Основы языка Python

А.Г. Трофимов
к.т.н., доцент, НИЯУ МИФИ

lab@neuroinfo.ru
http://datalearning.ru

Курс “Программирование в Python”

Июнь 2022

Содержание

- Введение
  Особенности и философия языка Python, области применения, средства разработки
- Типы данных в Python
  Особенности типизации, преобразование типов, строки и срезы
- Управление потоком команд
- Функции в Python
  Анонимные функции, область видимости имен, передача параметров, глобальные переменные, вложенные функции
- Модули м пакеты
- Коллекции и генераторы
  Списки, кортежи, множества, словари, функции и методы списков, создание и итерирование генераторов списков
- Декораторы функций
  Оператор декорирования, использование и создание декораторов
- Обработка исключений и менеджеры контекста

История языков программирования

- **50-е годы**
  Начало общения с компьютером, язык ассемблера, появление языка Фортран, эффективной альтернативы ассемблеру, предназначавшегося для математических вычислений
- **60-70-е годы**
  Обучение на языках ассемблера или Фортране требовало много сил, появление языков для обучения программированию Basic и Pascal, системное программирование – разработка языка C
- **80-е годы**
  Появление ООП, которое должно было упростить создание крупных промышленных программ, язык C++
- **90-е годы**
  Развитие персональных компьютеров и сети Интернет, потребность в новых технологиях и языках программирования, языки Java, Python, PHP
- **2000-е годы**
  Тенденция объединения технологий вокруг крупных корпораций, развитие языка C# на платформе .NET

История языков программирования

	Язык ассемблера	Микрокомпьютеры с очень ограниченными ресурсами.		50-ые
	Fortran	Математические расчеты.		
	Basic	Языки для обучения программированию.		60-70-ые
	Pascal			
	C	Системное программирование (драйвера и пр.)		ОС UNIX
	C++	Включает все возможности языка C, реализует ООП подход.	Потребность в больших программах - появился подход ООП.	80-ые
	Java	Крупные программы для бизнеса (ООП). Сложно написать плохую программу.	Потребность в программистах и переносимости. Автоматизировать кофемашину.	90-ые
	Python	Автоматизация рутинной деятельности (быстро), обучение программированию.		Персональные ПК, Интернет
	PHP	Разработка динамических сайтов.		
	C# (.NET)	Крупные программы для бизнеса (ООП). Много общего с Java. Зависимость от продуктов Microsoft.	Обобщение и объединение: собрать всё лучшее, что было до этого.	2000-ые

Язык Python

Python был разработан в конце 1989 года Гуидо ван Россумом во время рождественских каникул, когда его исследовательская лаборатория была закрыта и ему просто нечего было делать

Гуидо обожал телевизионную передачу Monty Python's Flying Circus (Летающий цирк Монти Пайтона), и для своего языка он выбрал имя Python

Guido van Rossum

Особенности Python

- **Простота**
  Python – простой и минималистичный язык, что позволяет сосредоточиться на решении задачи, а не на самом языке
- **Лёгкость в освоении**
  Python обладает исключительно простым удобочитаемым синтаксисом
- **Свобода использования и открытость**
  Python – пример свободного и открытого программного обеспечения FLOSS (Free/Libre and Open Source Software)
  Свободное распространение и использование, открытые исходные коды, возможность вносить изменения
  Python был создан и постоянно улучшается сообществом, которое хочет сделать его лучше

Особенности Python

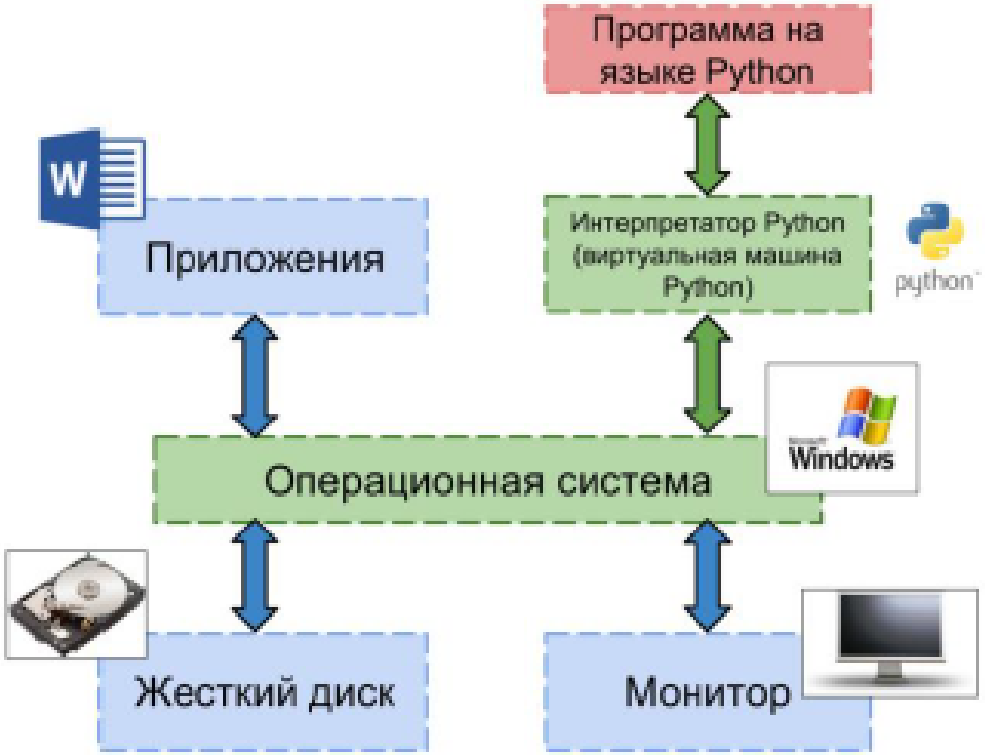
- **Язык высокого уровня**
  Не придётся отвлекаться на низкоуровневые процедуры управления памятью, файловой системой и пр.
- **Кросс-платформенность**
  Python можно использовать в GNU/Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga,AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE и даже на PocketPC
- **Интерпретируемый язык**
  Python не требует компиляции в бинарный код

Особенности Python

- **Объектно-ориентированный язык**
  Python поддерживает как процедурно-ориентированное, так и объектно-ориентированное программирование
- **Расширяемый язык**
  Из программы на Python может быть вызвана программа, написанная на языке C или C++ (например, для скрытия части алгоритма или повышения быстродействия)
- **Встраиваемый язык**
  Код Python'а можно встраивать в программы на C/C++
- **Обширные библиотеки**
  Стандартная библиотека Python предоставляет массу возможностей, в т.ч. для работы со строками, коллекциями, файлами и т.д., написано множество специальных библиотек

Python – интерпретируемый язык

Для запуска программ на языке Python необходима программа-интерпретатор (виртуальная машина) Python
Интерпретатор скрывает от Python-программиста все особенности операционной системы, что обеспечивает кросс-платформенность



Области применения Python

- Системное программирование
- Разработка программ с графическим интерфейсом
- Разработка динамических веб-сайтов
- Интеграция компонентов
- Разработка программ для работы с базами данных
- Быстрое создание прототипов
- Разработка программ для научных вычислений
- Разработка игр
- ...

Области применения Python

Где используется Python?

- Компания Google использует Python в своей поисковой системе и оплачивала труд создателя Python Гвидо ван Россума
- Компании Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm и IBM используют Python для тестирования аппаратного обеспечения
- Служба коллективного использования видеоматериалов YouTube в значительной степени реализована на Python
- NSA (National Security Agency) использует Python для шифрования и анализа разведданных
- Компании JPMorgan Chase, UBS, Getco и Citadel применяют Python для прогнозирования финансового рынка
- Веб-фреймворк App Engine от компании Google использует Python в качестве прикладного языка программирования
- NASA, Los Alamos, JPL и Fermilab используют Python для научных вычислений

Философия Python

Разработчики языка Python придерживаются философии программирования, называемой "The Zen of Python":

- Beautiful is better than ugly (Красивое лучше, чем уродливое)
- Explicit is better than implicit (Явное лучше, чем неявное)
- Simple is better than complex (Простое лучше, чем сложное)
- Complex is better than complicated (Сложное лучше, чем запутанное)
- Flat is better than nested (Плоское лучше, чем вложенное)
- Sparse is better than dense (Разреженное лучше, чем плотное)
- Readability counts (Читабельность имеет значение)
- ...

Полный текст выдаётся интерпретатором Питона по команде

```
import this
```

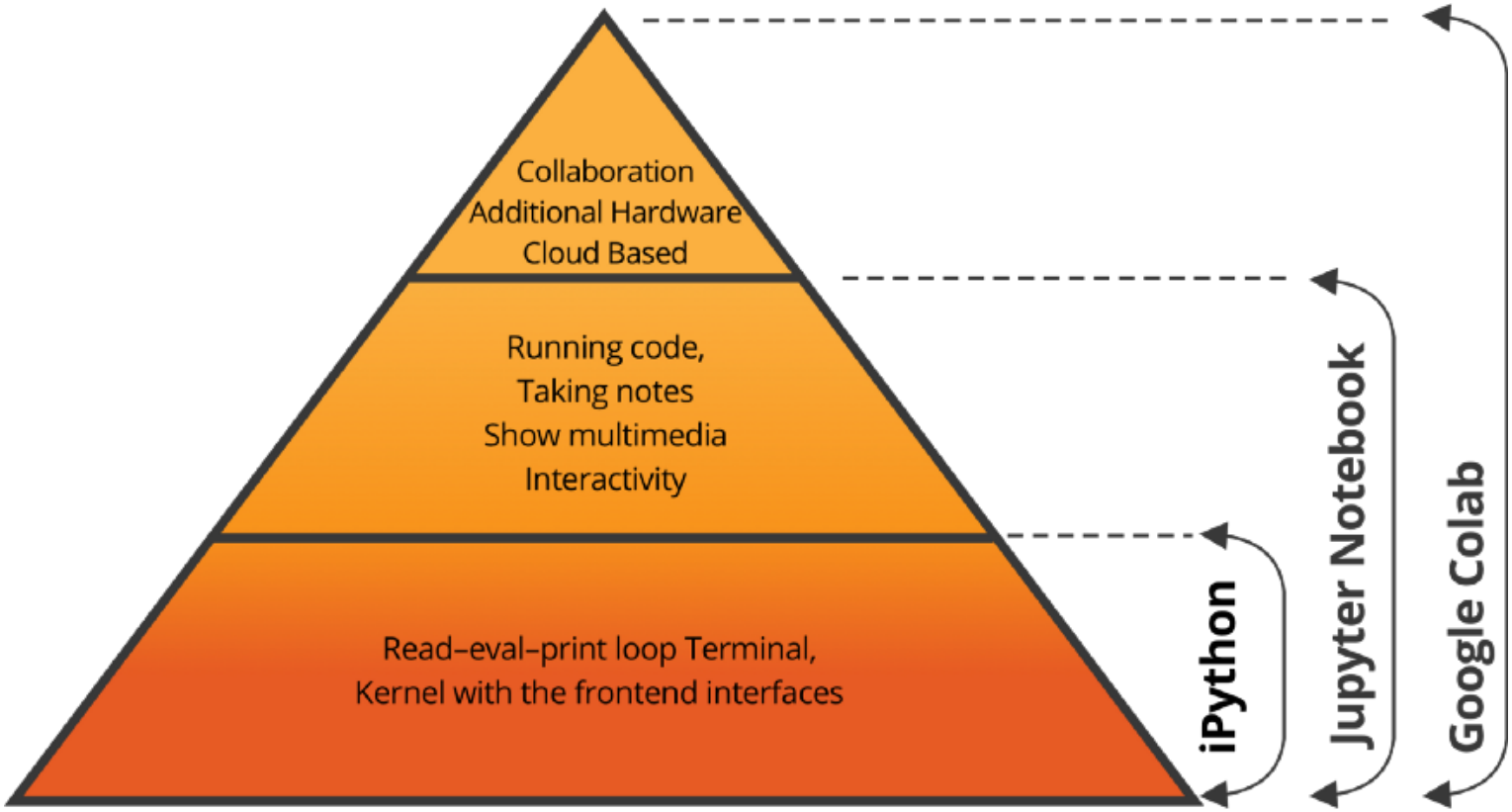
Что говорят программисты

- *Эрик С. Рэймонд* – автор работы «Собор и Базар», а также человек, который ввёл термин “Open Source”. Он говорит, что Python стал его любимым языком программирования
- *Брюс Экель* – автор книг «Думаем на Java» и «Думаем на C++». Он утверждает, что ни на одном языке программирования его работа не была столь эффективной, как на Python. Кроме того, он считает, что Python – это, пожалуй, единственный язык, стремящийся облегчить жизнь программисту
- *Питер Норвиг* – автор книг по программированию на Lisp, директор по качеству поиска в Google. Он говорит, что Python всегда был неотъемлемой частью Google. Python – один из официальных языков разработки Google (наряду с C++ и Java)

Среды разработки Python (IDE)

- **Текстовые редакторы с поддержкой Python:**
  - Eclipse + PyDev
  - Microsoft Visual Studio
  - Atom
  - GNU Emacs
  - ...
- **IDE, разработанные для Python:**
  - PyCharm
  - Spyder
  - Thonny
  - ...
- **Веб-ориентированные IDE:**
  - Jupyter Notebook (https://jupyter.org)
  - Google Colab (https://colab.research.google.com)

Jupyter Notebook vs Google Colab



Google Colab: большинство библиотек предустановлено, файлы хранятся в облаке, возможность коллаборации с другими разработчиками

PEP

PEP (Python Enhancement Proposal) – документ, содержащий рекомендации по написанию кода на Python

Самый известный PEP – PEP8, создан на основе рекомендаций Гуидо ван Россума. Основная цель PEP8 – улучшить читабельность и логичность кода на Python

Содержание PEP8:

- Внешний вид кода
- Комментарии
- Контроль версий
- Соглашения по именованию
- Общие рекомендации

Примеры рекомендаций PEP8

- Избегайте использования пробелов внутри круглых, квадратных или фигурных скобок

```
spam(ham[1], {eggs: 2}) # good
spam( ham[ 1 ], { eggs: 2 } ) # bad
```

- Никогда не используйте символы l (буква “эль”), O (буква “о”) или I (буква “ай”) как однобуквенные идентификаторы
- Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания
- Имена классов должны следовать соглашению CapWords
- Сравнения с None должны обязательно выполняться с использованием операторов is или is not, а не с помощью ==
- Всегда используйте def, а не лямбда-выражения
- Не сравнивайте логические типы с True и False с помощью ==

```
if greeting: # good
if greeting == True: # bad
```

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)
```

Однострокoвые 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

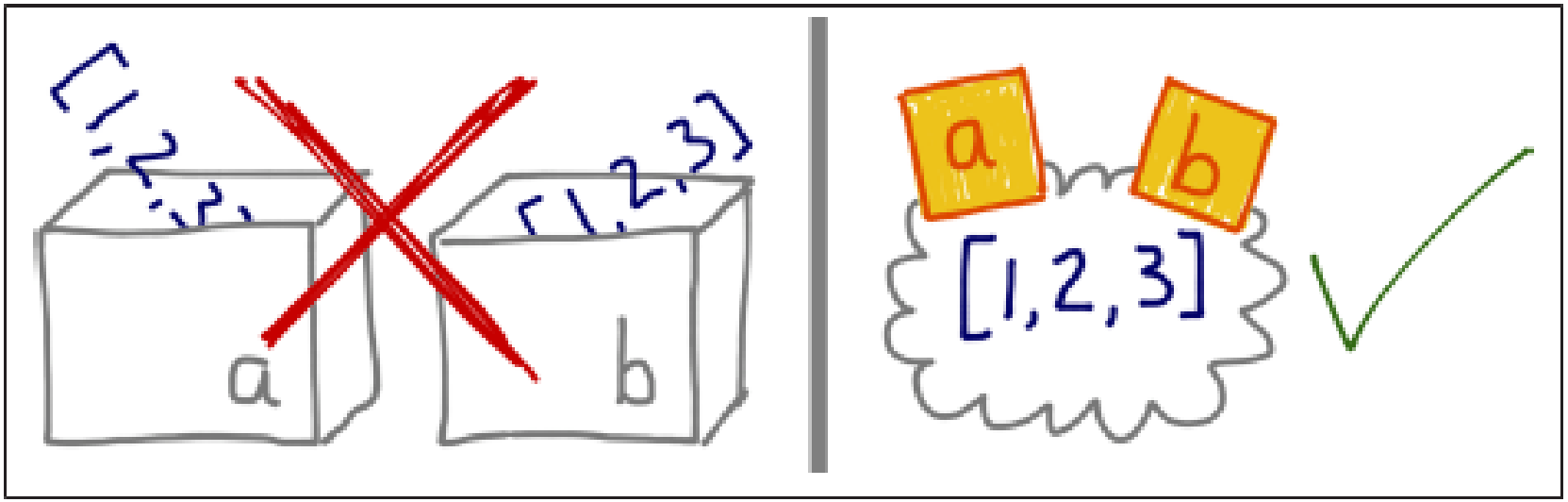
Каждый объект имеет три атрибута:
идентификатор, значение и тип

```
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

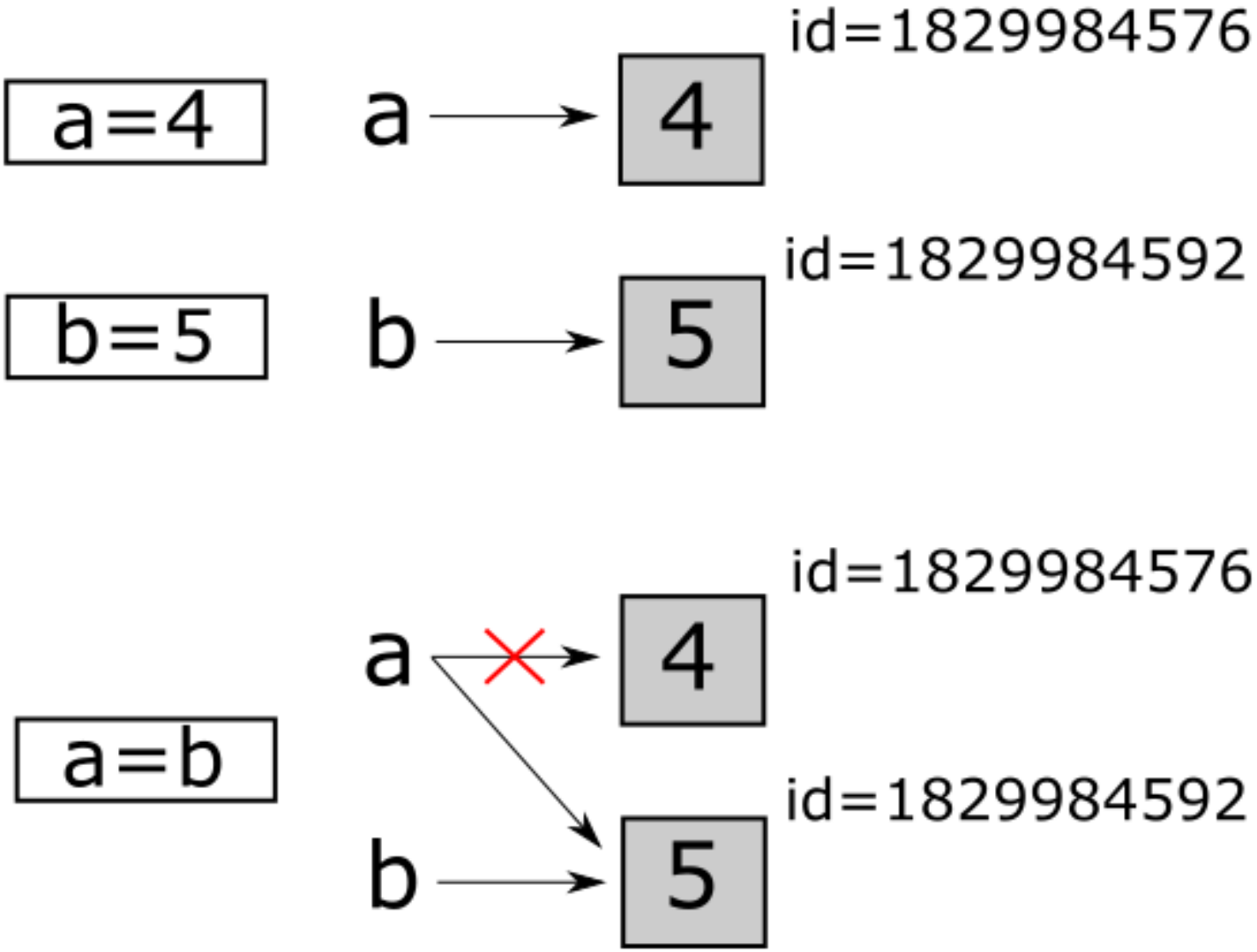
- Неизменяемые типы (immutable)
  bool, int, float, complex, str, tuple, frozen set
  Неизменяемость типа данных означает, что созданный объект данного типа больше невозможно изменить

