PYTHON

1. Характеристика языка

Python — мультипарадигмальный интерпретируемый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Создатель — Guido van Rossum. Первая версия — 1991 г.

Основные сферы применения Python:

- Веб-разработка: синхоронные фреймворки Django, Flask и асинхронные Tornado, Twisted, Aiohttp.
- Научные расчеты (как альтернатива Matlab/R): библиотеки Numpy, SciPy. Научный софт: Nmag и др.
- Машинное обучение и Data science — библиотеки Pandas, Scikit-Learn, PyBrain, Theano, Tensorflow и др.
- Тестирование ПО.
- Скрипты для автоматизации (замена bash).

Основные реализации (интерпретаторы) Python:

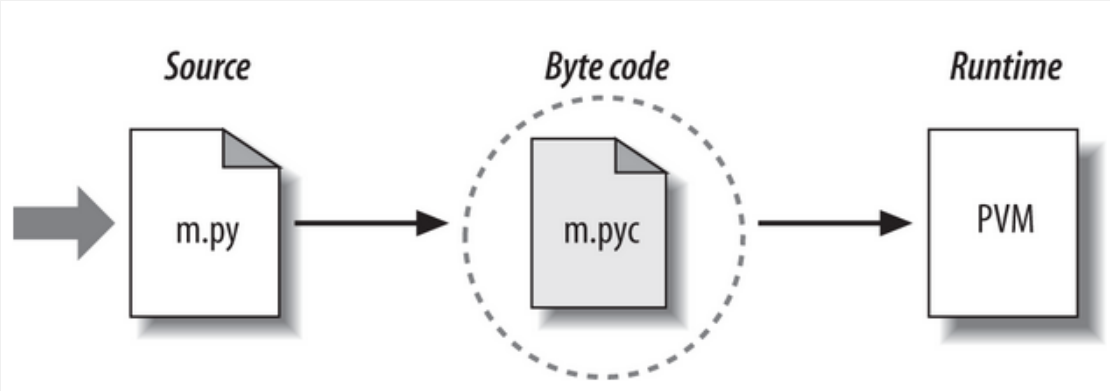
- **CPython** — эталонная реализация языка Python. CPython — интерпретатор байт-кода, написанный на C. Хорошо интегрируется с кодом, написанными на C.
- **Jython** = JAVA + Python. Позволяет запускать код Python на виртуальной машине JVM.
- **IronPython** — интерпретатор Python, написанный на C#, для платформы Microsoft .NET или Mono.
- **PyPy** — изначально был интерпретатором Python, написанным на Python. В текущих версиях используется JIT-компилятор, который превращает Python-код в машинный код во время выполнения программы. Производительность выше, чем у CPython. Имя пакета `pypy`.

Cython — отдельный язык программирования с Python-подобным синтаксисом, который транслируется в код C/C++ и компилируется.

Среды разработки:

- Linux: `sudo apt-get install python3 python3-numpy python3-pip` и т.д.
- Windows: интерпретатор python + IDE PyCharm + Anaconda.
- Android: `QPython3`.

Python (CPython) — интерпретируемый язык. Основа интерптератора — **PVM** (Python Virtual Machine).



Python 3.0 — версия Python, вышедшая в 2008 г., где устранены некоторые недостатки архитектуры. Неполная совместимость с Python 2.x.

Таблица 1. Python2 vs Python3

Python2	**Python3**
`print "Hello"`	`print("Hello")`
`xrange(10)`	`range(10)`
`input()`	`eval(input())`
`5/2 == 2`	`5/2 == 2.5`
`len("Питон") == 10`	`len("Питон") == 5`

```
1 from __future__ import division, print_function, absolute_import
```

1.1. **Пример программы на Python.**

```
#Пример программы
import sys
from math import sin, cos, sqrt
from numpy import *
print('Error', file=sys.stderr)
```

`help()` — справка;

`dir()` — возвращает список атрибутов и методов произвольного объекта;

`type()` — возвращает тип объекта;

2. Типы данных в Python

Все данные в Python являются **объектами**, в том числе встроенные типы, функции, экземпляры классов и сами классы, модули и файлы, и т.д.

В Python используется **динамическая типизация**, т.е. тип данных определяется автоматически при присвоении значения переменной.

Переменная создается в момент присваения ей значения.

Присвоение значения осуществляется с помощью знака (=).

`del` — оператор удаления переменной.

```
1 age = 23 # Присвоение значения 23 переменной age
2 print(age)
3 country = "Swiss"
4 print(country)
```

Таблица 2. Встроенные типы данных Python3

Типы данных	**Класс**	**Примеры литералов**
Булевский тип	`bool`	`True, False`
Числовые типы	`int, float, complex`	`-5; 1.0; 3.5-1j`
Строки	`str`	`"Hello"` либо `'Hello'`
Множества	`set`	`{1,2,3}`
Списки	`list`	`[], [True, 786, 3.14, 'text', 70.2]`
Кортежи	`tuple`	`(), (True, 786, 3.14, 'text', 70.2)`
Словари	`dict`	`{}, {1 : 'One', 2 : 'Two'}`
Файлы		`f = open(filename)`
«Нулевой тип»	`NoneType`	`None`
Прочие типы	`type`	`int, str, ...`

Преобразование типов данных:

```
1 int(x [,base]), float(x), complex(real [,imag]),
2 str(x), tuple(s), list(s), dict(d);
```

Документация: `https://docs.python.org/3/library/stdtypes.html`

2.1. **Булевский тип (bool).**

Значения: `True, False`.

Операции над типом bool: `or, and, not`.

В False преобразуются следующие объекты: None, False, 0, 0.0, 0j, Decimal(0), Fraction(0, 1); `'', (), [], {}`, set(), range(0).

Операции сравнения типов: `<, <=, >, >=, ==, !=, is, is not`.

2.2. **Числовые типы.**

Встроенные числые типы в Python3 — `int, float, complex`.
(в python2 есть также long)

Целый тип `int` имеет произвольный диапазон значений.

Основные операции над числами: `+ - * / **`.

Целая часть и остаток: `// %`.

В Python3 операция `/` всегда возвращает float, `//` — int или float.

Преобразование к другим система счисления: `int, bin, hex, oct; int(str,base)`.

Есть также `format()` (и `f'` в python-3.6):

```
>>> bin(14)
'0b1110'
>>> format(14, 'b')
'1110'
>>> f'{14:b}'  # python >=3.6
'1110'
```

Бесконечность:

```
inf = float("inf")
minus_inf = float("-inf")
```

Модули `numbers, fractions` — расширенные возможности для работы с числами.

Встроенные математические функции: `sqrt, abs, pow`. Больше в модуле `math`.

2.3. **Списки (класс list).**

```
a = [1, 3, 3, 2]
print(a[0]) # обращение по индексу
b = [x**2 for x in a]   # list comprehension (генератор списков)
print(a)   # [1, 3, 3, 2]
print(b)   # [1, 9, 9, 4]
del a[1]   # удаление элемента
```

`len()` — встроенная функция, возвращает количество элементов в последовательности (в списке и т.п.).

`in` — проверят, содержит ли последовательность данное значение (`x in ls`).

Некоторые методы класса list:

- Вставка и удаление элементов: `append()`, `insert()`, `remove()`, `pop()`.
- Сортировка элементов списка: `sort()` и `reverse()`.

(Например, `insert(index, object)` — для вставки нового элемента в середину или начало).

Копирование списков.

```
>>> L = [1, 2, 3]
>>> S = L
>>> L[0] = 9   # изменим первый элемент списка L
>>> L
[9, 2, 3]
>>> S            # в списке S первый элемент тоже изменился
[9, 2, 3]      # так как это один и тот же список на самом деле
```

Сделать реальную копию списка:

```
>>> L = [1, 2, 3]
>>> S = L.copy()
>>> L[0] = 9
>>> L
[9, 2, 3]
>>> S       # теперь все в порядка
[1, 2, 3]   # список S не затронут
```

Почему так?

— Инструкция присваивания (=) всегда создает ссылку на объект (но не создает копии объектов).

— Переменные создаются при первом присваивании.

— Перед использованием переменной ей должно быть присвоено значение.

Способы копирования списков:

```
a = [1, 2, 3]
b = a.copy()  # python >= 3.4
b = a[:]
import copy
b = copy.copy(a)
b = list(a)
```

Вложенные списки.

```
a = [1, 2, 3]
a[0] = a
```

Внутренняя реализация списков в CPython.

Питоновкий `list` — аналог `vector` в STL, реализован как одномерный массив указателей, а именно в виде структуры

```
typedef struct {
    PyObject_VAR_HEAD
    PyObject **ob_item; // массив указат. на элементы списка
    Py_ssize_t allocated; // количество выделенной памяти.
} PyListObject;
```

Сложность операций:

- добавление (append) элемента в конец списка — $O(1)$;
- выталкивание (pop) — $O(1)$;
- вставка (insert) — $O(n)$;
- удаление (remove) — $O(n)$;

Сортировка списков осуществляется алгоритмом Timsort:

Time: best — $O(n)$, average — $O(n \log n)$, worst — $O(n \log n)$;

Memory: $O(n)$.

(хороший по скорости, плохой по памяти)

```
list.sort(key=func, reverse=True|False)
```

Пример:

```
lst.sort(key=lambda obj: obj.value)
```

2.4. **Кортежи (Класс tuple).**

Кортеж (`tuple`) — это неизменяемая последовательность элементов.

```
a = (1, 3, 3, 2)
a = 1, 3, 3, 2
a = 1,          # то же самое, что и (1,) или tuple([1])
a = ()          # кортеж из 0 элементов
```

Применение:

```
a, b = b, a
```

Зачем нужны кортежи?

- обезопасить данные от случайного изменения;
- размер на 8(?) байт меньше, чем у списка;
- можно использовать в качестве ключа у словаря, т.к. он хешируем;
- производительность:

  В CPython кортежи хранятся в одном блоке памяти, поэтому создание нового кортежа в худшем случае приводит к одному вызову для выделения памяти. Списки распределяются в двух блоках: фиксированный со всей информацией об объекте Python и блоком с переменным размером для данных.

Внутренняя реализация кортежей в CPython:

```c
typedef struct {
    PyObject_VAR_HEAD
    PyObject *ob_item[1];
} PyTupleObject;

typedef struct {
    PyObject_VAR_HEAD
    /* Vector of pointers to list elements.  list[0] is ob_item[0], etc. */
    PyObject **ob_item;
    Py_ssize_t allocated;
} PyListObject;
```

```
sys.getsizeof((1,2,3,4,5))
```

2.5. **Множества.**

Множество (`set`) — неупорядоченная коллекция уникальных элементов:

```
a = {1, 3, 3, 2}
b = [x**2 for x in a]
print(a)   # {1, 2, 3}
print(b)   # [1, 4, 9]
>>> 3 in a
>>> 't' not in a
```

Операции: `in, not in`.

Проверка `in` для `set` быстрее, чем для `list`.

Операции над множествами: `& | - ^ <= >=`.

2.6. **Словари.**

Словарь (`dict`) = ассоциативный массив — неотсортированная колекция элементов, доступ к которым осуществляется по ключу:

```
>>> d = {'black' : 0, 'white' : 1}
>>> d['white']
1
>>> 'black' in d
True
```

```
d.keys()
d.values()
d.items()
```

2.7. **Строки.**

Строки (`str`) в python — последовательности unicode-символов.

Можно использовать апострофов либо кавычки. Оба варианта эквивалентны:

```
s = 'python'
s = "python"
```

Тройные кавычки используются для записи многострочных блоков текста:

```
text = '''многострочный
   блок текста'''
```

Строка — неизменяемый объект.

Сырая строка:

```
s = r'\t\t\t'
```

Форматирование строк.

1) В стиле «сишного printf»:

```
format % values
```

Например,

```
x = 7
str = 'x=%d' % x              # out: x=7
str = '%d+%d' % (x+1, x+2)    # out: 8+9
str = '%(a)d %(b)d %(a)d' % {'a':x, 'b':x+1}  # out: 7 8 7
```

2) Метод `format`:

```
print('i={0:03d}: t={1:.3f}, p={2:.2e}'.format(4,4,4))
t = 1000
print('i={i:03d}: t={t:.3f}, p={p:.2e}'.format(t=t,i=3,p=10))
```

3) Python >=3.6:

```
x = 1000
print(f'x = {x}')
```

Операции над символами.

- `ord`
- `chr`

Remark: Больше структур данных в модуле `collections`.

2.8. id, is, type.

`id` — идентификатор объекта:

```
>>> id(11)
139113728
>>> id(11)
139113728
>>> id(11111111111)
3072215544
>>> id(11111111111)
3072216264
```

Некоторые часто используемые объекты (например, True и False, все односимвольные строки и короткие числа) выделяются один раз интерпретатором, и каждая переменная, содержащая этот объект, ссылается на нее. Другие номера и более крупные строки выделяются по требованию.

`type` — определяет тип объекта:

```
>>> type(55)
<class 'int'>
>>> print(int)
<class 'int'>
```

`==` сравнивает объекты на равенство/неравенство.

`is` сравнивает идентификаторы — является ли это один и тот же объект.

```
>>> a = 123456
>>> b = 123456
>>> a == b
True
>>> a is b
False
```

```
>>> [1, 2] == [1, 2]
True
>>> [1, 2] is [1, 2]
False
```

Применение `is`:

```
>>> type(55) is int  # но можно type(55) == int
True
```

`is` рекомендуется используется для сравнения с `None` (поскольку `==` может быть переопределен через `__eq__`).

`isinstance(x, A)` — определяет принадлежит ли объект `x` данному классу `A` либо производному от него.

3. Управляющие операторы (циклы и ветвления)

- Ветвление: `if`/`elif`/`else`.
- Циклы: `while`, `for`.

Remark: в Python блоки кода выделяются с помощью горизонтальных отступов.

```
1 for x in [1, 2]:
2     print(x)
3     print(x^2)
4 print(x^3)
```

В этом примере 2-ая и 3-я строки кода включены в тело цикла `for`, а инструкция 4-ой строки расположена вне цикла.

3.1. Ветвление — if.

```
if <условие1>:
   <блок1>
elif <условие2>:
   <блок2>
else:
   <блок3>
```

Например,

```
if 1 > 2:
   print('условие выполнено')
```

Нет оператора множественного выбора (switch – case) как в C++.

Используется `if`/`elif`/.../`elif`/`else`.

Тернарный оператор:

```
res = 'even' if x % 2 == 0 else 'odd'
```

3.2. **Циклы.** Операторы циклов:

- `for .. in ..` — обход всех элементов коллекции;
- `while` — цикл с предусловием[1].

Например,

```
for x in {3, 2, 1}:
   print(x)
```

Стандартная форма цикла `while` следующая:

```
while <условие>:
   <блок>
```

Вспомогательные инструкции:

- `break`
- `continue`
- `pass` — это пустая инструкция
- `pass` — `else` располагается после цикла и всегда выполняется в том случае, если цикл завершается обычном способом (без прерывания с помощью `break`)

[1] В Python нет цикла с постусловием do...while

Класс range.

`range(start, stop, step)` — последовательность чисел с заданным шагом.

Если `step>0`, то образует возврастающую последовательность чисел с условием

start <= start + i * step < stop

где i — целое неотрицательное число. Например,

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

Применение

— в циклах:

```
for x in range(0, 10, 2):
    print(x)
```

— для генерации списков (и других последовательностей):

```
>>> [x**2 for x in range(0, 10, 2)]
[0, 4, 16, 36, 64]
```

Объявление функции

```
1 def func(x):
2     y = x*x
3     return y
4 z = func(9)
5 print(y)
```

4.1. **Локальные и глобальные переменные.**

- Если переменной присваивается значение внутри функции, то она считается *локальной* переменной функции.
- Если присваивание происходит за пределами функции, то она является *нелокальной* для данной функции.
- Если присваивание происходит за пределами всех инструкций `def`, то она является *глобальной*.

Разрешение имен по правилу LEGB:

- Local
- Enclosing (Nonlocal)
- Global
- Built-in

`global` — позволяет изменять переменные, находящиеся на верхнем уровне модуля.

`nonlocal` — позволяет изменять переменные в объемлющих функциях.

```
#Использование global
x = 1;
def func():
   global x
   x = 2
func()
print('x =', x) # Result: x = 2
```

(избегать использования глобальных переменных)

Что будет напечатано?

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

x = 5
f()
```

```
def f():
   print(x)
   x += 2

x = 5
f()
```

Выполнение присваивания переменной значения в области видимости делает ее *локальной* в этой области.

A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block.

`locals(), globals()`

4.2. **Параметры по умолчанию.**

```
def f(x, y=3):
   return x + y
```

```
def foo(a=[]):
    a.append(1)
    print(a)
foo()
foo()
foo()
```

Значения по-умолчанию вычисляются один раз в момент объявления функции и ссылка на него сохраняется в атрибуте `__defaults__` (для Python3).

4.3. **Именованные аргументы.**

```
def f(a, b, c):
   print(sum([a,2*b,3*c]))

f(a=1, c=2, b=3)
```

4.4. **Произвольное число аргументов.**

```
def mysum(*args):
   print(sum(args))

mysum(3,2,1,3,4,5)
```

Для именованных аргументов:

```
def f(*pargs, **kargs):
   print(sum(pargs))
   print(kargs)
f(3,2,1,a=3,b=4,c=5)
```

4.5. **lambda-функции.**

Лямбда-функция (или анонимная функция) — это функция, при определении которой не нужно указывать ее имя.

```
1 func = lambda x, y: x**2 + y**2
2 func(2, 3)
```

```
1 (lambda x: x+2)(5)
```

Список литературы

[1] Лутц, М. Изучаем Python / М. Лутц. — 4-ое изд. — Пер. с англ. — СПб.: Символ-Плюс, 2011. — 1280 с.

[2] Сузи Р. А. Python : [полное руководство]. — Санкт-Петербург [и др.] : БХВ-Петербург, 2002. — 748 с. [чз]