Основы языка Python

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

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

Курс “Программирование в Python”

Май 2018

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 некуда было деваться

Гуидо обожал телевизионную передачу Monty Python's Flying Circus (Летающий цирк Монти Пайтона), и когда пришло время дать название своему языку, 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

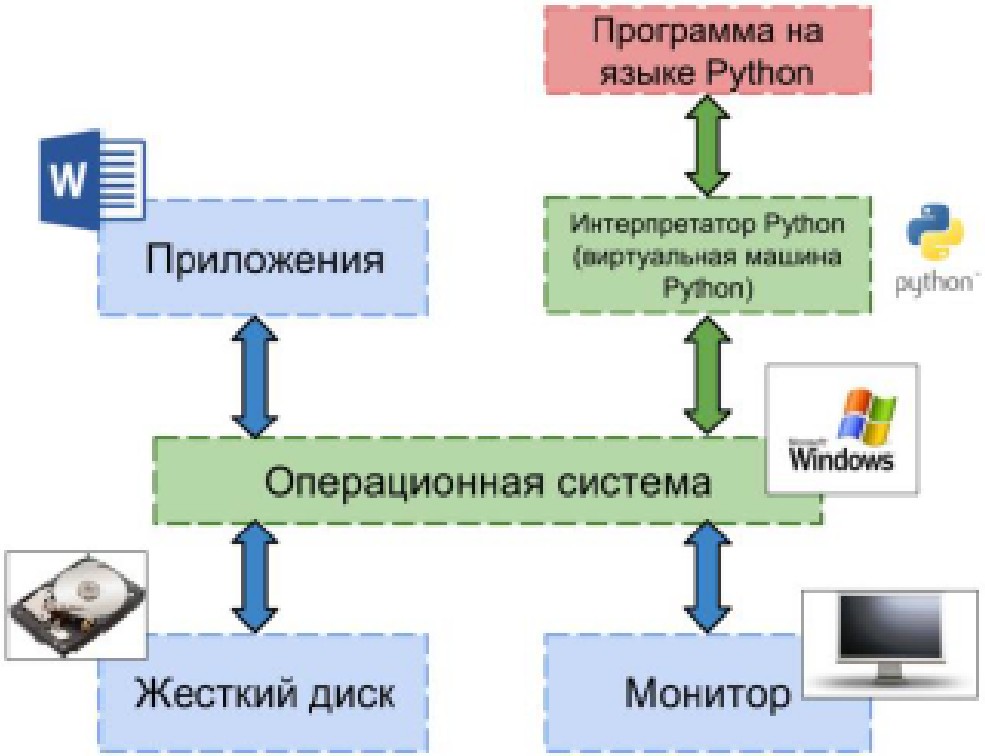
- **Интерпретируемый язык**
  Python не требует компиляции в бинарный код

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, начиная от поиска текста по шаблону и заканчивая сетевыми функциями, 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

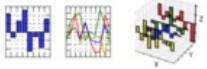
Области применения Python

Системное программирование

Игры, искусственный интеллект

Графический интерфейс

pandas



Программирование научных вычислений

python™

Веб-сценарии

Быстрое создание прототипов

Интеграция компонентов

Приложения баз данных

Где используется Python?

- Компания Google использует Python в своей поисковой системе и оплачивает 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 языка Lisp, а также директор по качеству поиска в Google. Он говорит, что Python всегда был неотъемлемой частью Google. Вы можете убедиться в этом, заглянув на страницу Google Jobs, на которой владение Python указано как требование для разработчиков программного обеспечения

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 предложений с одинаковым отступом

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

Перенос строки:

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

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):
  list – список, tuple – кортеж, 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

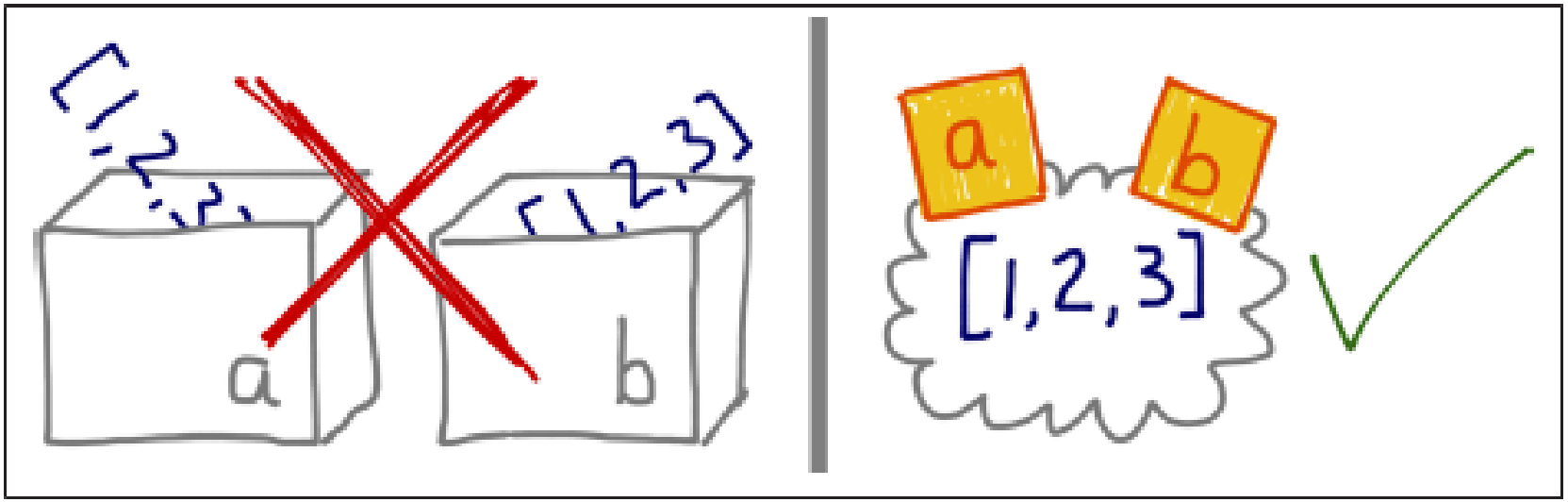
Каждый объект имеет три атрибута:
идентификатор, значение и тип

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

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

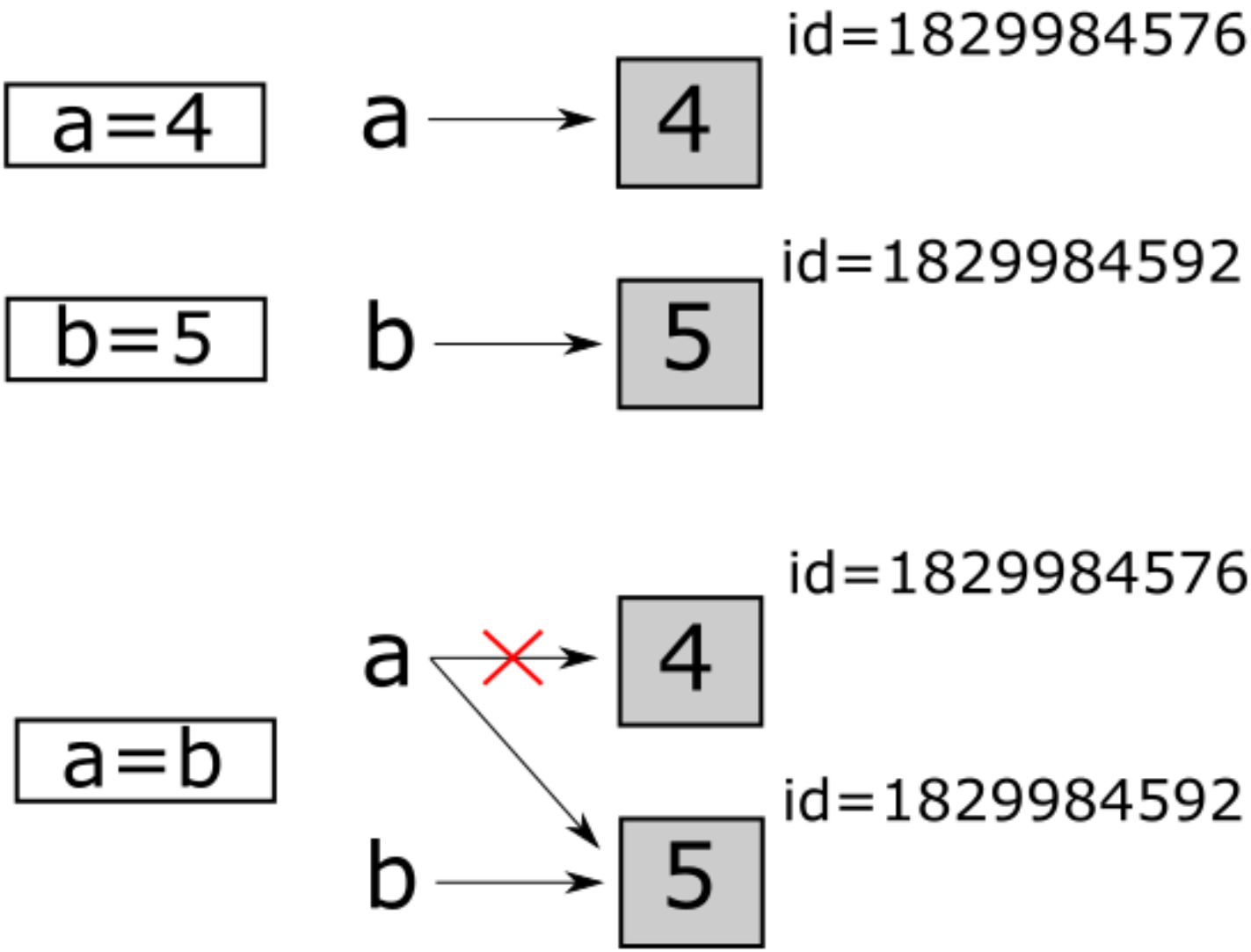
- Неизменяемые типы (immutable)
  bool, int, float, complex, str, tuple, frozen set
  Неизменяемость типа данных означает, что созданный объект данного типа больше невозможно изменить

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

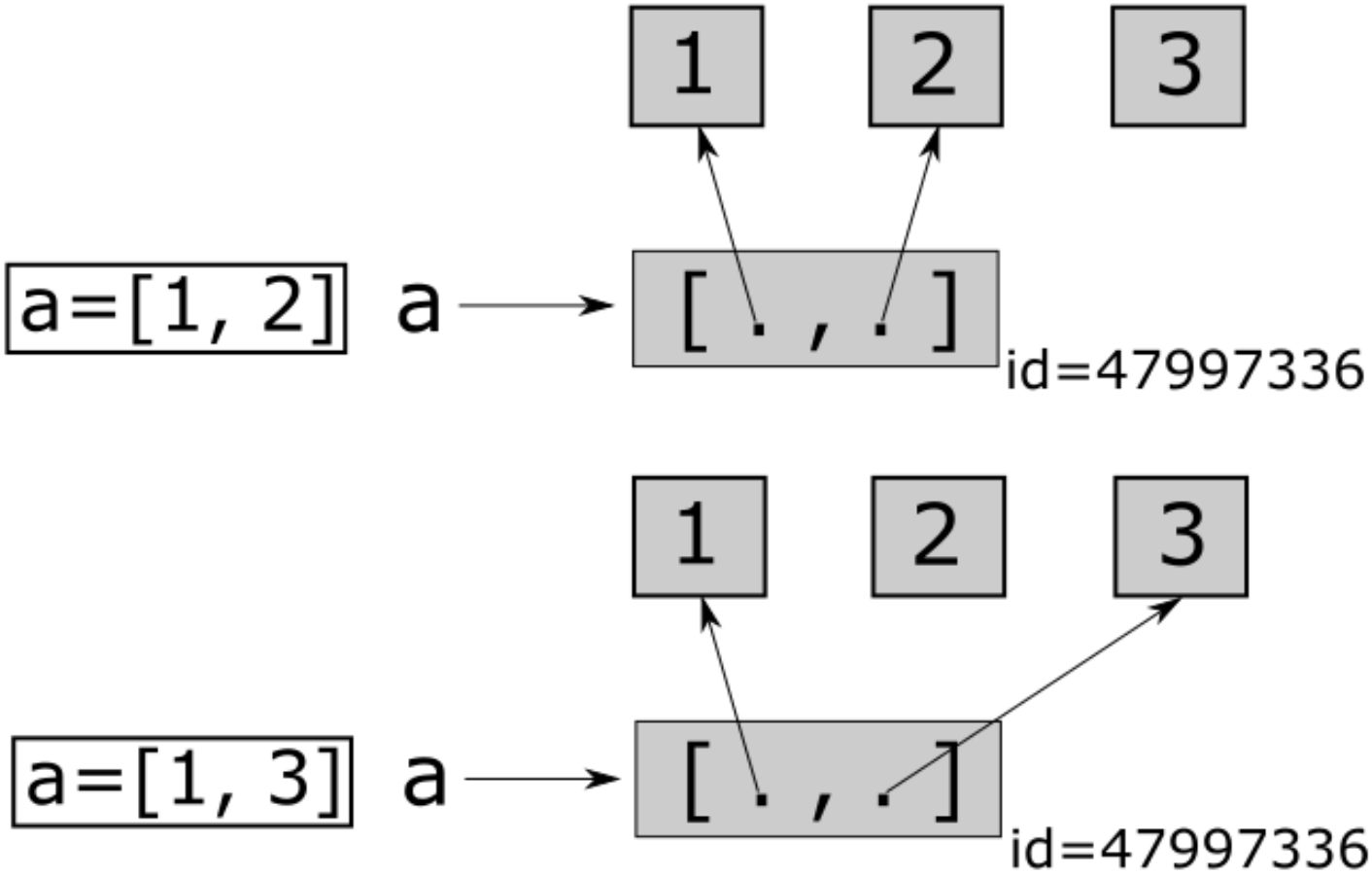
- Изменяемые типы (mutable)
  list, set, dict
  Значение объекта можно менять

```
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 = '{1}, {2}, {3}'.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'
```

Оператор %

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

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 соответствует байтовая строка `\x41`, U+20AC – 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

print('Done')
if guess == number: print('Good') # one-line block
```

- **while–else**
- **for–else**
- **break**
- **continue**

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) # 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
```

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() # x=50 x=2
print('x =', x) # x=2
```

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
```

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 is referenced before assignment
```

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

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

from math import sqrt # import function sqrt
sqrt(9)

dir(math) # list of all function in module
help(math.sqrt) # description of math.sqrt

# reload module
import imp
imp.reload(mymodule)
mymodule.say_hello()
```

__builtins__ – модуль, который Python загружает в память в момент начала работы

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

Пакет fincal содержит в себе три модуля: simper.py, compper.py и annuity.py

```
fincalc
|-- __init__.py
|-- simper.py
|-- compper.py
|-- annuity.py
```

```
# import module simper.py from package fincalc
import fincalc.simper
fv = fincalc.simper.fv(pv, i, n) # usage
import fincalc.simper as sp
fv = sp.fv(pv, i, n)
from fincalc import simper
fv = simper.fv(pv, i, n)
```

Файл __init__.py

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

Пример файла __init__.py:

```
__all__ = ["simper", "compper", "annuity"]
```

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]
```

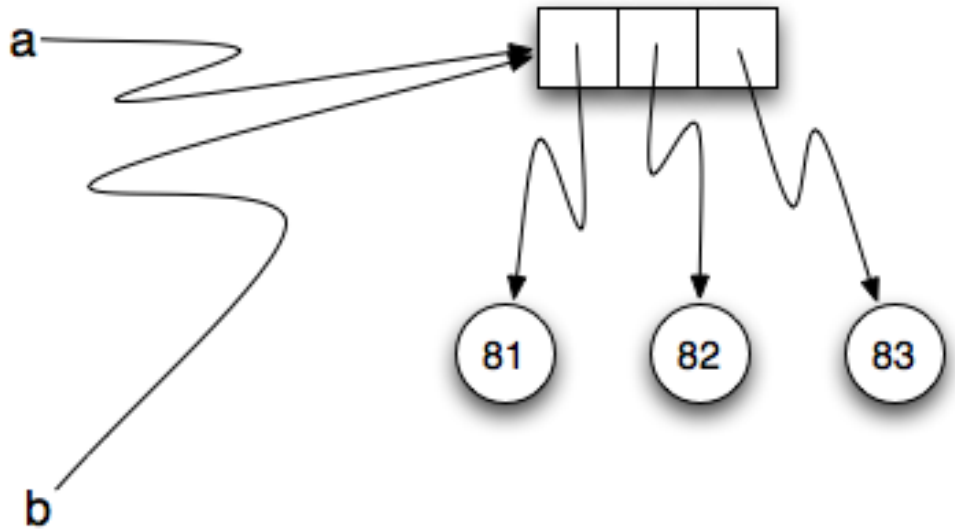
В Python две переменные называются псевдонимами, если они ссылаются на одинаковые адреса памяти

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–else** 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 генератора

range() – генератор последовательности целых чисел

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

Функции генератора 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
```

Функции генератора. Пример

```
type(my_range) # function
a = my_range()
type(a) # generator
next(a) # 0.0
next(a) # 0.1
...
next(a) # 0.9
next(a) # error
b = [x+2 for x in my_range(0,0.5)]
# [2.0, 2.1, 2.2, 2.3, 2.4]
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

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)
# 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}
```

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

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

    return my_wrapper
```

Декораторы. Примеры

```
def add2(a,b):
    return a+b

add2_decored = my_decorator(add2)
add2_decored(2,5)
# Function add2 start
# Result: 7

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

Декораторы. Примеры

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)
# Function add2 start
# Result: 7
```

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

Для функции может быть несколько декораторов

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

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

Передача параметров в функции

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

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')
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)
```

Функция map

Функция **map** применяет указанную функцию к каждому элементу последовательности

```
a = [4,16,9]
b = map(sqrt, a) # map
next(b); # 2
next(b); # 4
next(b); # 3

b = map(lambda x: 2*x, a) # map
next(b); # 8
next(b); # 32
next(b); # 18

b = map(lambda x,y: x+y, [1,2,3], [10,11,12])
a = list(b) # [11,13,15]
```

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')
```

Обработка исключений

```
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!')
```

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__()
```

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

Оператор with. Пример

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")
```

Работа с файлами

```
f = open('newfile.txt', 'w') # open for writing
text = f.read() # read all file
text = f.read(n) # read n chars
line = f.readline() # read line
lines = f.readlines() # read all lines
# read all lines
for line in f:
    print(line)

lines = [line for line in f] # list of lines
f.write('My text\n') # write to file
f.close() # close file
```

Объектно-ориентированное программирование в Python

Python – полностью объектно-ориентированный язык программирования

Класс – тип, описывающий устройство объектов

Объекты – экземпляры классов

```
class A:
    x = 15
    def print_x(self):
        print(self.x)

    def sum(self, a, b):
        return self.x+a+b

a = A() # create instance
a.print_x() # 15
b = a.sum(5,2) # 22
```

Аргумент self

Каждый метод объекта должен иметь обязательный первый аргумент **self**

```
class A:
    def hello():
        print('Hello, world!')

a = A()
a.hello() # error: self is absent
A.hello() # 'Hello, world!'
```

При вызове `a.hello()` происходит вызов `A.hello(a)`

При вызове `obj.func(x)` происходит вызов `MyClass.func(obj, x)`

Конструктор и деструктор класса

```
class A:
    def __init__(self, x):
        self.x = x
        print('Hello!')

    def __del__(self):
        print('Good bye!')

a = A() # error: constructor has argument x
a = A(15) # Hello!
a.x # 15
del a # Good bye!
```

При создании экземпляра класса `a = A(15)` происходит создание объекта `a` и вызов конструктора `A.__init__(a,15)`

Значения по умолчанию

Использование в методах (как и в любых функциях) значений по умолчанию для изменяемых типов может привести к ошибкам

```
class Bus:
    def __init__(self, passengers=[]):
        self.passengers = passengers

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)
```

Значения по умолчанию. Пример

```
bus1 = Bus(['Alice','Bill'])
bus1.passengers # ['Alice','Bill']
bus1.pick('Charlie')
bus1.drop('Alice')
bus1.passengers # ['Bill','Charlie']

bus2 = Bus()
bus2.pick('Carrie')
bus2.passengers # ['Carrie']

bus3 = Bus()
bus3.passengers # ['Carrie']
bus2.passengers is bus3.passengers # True
bus1.passengers # ['Bill','Charlie']
```

Значения по умолчанию

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

```
class Bus:
    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
         else:
            self.passengers = passengers
```

Значения по умолчанию создаются лишь однажды, когда функция определяется

Переменные класса и переменные объекта

- **Переменные класса** (Class attributes)
  Разделяемы – доступ к переменным класса могут получать все экземпляры класса
  Существует только одна для всех экземпляров класса
  Обращение через `ClassName.var_name` или `obj.var_name`
- **Переменные объекта** (Instance attributes)
  Принадлежат каждому отдельному экземпляру класса
  Обращение через `obj.var_name`

В Python все переменные классов и объектов – публичные (public), кроме переменных, чьё имя начинается с двойного подчёркивания `__` (например, `__x`)

Переменные класса и переменные объекта

Переменная объекта с тем же именем, что и переменная класса, сделает недоступной (“спрячет”) переменную класса

```
class A:
    x = 10


a = A()
b = A()
a.x # 10 (class attribute)
b.x # 10 (class attribute)
A.x # 10 (class attribute)

a.x = 4 # this creates new instance attribute x
a.x # 4 (a instance attribute)
b.x # 10 (class attribute)
A.x # 10 (class attribute)
```

Переменные класса и переменные объекта

Словарь всех переменных объекта хранится в системном атрибуте `__dict__`
Словарь переменных класса – в атрибуте `__class__.__dict__`

```
class A:
    x = 10


a = A()
b = A()
a.x = 4
a.__dict__ # {'x':4}
a.__class__.__dict__ # {'x':10}
b.__class__.__dict__ # {'x':10}
A.__dict__ # {'x':10}
```

Методы класса и методы объекта

- **Методы класса** (Class methods)
  Каждый метод класса должен иметь обязательный первый аргумент `cls`
  Вызов через `ClassName.method()` или `obj.method()`
  При вызове `obj.method()` происходит вызов `ClassName.method(ClassName)` (первым аргументом неявным образом передаётся класс)
- **Методы объекта** (Instance methods)
  Каждый метод объекта должен иметь обязательный первый аргумент **`self`**
  Вызов через `obj.method()`
  При вызове `obj.method()` происходит вызов `ClassName.method(obj)` (первым аргументом неявным образом передаётся сам объект)

Статические методы

Статические методы (static methods) не используют данные экземпляров класса

Вызов через `ClassName.method()` или `obj.method()`

Статический метод может рассматриваться как обычная функция, объявленная в пространстве имён класса

В Python все методы классов и объектов – публичные (public), кроме методов, чьё имя начинается с двойного подчёркивания `__` (например, `__method()`)

В Python все методы классов и объектов – виртуальные (virtual)

Объявление методов класса

```
class A:
    def func1(self, x):
        ...
    def func2(cls, x):
        ...
    def func3():
        ...
    def func4():
        ...
    func2 = classmethod(func2)
    func3 = staticmethod(func3)
```

Для func3 возможны вызовы A.func3() и A().func3()

Для func4 – только вызов A.func3()

Использование декораторов

```
class A:
    # instance method
    def func1(self, x):
        ...

    @classmethod
    def func2(cls, x):
        ...

    @staticmethod
    def func3():
        ...

    def func4():
        ...
```

Методы класса и методы объекта. Пример

```
class MyClass:
    @classmethod
    def func(cls, arg):
        print('Class %s: %d' % (cls.__name__, arg))

    # instance method
    def f(self):
        self.func(10)

MyClass.func(0) # 0
MyClass.f() # error: f is not class method
a = MyClass() # instance
a.func(0) # 0
a.f() # 10
```

Зачем нужны методы класса?

Методы класса могут использоваться для создания объектов класса

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @classmethod
    def fromList(cls, list):
        return cls(list[0],list[1])

p = Point(2,4) # x=2, y=4
p = Point.fromList([2,4]) # x=2, y=4
```

Свойства класса: геттеры и сеттеры

Обращение к атрибутам класса через геттеры и сеттеры

```
class A:
    def __init__(self, x):
        self.__x = x
    def get_x(self):
        return self.__x
    def set_x(self, x):
        self.__x = x

a = A(5)
a.get_x() # 5
b = A(10)
a.set_x(a.get_x() + b.get_x()) # ugly style
a.get_x() # 15
```

Свойства класса: декораторы

```
class A:
    def __init__(self,x):
        self.x = x
    @property
    def x(self):
        return self.__x
    @x.setter
    def x(self, x):
        self.__x = x


a = A(5)
a.x # 5
b = A(10)
a.x = a.x + b.x
a.x # 15
```

