for <переменная цикла> in <исходный список> if <усло
вие, по которому берем элемент>]
```

Парочка примеров: