Строки и отступы

Физическая строка – это то, что вы видите, когда набираете программу

Логическая строка – это то, что Python видит как единое предложение

```
a = 'Hello, world!'  b = "Hello, world!" # error
a = 'Hello, world!'; b = "Hello, world!" # ok
```

Блок – набор команд с одинаковым отступом

```
a = 'Hello, world!'
 b = "Hello, world!" # error
```

Перенос строки:

```
print \
(i)  # it's the same as print(i)
```

Однострочковые выражения

```
# 3 statements in one line
print ('Hi'); print ('Hello'); print ('Hola!') # ok
# Using semicolons with loops
for i in range(4): print ('Hi'); print ('Hello') # ok
# The same result
for i in range(4):
    print ('Hi')
    print ('Hello')
# Expression and block in one line
print ('Hi'); for i in range (4): print ('Hello') # error
```

Использование разделителя “;” считается плохим стилем, лучше избегать

Типизация в Python

Python – язык со строгой неявной динамической типизацией

Динамическая типизация – переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной

Строгая (сильная) типизация – язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования, например, нельзя вычесть из строки множество

Python использует неявную типизацию – задача определения типа переменных возложена на интерпретатор

Типы данных в Python

- NoneType

```
a = None
if a is None:
    print('a is None')
else:
    print('a is not None')
```

- Логические переменные (Boolean type)

```
a = x>5 # True or False
type(a) # bool
```

- Числа (Numeric type)

```
type(4) # int
type(4.2) # float
type(complex(2,4)) # complex
```

Типы данных в Python

- Строки (Text Sequence Type)

```
type('hello') # str
```

- Байтовые строки (Binary Sequence Types)

```
a = b'hello'
a = bytes('hello', encoding = 'utf-8')
type(a) # bytes
bytes([50,100,76,72,41])      # 2dLH)
```

- Списки (Sequence Type):
  tuple – кортеж, list – список, range – диапазон
- Множества (Set Types):
  set – множество, frozenset – неизменяемое множество
- Словари (Mapping Types):
  dict – словарь

Инициализация переменных

Любая переменная в Python (в том числе типов str, float, int) является объектом

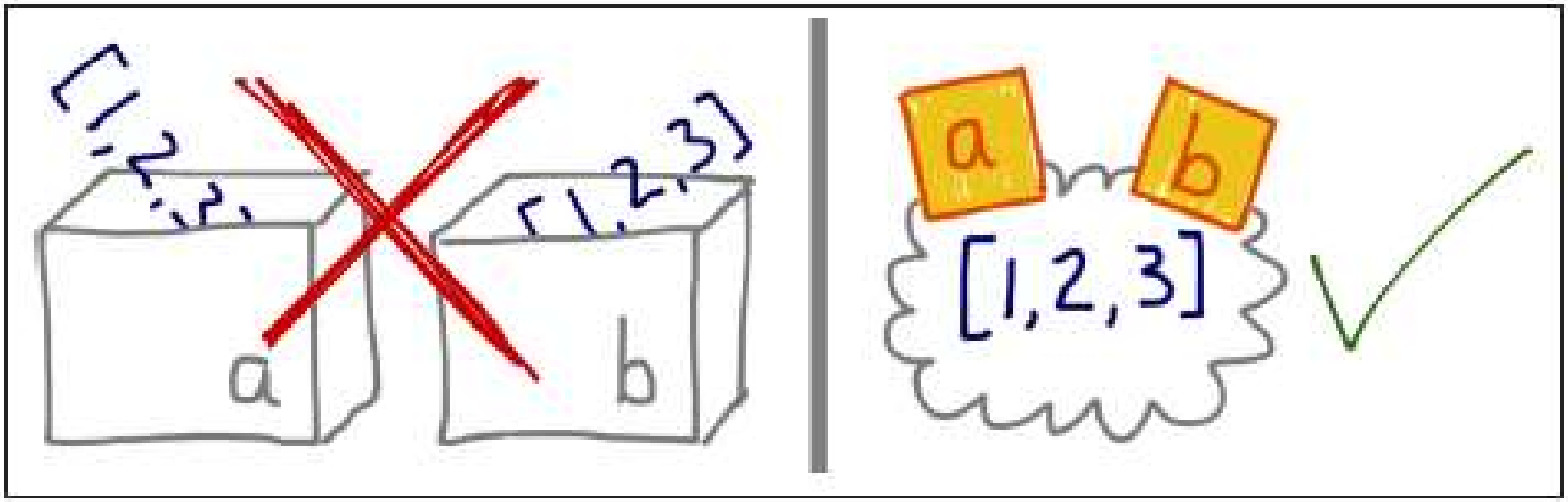
Каждый объект имеет три атрибута:
идентификатор, значение и тип

```
a = 5
id(a) # 1396862560
print (a) # 5
type(a) # int
```

При инициализации переменной происходит следующее:

- создается целочисленный объект 5
- создается ссылка между переменной a и целочисленным объектом 5

Variables Are Not Boxes



Python variables are sticky notes

```
a = [1,8,7,3]
b = a
b is a # True
id(b) == id(a) # True
```

Изменяемые и неизменяемые типы

- Неизменяемые типы (immutable)

  bool, int, float, complex, str, tuple, frozen set

  Неизменяемость типа данных означает, что созданный объект данного типа больше невозможно изменить

```
s = 'hello'
s[0] = 'H' # error
```

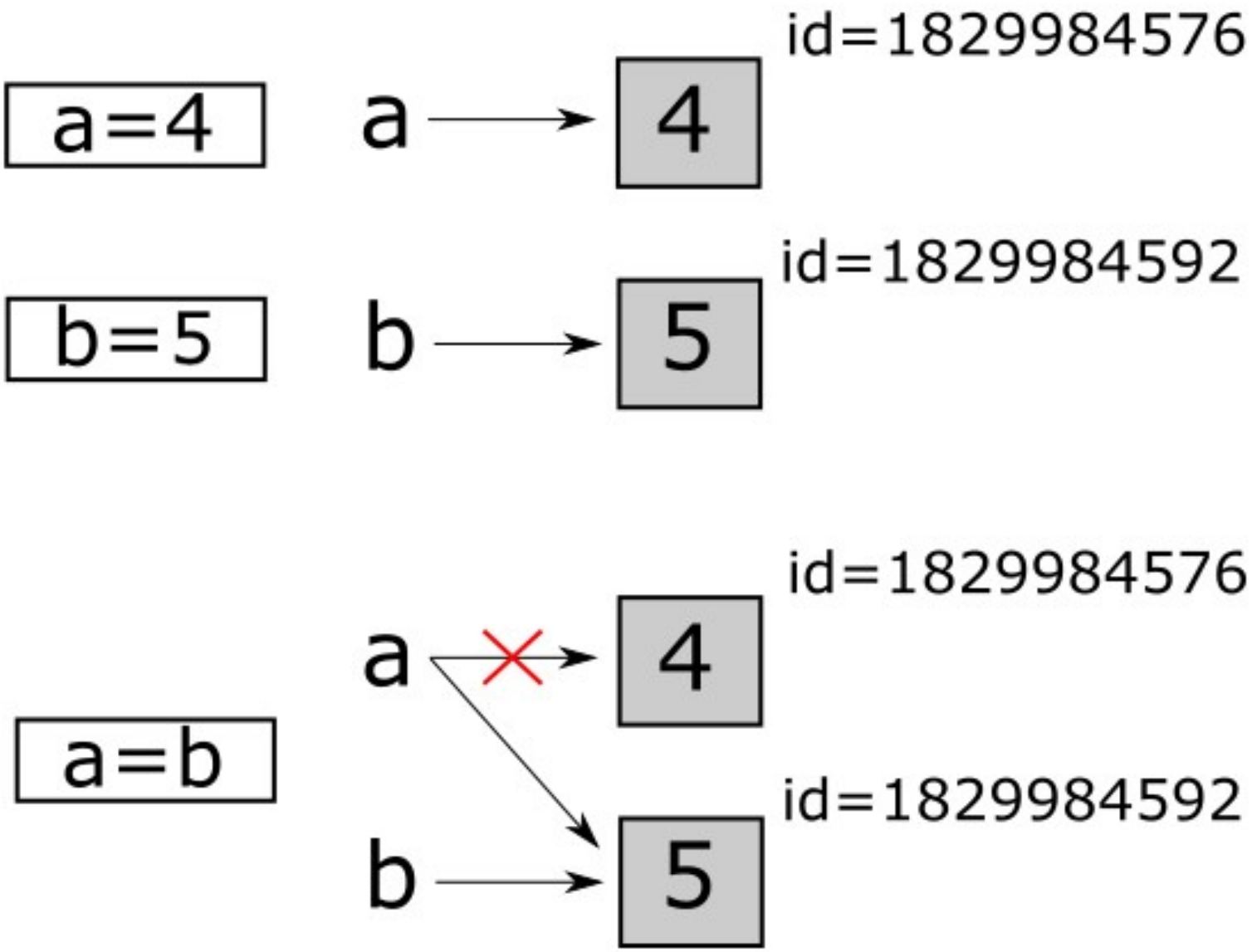
- Изменяемые типы (mutable)