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

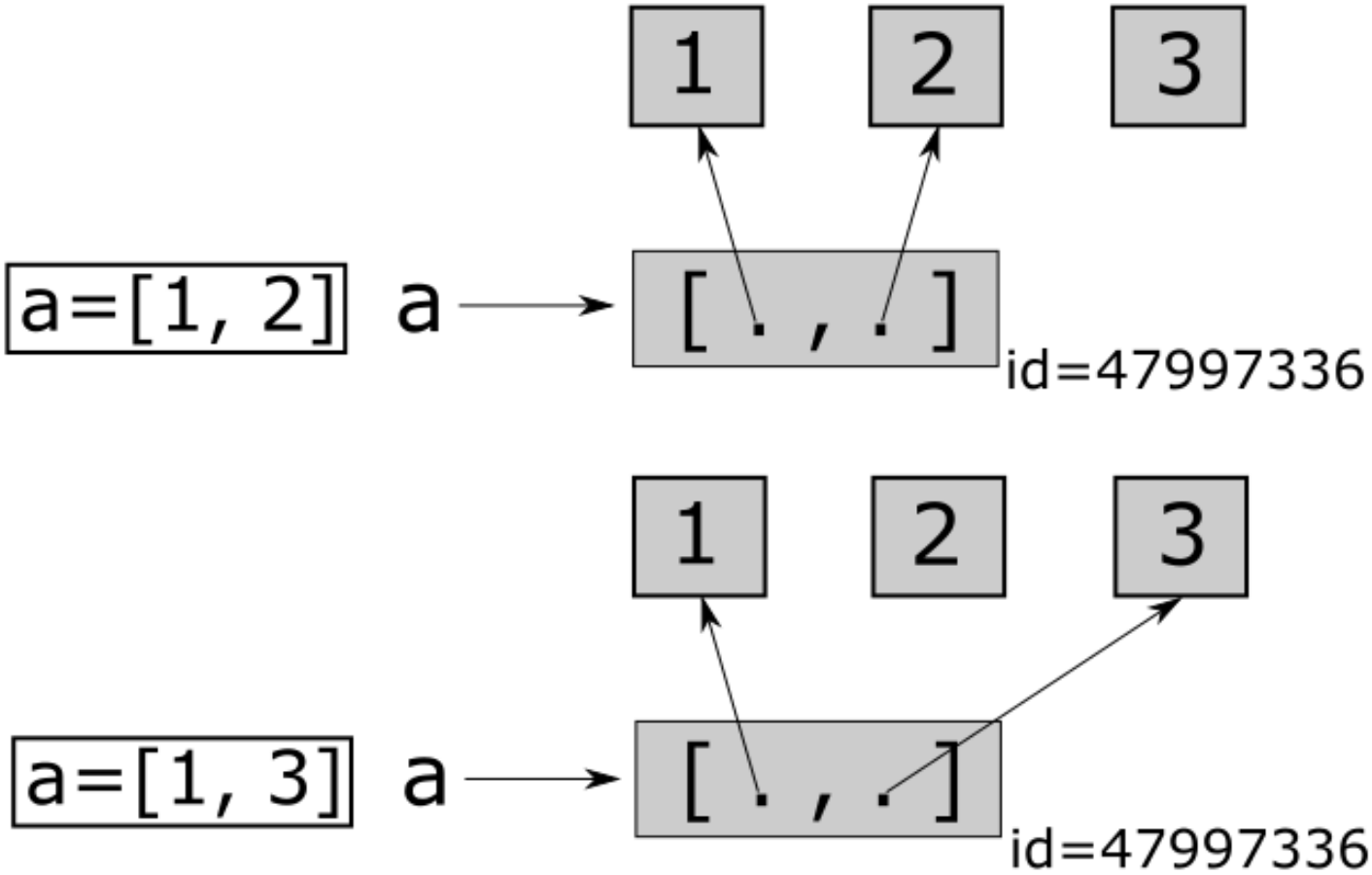
- Изменяемые типы (mutable)
  list, set, dict
  Значение объекта можно менять

```
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

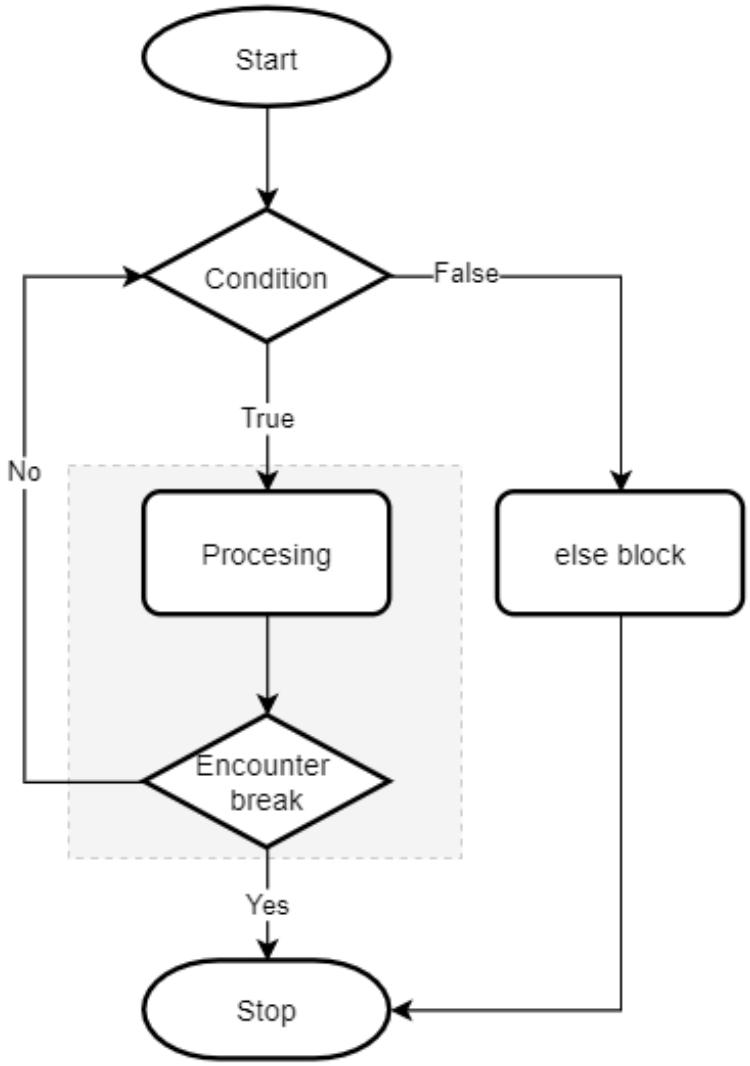
break – прерывание выполнения цикла

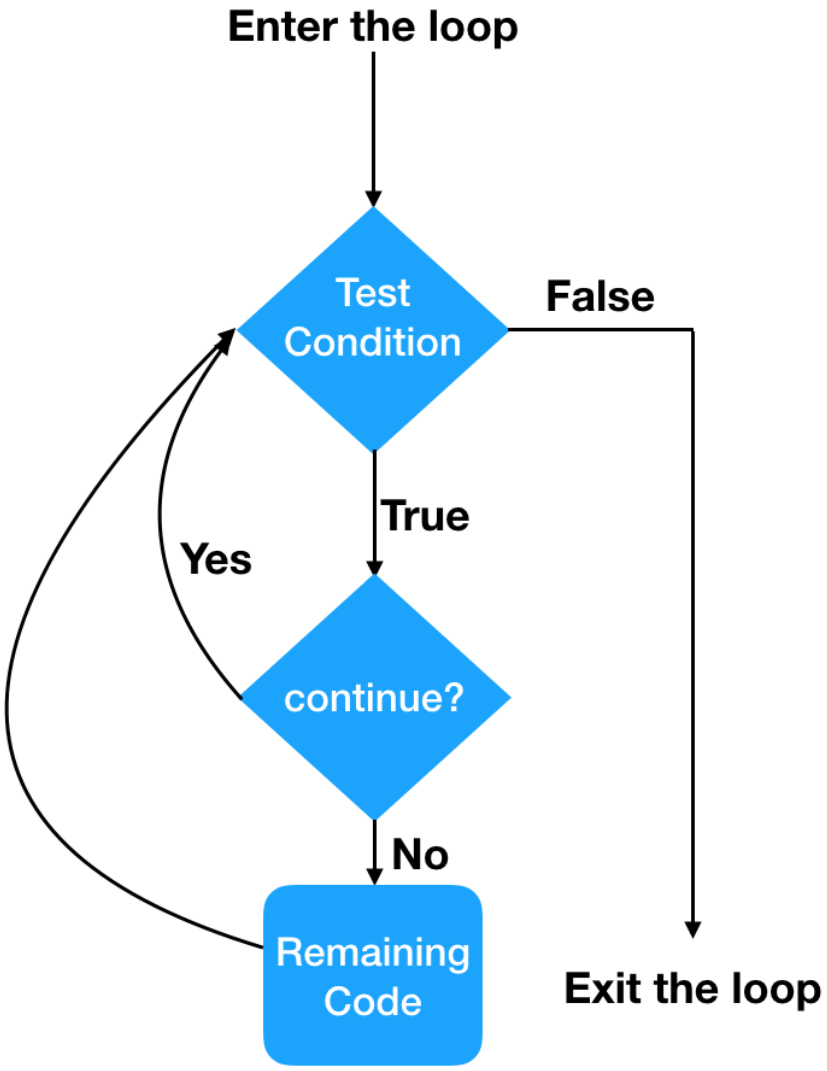
continue – прерывание текущей итерации цикла

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']
```