Особенности декораторов свойств

- Имя геттера и сеттера совпадает с именем свойства, геттер декорируется `@property`, сеттер – `@varname.setter`
- В классе хранится приватная переменная, соответствующая свойству (например, `__varname`)
- В теле геттера и сеттера обращение к свойству проводится через приватную переменную `self.__varname`
- В остальных методах класса, в том числе конструкторе, обращение к свойству – через объявленные геттер и сеттер `self.varname`
- В классе определяются две функции с одинаковым именем `varname` – это возможно благодаря использованию декораторов

Наследование

```
class Mydict(dict):
    def get(self, key, default=0):
        return dict.get(self, key, default)

a = dict(a=1, b=2) #  {'a':1, 'b':2}
b = Mydict(a=1, b=2) #  {'a':1, 'b':2}
print(a.get('v')) # None
print(b.get('v')) # 0
```

Для вызовов функций родительского класса может быть использована функция **super**()

```
class MyClass(BaseClass):
    def method(self, arg):
        super().method(arg)
```

Переопределение и перегрузка методов

Переопределение метода (overriding) – приём в ООП, позволяющий классам-потомкам реализовывать новое поведение метода, определённого в родительском классе

Перегрузка метода (overloading) – определение метода с тем же именем, но отличающегося в сигнатуре (т.е. принимающего или возвращающего значения различных типов, имеющего отличающееся число параметров и т.п.)

В Python отсутствует возможность явной перегрузки метода

Определение метода в классе-потомке с тем же именем, что и в родительском классе, его переопределяет

Переопределение. Пример

```
class A:
    def f(self, x):
        print(x)

class B(A):
    def f(self, x, y): # it's overriding
        print(x+y)

a = A()
a.f('Hello!') # 'Hello!'
b = B()
b.f('Hello!') # error: f takes 2 parameters
b.f('Hello,',' world!') # 'Hello, world!'
```

Перегрузка. Пример

```
class A:
    def f(self, x):
        print(x)

class B(A):
    def f(self, x, y=None): # it's overriding
        if y==None:
            super().f(x)
        else:
            print(x+y)

b = B()
b.f('Hello!') # call A.f('Hello!')
b.f('Hello,',' world!') # 'Hello, world!'
```

Наследование от встроенных классов

Большинство встроенных классов (built-in classes) реализованы на языке C. Их реализация не может вызывать переопределённые в потомках методы

```
class Mydict(dict):
    def __setitem__(self, key, value):
        super().__setitem__(key,[value]*2)

a = Mydict(key1=1) # {'key1':1}
a['key2'] = 2 # {'key1':1, 'key2':[2,2]}

# insert new key via update()
a.update(key3=3)
# {'key1':1, 'key2':[2,2], 'key3':3}
# update() use built-in implementation of __setitem__
```

Перегрузка операторов

При вызове оператора объекта происходит вызов соответствующей “магического” метода объекта (magic method)

Например:

```
a = [1,2,3]
b = [4,5]
c = a+b # [1,2,3,4,5]
# it is equivalent to
c = a.__add__(b) # [1,2,3,4,5]

a==b # False
# it is equivalent to
a.__eq__(b) # False
```

“Магические” методы могут быть переопределены

Переопределение операторов. Пример

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return '({}, {})'.format(self.x,self.y)

    def __eq__(self, other): # operator ==
        return (self.x==other.x)and(self.y==other.y)

    def __add__(self, other): # operator +
        return Point(self.x+other.x,self.y+other.y)
```

Переопределение операторов. Пример

```
p1 = Point(2,4)
print(p1) # (2,4)
p2 = Point(3,10)
print(p2) # (3,10)
p = p1+p2
print(p) # (5,14)
```

Типы операторов, которые могут быть переопределены:

- Унарные операторы (+, −, ~, **abs**(), ...)
- Бинарные операторы (+, −, ∗, /, //, ...)
- Операторы расширенного присваивания (+=, −=, ∗=, /=, ...)
- Операторы сравнения (<, <=, ==, =, >=, >!)

Абстрактные классы

Для работы с абстрактными классами (abstract classes) используется библиотека abc (Abstract Base Class)

```
from abc import ABC, abstractmethod
class MyAbstract(ABC):
    @abstractmethod
    def foo(self):
        pass

class MyClass(MyAbstract):
    def foo(self):
        print('Hello, world!')

a = MyAbstract() # error: class is abstract
a = MyClass() # ok
```