  list, set, dict

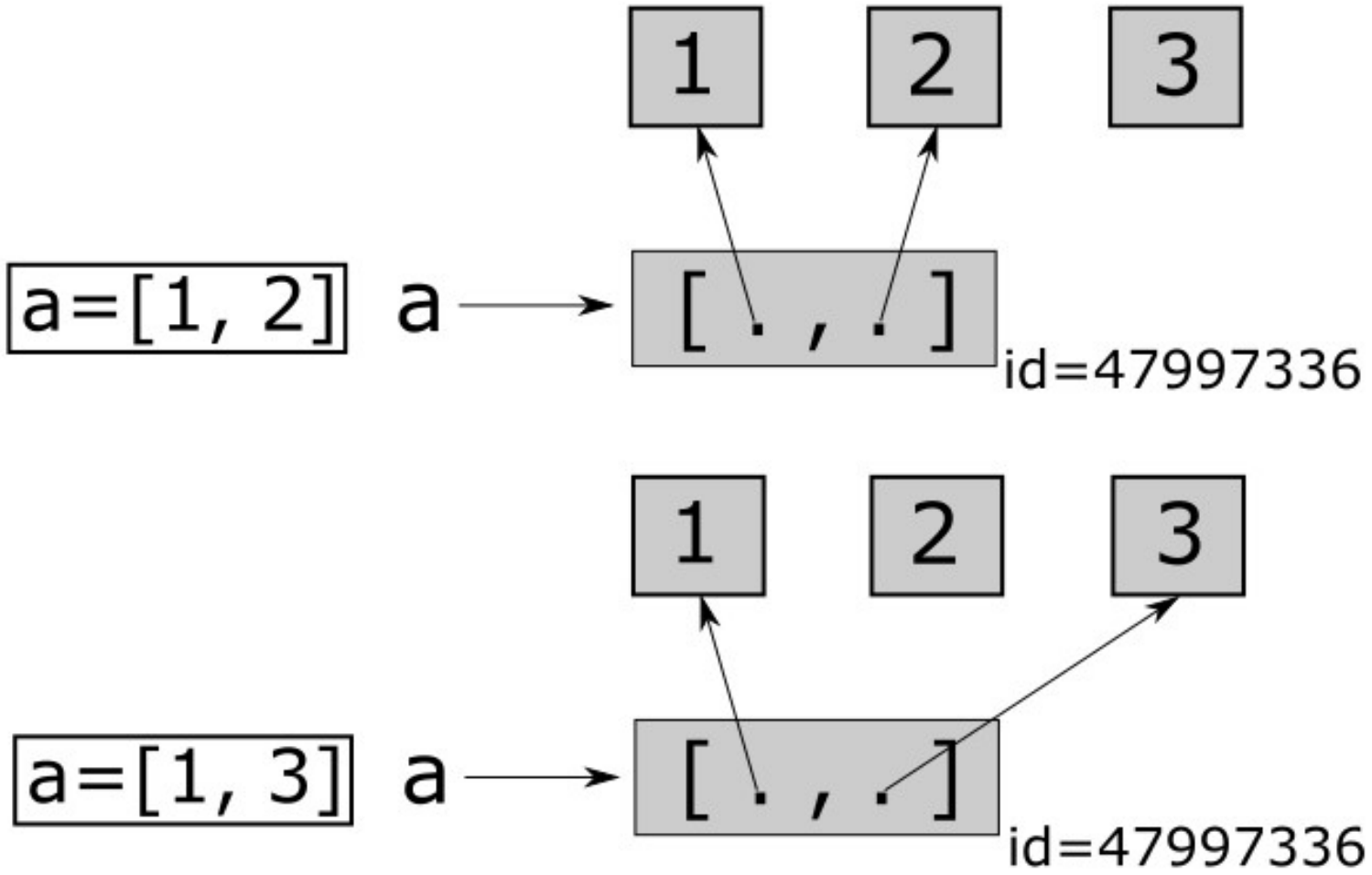
  Значение объекта можно менять

```
a = [10,12]
id(a) # 47997336
a[1] = 5 # [10,5]
id(a) # 47997336
```

Неизменяемые типы



Изменяемые типы



Преобразование типов

- **bool**(x) – преобразование к типу bool
- **int**(x) – преобразование к целому типу
- **float**(x) – преобразование к вещественному типу
- **complex**(x) – преобразование к комплексному типу
- **str**(x) – преобразование к строковому типу
- bytes(x) – преобразование к байтовой строке

```
int(4.8) # 4
str(12.4) # '12.4'
int('a') # error
float('a') # error
complex(4) # 4+0j
complex(4, 2) # 4+2j
ord('a') # 97
chr(97) # 'a'
```

Строки

```
a = 'Hello, world!'
b = "Hello, world!"
c = '''Multiline string
It's a second line'''
d = "It's a line \
the same line"
e = "Hello," " world!" # concatenation
e = "Hello,"+" world!" # concatenation
e = "Hello!"*3 # Hello!Hello!Hello!
f = 'Hello\nworld' # multiline string
g = r'Hello\nworld' # Hello\\nworld
g = b'hello' # byte string
g = bytes('hello', 'utf-8') # byte string
h = '\x4b\x4c' # 'KL'
```

Срезы (Slices)

S	a	m	m	y		S	h	a	r	k	!
0	1	2	3	4	5	6	7	8	9	10	11

S	a	m	m	y		S	h	a	r	k	!
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
s = 'spameggs'
len(s) # 8
s[0] # 's'
s[3:5] # 'me'
s[:6] # 'spameg'
s[1:] # 'pameggs'
s[:] # 'spameggs'
s[2:-2] # 'ameg'
```

```
s[::-1] # 'sggemaps'
s[3:5:-1] # ''
s[2::2] # 'aeg'
s[i:j:step]

b = b'spam'
len(b) # 4
b[0] # 115
```

Форматирование строк. Функция format

```
s = '{0}, {1}, {2}'.format('a','b','c') # 'a, b, c'
s = '{}, {}, {}'.format('a','b','c') # 'a, b, c'
s = '{2}, {1}, {0}'.format('a','b','c') # 'c, b, a'
s = '{:>30}'.format('right aligned')
#'                  right aligned'
s = '{:^30}'.format('centered')
#'           centered           '
s = '{:*^30}'.format('centered')
#'***********centered***********'
s = '{:+f}; {:+f}'.format(3.14,-3.14)
#'+3.140000; -3.140000'
s = '{0:.2f}, {1:.4f}'.format(3.1415,-3.1415)
#'3.14, -3.1415'
s = '{0[0]}, {0[1]}, {1}'.format([3,5],'b') # 3, 5, b
```

Форматирование строк. F-строки

Оператор %

```
s = '%.2f %d %s\n' % (3.14,10,'abc') # '3.14 10 abc'
```

F-строки

```
name = "Eric"
age = 74
print(f"Hi, {name}. You're {age}") # Hi, Eric. You're 74
print(f"{2 * 37}") # 74
def to_lower(s):
    return s.lower()
name = "Eric Idle"
print(f"{to_lower(name)} is funny") # eric idle is funny
print(f"{name.lower()} is funny") # eric idle is funny
```

Строки и байтовые последовательности

Код символа Unicode (code point) – число от 0 до 10FFFF

Байтовая последовательность, соответствующая символу, зависит от кодировки символа

Например, code point U+0041 (символ A) соответствует байтовая строка `\x41`, U+20AC (символ €) – байтовая строка `\xe2\x82\xac` в кодировке UTF-8

Encoding – конвертация из code point в байтовую строку
Decoding – конвертация из байтовой строки в code point

```
s = 'cafe'
len(s) # 4
b = s.encode('utf8') # b'caf\xc3\xa9'
len(b) # 5
b.decode('utf8') # 'cafe'
```

Управление потоком команд. if–elif–else

```
if guess == number:
    print('Good') # block
elif guess < number:
    print('Less') # block
else:
    print('Mode') # block

# one-line block
if guess == number: print('Good')
# one-line if-else
print('Good') if guess == number else print('Not good')
# one-line conditional assignment
s = 'Good' if guess == number else 'Not good'
```

Управление потоком команд. for–else

```
fruits = ['apple', 'banana', 'mango']
for fruit in fruits:
    print(fruit.capitalize())
# Apple Banana Mango
```

Блок **else** выполняется только если цикл завершился полностью (без **break**)

```
for n in range(2,10):
    for x in range(2,n):
        if n % x == 0:
            print(n, 'equals', x, '*', n/x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

Управление потоком команд. while–else

```
do = True
while do:
    print('*')
    do = False
# Output: *
```

```
for n in range(2,10):
    x = 2
    while x<n:
        if n % x == 0:
            break
        x += 1
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

Управление потоком команд. `break`, `continue`, `pass`

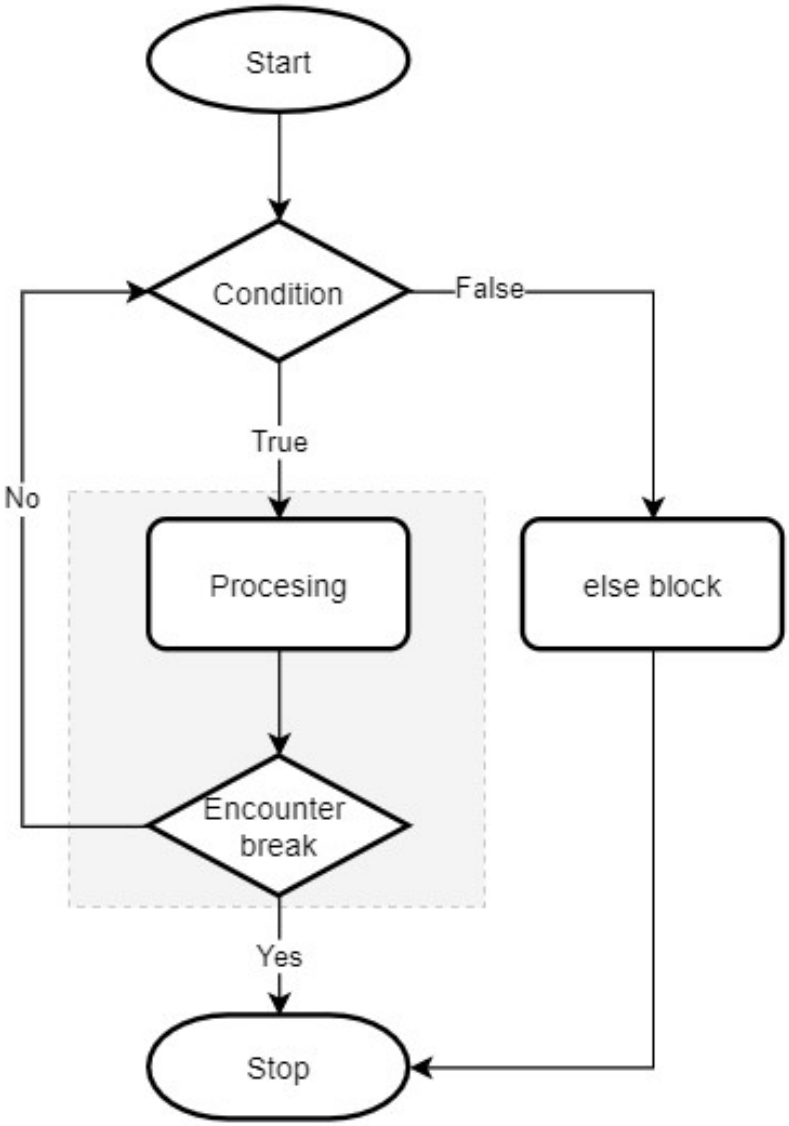
`break` – прерывание выполнения цикла

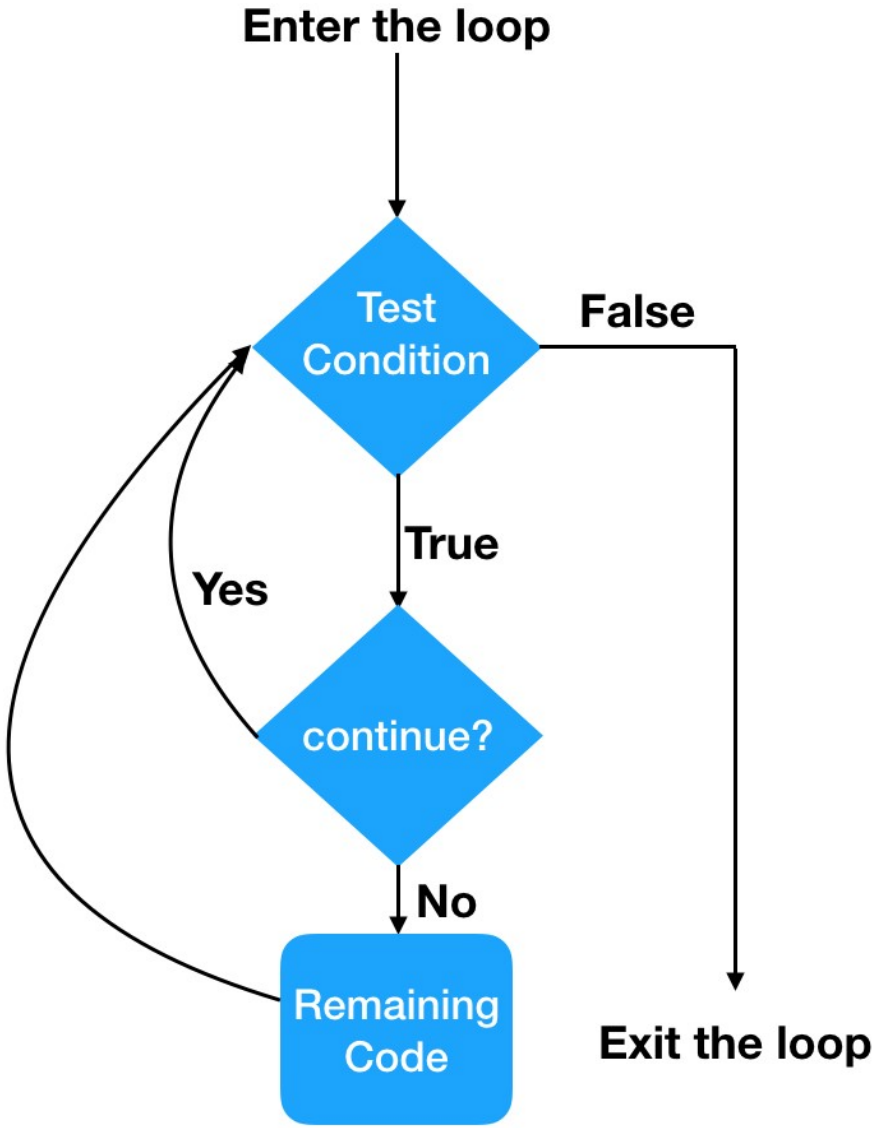
`continue` – прерывание текущей итерации цикла

`pass` – продолжение текущей итерации цикла