Модули (Modules)

Модуль – это файл .py с набором функций

mymodule.py:

```
def say_hello():
    print('Hello! It's module mymodule.py')

__version__ = '0.1'
# End of module mymodule.py
```

test.py:

```
import mymodule
mymodule.say_hello()
print ('Version', mymodule.__version__)
```

Подключение модулей

- Импорт всех функций и переменных модуля

```
import mymodule # import all functions
mymodule.say_hello()

import mymodule as mm # use alias
mm.say_hello()

from mymodule import * # bad style
say_hello()
```

- Импорт отдельных функций и переменных

```
from mymodule import say_hello,say_goodbye
say_hello()
```

Магический атрибут __all__

Список имен модуля, загружаемых через `from mymodule import *`, может быть определен в магическом атрибуте `__all__` модуля

mymodule.py:

```
__all__ = ['say_hello']
```

test.py:

```
from mymodule import *
say_hello() # ok
say_goodbye() # error: undefined name
```

Если атрибут `__all__` не определен, то подгружаются все методы и переменные модуля, кроме тех, которые начанаются с символа подчеркивания (_)

Подключение модулей. Замечания

- В момент начала работы программы Python загружает в память модуль `__builtins__`, содержащий базовые функции и переменные
- Список подгружаемых имен модуля возвращает функция **dir**:

```
import mymodule
print(dir(mymodule))
```

- Загрузка модуля возможна только один раз, при первом вызове **import**. Для повторной загрузки модуля используется библиотека `importlib`

```
import importlib
importlib.reload(mymodule)
mymodule.say_hello()
```

Исполняемый код в модулях

Модуль может содержать исполняемый код

fibo.py:

```
def fib(n):
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result

print("It's module fibo") # executable statement
```

Код модуля выполняется только один раз при первом подключении модуля. При повторной загрузке модуля (с помощью `importlib.reload`) код модуля выполняется повторно

Выполнение модуля как скрипта

У каждого модуля (файла) имеется магический атрибут `__name__`, которому присваивается имя модуля при импорте

```
import math
print(math.__name__) # math
```

Когда файл исполняется как скрипт, значению `__name__` присваивается строка `'__main__'`

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

if __name__ == "__main__":
    # do if called as a script
    print(sq(5))
```

Пакеты (Packages)

Пакеты – это каталоги с модулями и специальным файлом `__init__.py`, который показывает Python, что этот каталог особый, так как содержит модули Python

Пакеты – удобный способ иерархической организации модулей

```
| — <some folder in sys.path>/
| |——— world/
|       |——— __init__.py
|       |——— asia/
|       |     |——— __init__.py
|       |     |——— india/
|       |         |——— __init__.py
|       |         |——— foo.py
|       |——— africa/
|             |——— __init__.py
|             |——— madagascar/
```

Пакеты. Пример

Например, пакет sound содержит в себе три модуля: wavread.py, wfilter.py и surround.py

```
sound
|-- __init__.py
|-- wavread.py
|-- wfilter.py
|-- surround.py
```

```
# import module wavread.py from package sound
import sound.wavread
w = sound.wavread.read(fname) # usage
import sound.wfilter as filt
res = filt.equalizer(w, n)
from sound import surround
res = surround.process(w)
```

Файл __init__.py

Файл `__init__.py` может быть пустым или может содержать переменную `__all__`, хранящую список модулей, который импортируется при загрузке через конструкцию
`from sound import *`

Пример файла `__init__.py`:

```
__all__ = ["wavread", "wfilter", "surround"]
```

Файл `__init__.py` позволяет инициализировать переменные пакета, подключить модули и пр.

Пример файла `__init__.py`:

```
import sys
if sys.version_info[0] >= 3:
    ...
```

Списки (Lists)

Список – это упорядоченная изменяемая коллекция объектов произвольных типов

Список – изменяемый тип данных, т.е. его можно модифицировать. Элементами списка могут быть любые объекты

Создание списков

```
a = [2,5,7]
a = [] # empty list
type(a) # list
a = list('list') # ['l','i','s','t']
a = ['s', 'p', ['isok'], 2]
a = [2,5,7]
b = a+[3,3] # [2,5,7,3,3]
b = [1,2]*3 # [1,2,1,2,1,2]
```

Доступ к элементам списка. Срезы

```
a = [1,3,8,7]
a[0] # 1
a[10] # error
a[:] # [1,3,8,7]
a[1:] # [3,8,7]
a[:3] # [1,3,8]
a[::2] # [1,8]
a[::-1] # [7,8,3,1]
a[:-2] # [1,3]
a[-2::-1] # [8,3,1]
a[1:4:-1] # []
a[10:20] # []
a[i:j:step]
```

Функции списков

```
a = [1,3,8,7]
len(a) # 4
min(a) # 1
max(a) # 8
sum(a) # 19
sorted(a) # [1,3,7,8]
del a[:-2] # [8,7]

a = [1,3,8,7]
a[1:3] = [0,0,0,0]
# [1,0,0,0,0,7]

if 7 in a:
    print('Element is in list')
```

Методы списков

```
a = [1,8,7,3]
a.sort() # a = [1,3,7,8]

a = [1,8,7,7,3]
a.count(7) # 2
a.count(27) # 0
a.index(8) # 1
a.append([15,20]) # [1,8,7,7,3,[15,20]]
a.extend([15,20]) # [1,8,7,7,3,[15,20],15,20]

a = [1,8,7,3]
a.reverse() # [3,7,8,1]
a.remove(7) # [3,8,1]
a.clear() # []
...
```

Псевдонимы (Aliases) в Python