```
for number in range(10):
    if number == 3:
        continue
    if number == 5:
        pass
    if number == 7:
        break
    print(number)
# Output: 0 1 2 4 5 6
```

Управление потоком команд. Блок-схемы





Функции

```
def sayHello():
    print('Hello, world!') # body

sayHello() # call
```

Область видимости переменных ограничена блоком функции

```
x = 50
def func(x):
    print('x =', x)
    x = 2
    print('x =', x)

func(x) # Output: x = 50    x = 2
print('x =', x) # x = 50
```

Анонимные функции

Для создания анонимных функций используется ключевое слово **lambda**

```
import math

def sqroot(x):
    return math.sqrt(x)

square_rt = lambda x: math.sqrt(x)
type(square_rt) # function
sqroot(49) # 7
square_rt(49) # 7

add2 = lambda a,b: a+b
add2(5,7) # 12
```

Анонимные функции. Пример

```
def xprint(x, f):
    print(f(x))

s = 'hello'
xprint(s, lambda word: word.capitalize()+'!')
# Hello!

w = ', world'
xprint(s, lambda word: word.capitalize()+w+'!')
# Hello, world!

a = (lambda x,y: x+y)(1,2) # 3
```

Функция map

Функция **map** используется для применения функции к каждому элементу итерируемой последовательности (списка или др.)

```
map(fun, iterable[, iterable2, iterable3, ...])
```

```
def sq(x)
    return x*x

nums = [1,2,3,4]
result = list(map(sq, nums)) # [1,4,9,16]
result = list(map(lambda x: x*2+3,nums)) # [5,7,9,11]
nums2 = [10,20,30,40]
result = map(lambda x,y: x+y,nums,nums2) # [11,22,33,44]
s = ['rat','cat','bat']
result = list(map(list, s))
# [['r','a','t'],['c','a','t'],['b','a','t']]
```

Функция reduce

Функция **reduce** последовательно применяет функцию к элементам итерируемой последовательности, сводя её к единственному значению

```
reduce(fun, iterable[, initializer])
```

где fun – функция двух аргументов x,y (x – текущий результат, y – текущий элемент)

```
import functools
nums = [1,2,3,4]
result = functools.reduce(lambda x,y: x+y, nums) # 10
# It's equivalent to (((1+2)+3)+4)
result = functools.reduce(lambda x,y: \
         x if (x > y) else y, nums) # 4
```

---

reduce считается устаревшей в Python 3, рекомендуется использовать циклы

Функция filter

Функция **filter** применяет функцию ко всем элементам итерируемой последовательности и возвращает итератор с теми объектами, для которых функция вернула True

---

```
filter(fun, iterable)
```

```
nums = [1,2,3,4]
result = list(filter(lambda x: x%2==0,nums)) # [2,4]

s = 'spameggs'
vowels = ['a','e','i','o','u']
result = list(filter(lambda c: c in vowels,s))
# ['a','e']
```

---

filter считается устаревшей в Python 3, рекомендуется использовать генераторы списков

Глобальные переменные

Зарезервированное слово **global** объявляется внутри функции

```
x = 50
def func():
    '''Prints a global variable.
    Some description goes here.'''

    global x
    print('x =', x)
    x = 2
    print('x =', x)

func() # Output: x=50    x=2
print('x =', x) # x=2
```

Глобальные переменные во вложенных функциях

```
def f():
    city = "Hamburg"
    def g():
        global city
        city = "Geneva"
    print("Before calling g: " + city)
    g()
    print("After calling g: " + city)
f()
print("Value of city in outer scope: " + city)
# Before calling g: Hamburg
# After calling g: Hamburg
# Value of city in outer scope: Geneva
```

global внутри вложенной функции g не изменяет значение переменной city в функции f

Нелокальные переменные

```
def func_outer():
    x = 2
    print('x =', x)
    def func_inner():
        nonlocal x
        x = 5

    func_inner()
    print('x =', x)

func_outer()
# x = 2
# x = 5
```

Ключевое слово `nonlocal` используется только во вложенных функциях

Нелокальные переменные. Пример