```
a = [1,8,7,3]
b = a # reference to a
b[0] = 15
print(a) # [15,8,7,3]
```

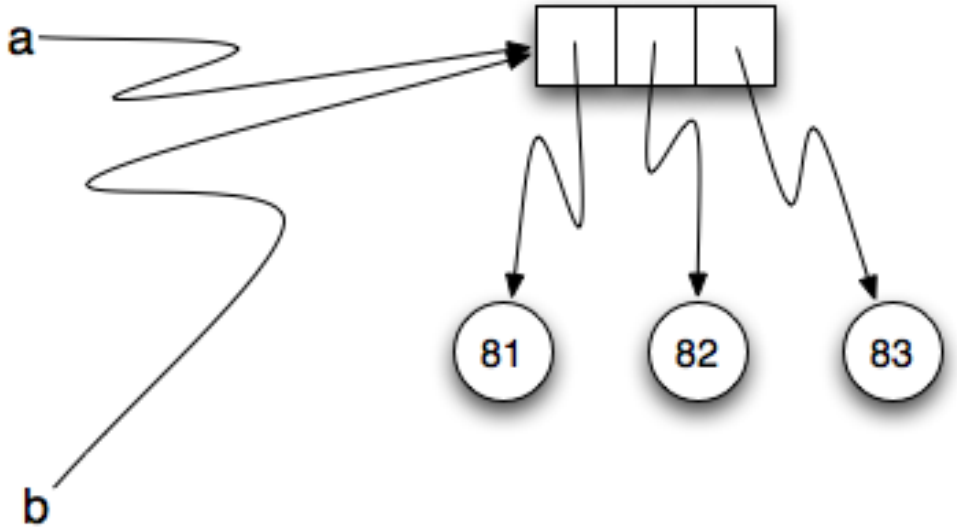
В Python две переменные называются псевдонимами, если они ссылаются на одинаковые адреса памяти

Как проверить, ссылаются ли переменные на один и тот же объект?

```
a = [1,8,7,3]
b = a
c = b is a # True
b = [1,8,7,3]
c = b is a # False
```

Псевдонимы. Иллюстрация

```
a = [81,82,83]
b = a
b == a # True
b is a # True
id(b) == id(a) # True
```



Копирование списков

- **Поверхностное копирование (shallow copy)**
  Создается новый объект, но он будет заполнен ссылками на элементы, которые содержались в оригинале

```
a = [4,3,[2,1]]
b = a[:] # b is a copy
b is a # False
b[2][0]=-100
print(a) # [4,3,[-100,1]]
```

- **Глубокое копирование (deep copy)**
  Создается новый объект и рекурсивно создаются копии всех объектов, содержащихся в оригинале

```
import copy
a = [4,3,[2,1]]
b = copy.deepcopy(a)
b[2][0]=-100
print(a) # [4,3,[2,1]]
```

Списки и строки

```
s = 'hello'
a = list(s) # ['h','e','l','l','o']
''.join(a) # 'hello'
' '.join(a) # 'h e l l o'
'**'.join(a) # 'h**e**l**l**o'
'h*e*l*l*o'.split('*') # ['h','e','l','l','o']
list(str(123)) # ['1','2','3']
```

Итерирование строк и списков

```
num = [0.8, 7.0, 6.8, -6]
for i in num:
    print(i)
s = 'hello'
for ch in s:
    print(ch)
```

Интернирование

```
a = 'abc'
b = a # not a copy, just reference to a
b = a[:]; b is a # True
c = str(a); c is a # True
```

Копирования объектов b и c не происходит
Та же ситуация с другими неизменяемыми типами: int, float, bytes, frozenset и др.

```
a = 'abc'; b = 'abc'
b is a # True
```

Создание нового объекта b не происходит, вместо этого создаётся ссылка на объект a
Интернирование (interning) – механизм оптимизации кода, при котором хранится лишь одна копия из множества одинаковых объектов

Диапазоны (Ranges)

Диапазон – последовательность целых чисел

```
r = range(i, j) # i, i+1, i+2, ..., j-1
r = range(i, j, step)
r = range(2, 20, 2) # 2 4 6 8 10 12 14 16 18
r = range(20, 2, -2) # 20 18 16 14 12 10 8 6 4
type(r) # range
```

Итерирование диапазонов

```
r = range(5) # 0 1 2 3 4
for i in r:
    print(i)
a = list(r) # [0,1,2,3,4]
for i in range(len(a)):
    print(a[i])
```

Итерируемые классы и итераторы

Инструкция **for–in** используется для итерирования по элементам последовательностей либо других типов, поддерживающих итерирование

Классы, поддерживающие итерирование, (iterables) реализуют метод `__iter__`, возвращающий итератор

Классы-итераторы (iterators) реализуют метод `__next__`

```
s = 'abc' # s is iterable
for c in s:
    print(c)

# user iteration
it = iter(s) # get iterator (type str_iterator)
for i in range(len(s)):
    print(next(it))
```

Генераторы списков (List Comprehensions)

Генераторы списков служат для создания новых списков на основе существующих

```
a = [c*3 for c in 'list']
# ['lll','iii','sss','ttt']
a = [c*3 for c in 'list' if c != 'i']
# ['lll','sss','ttt']
a = [i**2 for i in range(1,5)]
# [1,4,9,16]
b = [i+2 for i in a]
# [3,6,11,18]
b = [(i+2 if i%2==0 else 0) for i in a]
# [0,6,0,18]
# value = i+2 if i%2==0 else 0
# b = [value for i in a]
```

Функции-генераторы

range() – генератор последовательности целых чисел

Генераторы не хранят значения в памяти, а генерируют их на лету

Функции-генераторы возвращают значение, используя yield

```
def my_range(first=0.0, last=1.0, step=0.1):
    number = first
    while number < last:
        yield number
        number += step

for x in my_range():
    print(x)
# 0.0 0.1 ... 0.9
```

Ключевое слово yield

При использовании ключевого слова yield функция возвращает значение без уничтожения локальных переменных, кроме того, при каждом последующем вызове функция начинает своё выполнение с оператора yield

```
type(my_range) # function
gen = my_range() # type(gen) - generator
for x in gen:
    print(x)
# 0.0 0.1 ... 0.9
for x in gen:
    print(x)
# Empty
```

Генераторы применяются в тех случаях, когда нет необходимости сохранять всю последовательность и промежуточные значения в памяти

Итерирование генератора

Для итерирования генератора служит функция **next**

```
gen = my_range()
next(gen) # 0.0
next(gen) # 0.1
...
next(gen) # 0.9
next(gen) # error
```

Функция-генератор может быть создана при помощи круглых скобок ()

```
gen = (x for x in [1,2,3])
gen = (x+2 for x in my_range(0,0.5))
for i in gen:
    print(i)
# 2.0 2.1 2.2 2.3 2.4
```

Кортежи (Tuples)

Кортеж – это упорядоченная неизменяемая коллекция объектов произвольных типов

Кортеж = неизменяемый список

Создание кортежей

```
a = (2,5,7)
a = 2,5,7 # parentheses are unnecesary
a = tuple() # empty tuple
a = () # empty tuple
type(a) # tuple
a = tuple('tuple') # ('t','u','p','l','e')
a = ('t', 'u', ['ple'], 2)
a = (2) # 2 (type int)
a = (2,) # (2,) (type tuple)
a = (1,2)*3 # (1,2,1,2,1,2)
```

Зачем нужны кортежи?

- Кортежи используются в тех случаях, когда набор значений не должен изменяться
- Кортежи занимают меньший объём памяти, чем списки

С помощью кортежей можно присваивать значения одновременно нескольким переменным:

```
(a,b,c) = (2,10,7)
# a=2, b=10, c=7
```

Если кортеж содержит изменяемые объекты, то их можно изменить:

```
a = (1,[5,7],'a')
a[1][0] = 12
# a = (1,[12,7],'a')
```

Функции и методы кортежей

Функции и методы кортежей аналогичны функциям и методам списков