```
def f():
    city = "Munich"
    def g():
        nonlocal city
        city = "Zurich"
    print("Before calling g: " + city)
    g()
    print("After calling g: " + city)

city = "Stuttgart"
f()
print("'city' in outer scope: " + city)
# Before calling g: Munich
# After calling g: Zurich
# 'city' in outer scope: Stuttgart
```

Переопределение глобальных переменных

Объявление локальной переменной с тем же именем, что и у глобальной переменной, сделает недоступной (“спрячет”) глобальную переменную

```
x = 10
def f():
    print(x)

def g():
    print(x)
    x += 1 # assignment declares x in g

f() # 10 (f uses global x)
g() # error: g declares local x
# and global x is referenced before assignment
```

Передача параметров в функцию

В Python существует единственный способ передачи параметров в функции – call by sharing
В функцию передаётся копия ссылки на объект, т.е. параметры внутри функции – это ссылки на параметры, указанные при вызове

```
def f(a, b):
    a = a + b
    return a

x = 1; y = 2
z = f(x,y) # z=3, x=1, y=2
x = [1,2]; y = [3,4]
z = f(x,y) # z=[1,2,3,4], x=[1,2], y=[3,4]
x = (1,2); y = (3,4)
z = f(x,y) # z=(1,2,3,4), x=(1,2), y=(3,4)
```

Передача параметров в функцию. Пример 1

```
def add_to_list(cities):
    print(cities)
    cities = cities + ["Birmingham","Bradford"]
    print(cities)

locations = ["London","Leeds","Glasgow","Sheffield"]
add_to_list(locations)
# ['London','Leeds','Glasgow','Sheffield']
# ['London','Leeds','Glasgow','Sheffield',
#  'Birmingham','Bradford']

print(locations)
# ['London','Leeds','Glasgow','Sheffield']
```

Передача параметров в функцию. Пример 2

```
def add_to_list2(cities):
    print(cities)
    cities.extend(["Birmingham","Bradford"])
    print(cities)

locations = ["London","Leeds","Glasgow","Sheffield"]
add_to_list2(locations)
# ['London','Leeds','Glasgow','Sheffield']
# ['London','Leeds','Glasgow','Sheffield',
#  'Birmingham','Bradford']

print(locations)
# ['London','Leeds','Glasgow','Sheffield',
#  'Birmingham','Bradford']
```

Передача параметров в функцию. Пример 3

```
def add_to_list2(cities):
    print(cities)
    cities.extend(["Birmingham","Bradford"])
    print(cities)

locations = ["London","Leeds","Glasgow","Sheffield"]
add_to_list2(locations[:])
# ['London','Leeds','Glasgow','Sheffield']
# ['London','Leeds','Glasgow','Sheffield',
#  'Birmingham','Bradford']

print(locations)
# ['London','Leeds','Glasgow','Sheffield']
```

Значения параметров по умолчанию

```
def hello(name="everybody"):
    """ Greets a person """
    print("Hello " + name + "!")

hello("Peter")
hello()
# Hello Peter!
# Hello everybody!
```

```
def sumsub(a, b, c=0, d=0):
    return a-b+c-d

print(sumsub(12, 4)) # 8
print(sumsub(42, 15, d=10)) # 17
```

Использование изменяемых объектов по умолчанию

```
def spammer(bag=[]):
    bag.append("spam")
    return bag

spammer() # ['spam']
spammer() # ['spam', 'spam']
spammer() # ['spam', 'spam', 'spam']
```

Значения по умолчанию создаются лишь однажды, когда функция определяется

При вызове `spammer()` параметр `bag` присваивается атрибуту `__defaults__` функции `spammer`. При каждом последующем вызове происходит добавление `'spam'` в `__defaults__`

Использование None как значение по умолчанию

Хорошей практикой считается не использовать изменяемые объекты в качестве значений по умолчанию параметров функций. Вместо этого рекомендуется использовать значение по умолчанию None с последуюшим созданием изменяемого объекта

```
def spammer(bag=None):
    if bag is None:
        bag = []
    bag.append("spam")
    return bag

spammer() # ['spam']
spammer() # ['spam']
```