```
a = (1,3,8,7)
len(a) # 4
min(a) # 1
max(a) # 8
sum(a) # 19
sorted(a) # (1,3,7,8)
del a[1] # error: tuple is immutable
a[1:3] # (3,8)
if 3 in a:
    print('Element is in tuple')
a.count(7) # 1
a.index(7) # 3
```

Множества (Sets)

Множество – это изменяемая неупорядоченная коллекция неизменяемых уникальных объектов

Создание множеств

```
a = {2,'abc',7,(2,3)}
a = {2,'abc',7,[2,3]} # error: list is mutable
a = set() # empty set
type(a) # set
a = set('hello') # {'h','e','l','o')
a = set([2,5,2,7]) # {2,5,7}
a = set( (2,5,2,7) ) # {2,5,7}
a = set(range(5)) # {0,1,2,3,4}

# remove duplicates from tuple/list
tuple(set( (3,6,3,5) )) # (3,5,6)
list(set([3,6,3,5])) # [3,5,6]
```

Операции над множествами

```
a = {2,'abc',7}
a.add(5) # {2,5,7,'abc'}
a.remove('abc') # {2,5,7}
a.remove('qwe') # error
a.discard('qwe') # do nothing if element doesn't exist

a = {2,5,7}
b = {5,7,12}
a.intersection(b) # a&b = {5,7}
b.intersection(a) # b&a = {5,7}
a.union(b) # a|b = {2,5,7,12}
a.difference(b) # a\b={2}
b.difference(a) # a\b={12}
if 5 in a: print('Element belongs to set')
```

Неизменяемые множества (Frozen sets)

Set – изменяемое множество

Frozen set – неизменяемое множество

```
a = {2,'abc',7,(2,3)}
b = frozenset(a)
type(b) # frozenset
c = set(b)
type(c) # set
c.add(12) # ok
b.add(12) # error: frozenset is immutable
b.remove(2) # error: frozenset is immutable
a == b # True
```

Отличие между Set и Frozen set такое же, как и между List and Tuple

Словари (Dictionaries)

Словарь – неупорядоченная изменяемая коллекция с произвольными ключами неизменяемого типа

Ключами словаря могут быть произвольные неизменяемые объекты (не обязательно одного типа)

```
a = {'key1':12, 'key2':15}
a = {101:12, 102:15}
a = {('s',5):12, (8,5):15}
a = dict() # empty dict
a = {} # empty dict
type(a) # dict
a = dict(key1=12, key2=15) # {'key1':12, 'key2':15}
# create from collection of tuples (key,value)
a = dict([('key1',12), ('key2',15)])
a = dict(( ('key1',12), ('key2',15) ))
a = dict({('key1',12), ('key2',15)})
```

Генераторы словарей (Dictionary Comprehensions)

Генераторы словарей служат для создания новых словарей на основе существующих списков

```
a = {c:c*3 for c in 'list'}
# {'i':'iii', 'l':'lll', 's':'sss', 't':'ttt'}
a = {c:c*3 for c in 'list' if c != 'i'}
# {'l':'lll', 's':'sss', 't':'ttt'}
a = {i:i**2 for i in range(1,5)}
# {1:1, 2:4, 3:9, 4:16}
b = ['abc', 'qwe', 'str']
a = {2*i:b[i] for i in range(len(b))}
# {0:'abc', 2:'qwe', 4:'str'}
b = [('key1', 12), ('key2', 15)]
a = {k:v for (k,v) in b}
# {'key1':12, 'key2':15}
```

Функции словарей

```
a = {'key1':12, 'key2':15, (1,2):18}
a['key1'] # 12
a[1] # error: key doesn't exist

len(a) # 3
del a[(1,2)]
# a={'key1':12, 'key2':15}

a['key1'] = [2,3]
# a={'key1':[2,3], 'key2':15}

if 'key1' in a:
    print('Key is in dict')
```

Методы словарей

```
a = {'key1':12, 'key2':15, (1,2): 18}
a.get('key1') # 12
a.get('s') # None (default)
a.keys() # 'key1', 'key2', (1,2)
a.values() # 12, 15, 18
a.items() # 'key1':12, 'key2':15, (1,2):18
b = a # b is a reference to a
b = a.copy() # b is a shallow copy of a
a.pop( (1,2) ) # 18, a= {'key1': 12, 'key2': 15}
a.clear() # {}
```

Итерирование словарей:

```
for (k,v) in a.items():
    print(k,':',v)
```

Функции с переменным числом параметров

```
def total(initial=5, *numbers, **keywords):
    count = initial
    for number in numbers:
        count += number
    for (k,v) in keywords.items():
        count += v
    return count

print(total(10, 1, 2, 3, vegetables=50, fruits=100))
```

Неключевые аргументы объединяются в кортеж
Ключевые аргументы объединяются в словарь

```
# numbers = (1,2,3)
# keywords = {'vegetables':50, 'fruits':100}
```

Функция zip

Функция **zip** упаковывает последовательности одинаковой длины в кортежи

```
a = [2,5,7]
s = 'qwe'
z = zip(a, s) # type(z) - zip
next(z) # (2,'q')
next(z) # (5,'w')
next(z) # (7,'w')
next(z) # error
for (i,c) in zip(a,s):
    print(i,':',c)

z = zip((1,2), 'ab', ['q','w'])
next(z) # (1,'a','q')
next(z) # (2,'b','w')
```

Оператор *

Оператор * распаковывает последовательность на отдельные элементы

```
a = [(1,'a'), (2,'b'), (3,'c')] # list
z = zip(*a) # zip
next(z) # (1,2,3)
next(z) # ('a','b','c')
```

```
def add3(a,b,c):
    return a+b+c

add3([1,4,5]) # error: missing arguments b,c
add3(*[1,4,5]) # 10
add3(*(1,4,5)) # 10
add3(*{1,4,5}) # 10
```

Оператор **

Оператор ** распаковывает словарь на отдельные элементы key:value

```
def add3(a,b,c):
    return a+b+c

d = {'a':1, 'b':2, 'c':3} # dict
add3(*d) # 'abc'
add3(**d) # 6
```

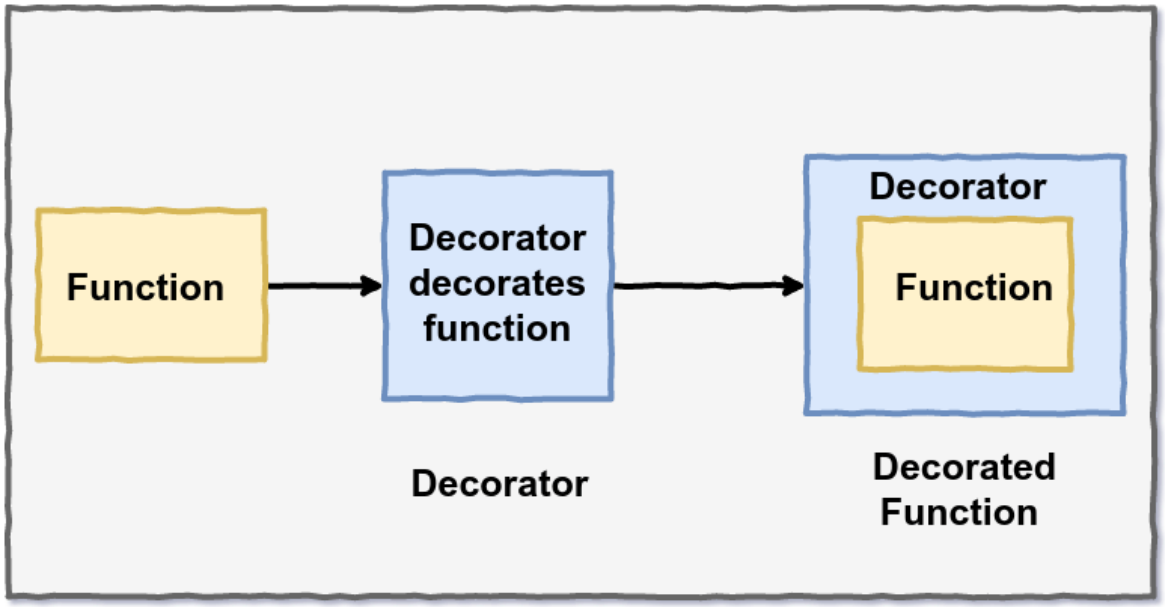
Пример использования:

```
def f(x, *args, **kwargs):
    kwargs['newArg'] = 5
    g(x, *args, **kwargs)
```

Декораторы

Декоратор – это обертка функции, которая позволяет изменить её поведение, не изменяя её код

Декоратор представляет собой функцию, которая получает на вход декорируемую функцию и возвращает декорированную функцию



Декораторы. Примеры

```
def my_decorator(func):
    def my_wrapper(a, b):
        print('Function', func.__name__, 'start')
        res = func(a, b)
        print('Result:', res)

    return my_wrapper

def add2(x, y):
    return x+y

add_decored = my_decorator(add2) # function
add_decored(2,5)
# Function add2 start
# Result: 7
```

Оператор декорирования функций

```
# assign to add2 decorator my_decorator
add2 = my_decorator(add2)
add2(2,5)
# Function add2 start
# Result: 7
```

Выражение `add2 = my_decorator(add2)` эквивалентно объявлению имени декоратора со знаком @ перед объявлением декорируемой функции

```
# another way to assign decorator my_decorator to add2
@my_decorator
def add2(a,b):
    return a+b

add2(2,5)
```

Декораторы. Примеры

```
def bold(func):
    def wrapper(who):
        print ("<b>")
        func(who)
        print ("</b>")
    return wrapper

def italic(func):
    def wrapper(who):
        print ("<i>")
        func(who)
        print ("</i>")
    return wrapper
```

Декораторы. Примеры

К функции может быть применено несколько декораторов

```
@italic
@bold
def hello(who):
    print ("Hello", who)

hello("World")
# <i>
# <b>
# Hello World
# </b>
# </i>
```

Декораторы. Примеры

```
import time
def measure_time(func):
    def wrapper(*arg):
        t = time.time()
        res = func(*arg)
        print(f"Took {time.time()-t} seconds to run")
        return res
    return wrapper

@measure_time
def func(n):
    sorted(range(n,0,-1))

func(10000)
# Took 0.0029933452606201 17 seconds to run
```

Ввод данных и обработка исключений

Функция **input** считывает строку, введенную пользователем

```
s = input('Enter string: ')
x = int(input('Enter integer: '))
int('abc') # error
int(12.5) # error
x = float(input('Enter float: '))
```

Для обработки исключений используется конструкция **try–except**

```
try:
    x = int(input('Enter integer: '))
except:
    print('Not integer entered')
```

try–except–else–finally

```
try:
    x = int(input('Enter integer: '))
    y = 1/x
except ValueError:
    print('Not integer entered')
except ZeroDivisionError:
    print('Division by zero')
except Exception:
    print('?')
else:
    # executes if no exception was raised
    print(y)
finally:
   # executes always
    print('Done!')
```

Блок finally

```
# without finally
try:
    run_code1()
except TypeError:
    run_code2()
other_code()
```

```
# with finally
try:
    run_code1()
except TypeError:
    run_code2()
finally:
    other_code()
```

Блок finally

```
try:
    x = 0
    y = 1/x # raises ZeroDivisionError,
#             which will be propagated to caller
finally:
    print('Bye!') # will be executed before
#                   ZeroDivisionError is propagated
```

```
try:
    run_code1()
except TypeError:
    run_code2()
    return None # The finally block is run before this
finally:
    other_code()
```

Оператор with

Конструкция `with—as` используется для оборачивания выполнения блока инструкций менеджером контекста

Менеджер контекста вызывает функции `__enter__()` при входе и `__exit__()` при выходе из контекста (в том числе, при возникновении исключения)

```
with A() as a:
    do_something()

# it is equivalent to:
# A.__enter__()
# do_something()
# A.__exit__()
```

Менеджер контекста используется, как правило, для инкапсуляции процессов инициализации и завершения

Создание менеджера контекста

```
class HelloContextManager:
    def __enter__(self):
        print("Entering the context...")
        return "Hello, World!"
    def __exit__(self, exc_type, exc_value, exc_tb):
        print("Leaving the context...")
        print(exc_type, exc_value, exc_tb)

with HelloContextManager() as hello:
    print(hello)
# Entering the context...
# Hello, World!
# Leaving the context...
# None None None
```

Параметры exc_type, exc_value, exc_tb метода __exit__() содержат информацию о возникшем исключении (None в случае успешного выхода из контекста)

Менеджер контекста для обработки исключений

`with...as` иногда более удобная конструкция, чем **`try...except...finally`**

```
class A:
    def __init__(self):
        self.x = 0
    def __enter__(self):
        return self
    def __exit__(self, exc_type, exc_value, exc_tb):
        if exc_type is ZeroDivisionError:
            return True # do not raise exception
        else:
            return False

with A() as a:
   print(f"inv={1/a.x}") # nothing will happen
```

Если метод `__exit__()` возвращает True, исключение не выбрасывается. В противном случае исключение будет выброшено

Декоратор @contextmanager

Менеджер контекста может быть создан с помощью декоратора @contextmanager, без необходимости создания класса

```
from contextlib import contextmanager

@contextmanager
def hello_context_manager():
    print("Entering the context...")
    yield "Hello, World!"
    print("Leaving the context...")

with hello_context_manager() as hello:
    print(hello)
# Entering the context...
# Hello, World!
# Leaving the context...
```

Monkey Patch

Monkey patch – подмена отдельных методов, атрибутов классов или функций с целью тестирования, создания заглушек или изменения функционала внешних библиотек

```
from contextlib import contextmanager
from time import time

@contextmanager
def mock_time():
    global time
    saved_time = time
    time = lambda: 42
    yield
    time = saved_time

with mock_time():
    print(f"Mocked time: {time()}") # 42
time() # 1652701904.2153995
```

Использование менеджера контекста для работы с файлами

Файл будет закрыт вне зависимости от того, что введёт пользователь

```
with open('newfile.txt','w',encoding='utf-8') as f:
    d = int(input())
    print('1 / {} = {}'.format(d,1/d), file=f)
```

Использование `try...except` вместе с `with...as`:

```
try:
    with open('example.txt','r') as file:
        contents = file.read()
        print(contents)
except:
    print ("Error opening file")
```

Множественные менеджеры контекста

Python поддерживает использование множественных контекстов

```
with A() as a, B() as b:
    do_something()

# rewrite lines in reverse order
with (
    open("input.txt") as in_file,
    open("output.txt", "w") as out_file
):
    for line in in_file:
        out_file.write(line[::-1])
```

Множественные менеджеры контекста, фактически, представляют собой вложенные менеджеры контекстов, помещенные в стек