Интерпретируемые языки программирования

Интерпретируемые языки не предназначены для процесса компиляции. Их выполнение происходит одновременно с чтением и разбором файла исходных кодов.

Для выполнения программы на интерпретируемом языке нужна специальная программа: интерпретатор.

Один из первых интерпретируемых языков — сценарий командной оболочки. Его задача — автоматизировать работу администратора путем выполнения заранее записанных команд, решающих типовую задачу.

Исходный код на интерпретируемом языке программирования носит название **сценарий** или **скрипт**.

Интерпретируемые языки программирования

Очевидный минус интерпретируемых языков — более низкая производительность.

Плюсы:

- Независимость языка от платформы. Все вопросы совместимости вынесены на уровень интерпретатора.
- Простота кода, избавление его от системной прослойки.
- Возможность встройки интерпретатора в другое ПО и автоматизация его функционирования с помощью сценариев.
- Кросс-платформенность на уровне исходного кода.

Python

Сравнительно молодой язык, первая версия интерпретатора: 1994 год. Язык учел большинство ошибок и наработок более старых языков.

Открытый исходный код, можно свободно использовать язык и интерпретатор для любых разработок.

Огромная стандартная библиотека, решающая большинство рутинных задач.

Огромная кодовая база, большое количество проектов на данном языке.

Сайт: http://python.org (интерпретатор, документация).

Средства разработки для Python

- Минимальный набор: любой текстовый редактор. Запуск в консоли.
- Стандартная поставка: среда разработки **IDLE** — продвинутый текстовый редактор с подсветкой синтаксиса, возможность запуска сценария во встроенном терминале.
- **Eclipse + PyDev**
- **PyCharm**
- **Spyder**. Используется в составе проекта Anaconda, заточенного под научные вычисления.

Установка пакетов

- Вручную. Файлы пакета копируются в папку lib/site-packages в папке интерпретатора. Подходит для небольших любительских библиотек.
- При помощи установочного файла.
- Используя консольную утилиту **pip** из папки scripts.

```
c:\python\scripts> pip help
c:\python\scripts> pip list
c:\python\scripts> pip search matplotlib
c:\python\scripts> pip install matplotlib
c:\python\scripts> pip install pyqt5 pyqt5-tools
```

Синтаксис языка

- Исходный код оформляется в сценарий с расширением «**.py**». Он же является запускаемым файлом.
- Одна строка — это всегда одна инструкция.
- Допускается разделять одну инструкцию на несколько строк, используя символ \ для экранирования переноса. Также допускаются выражения в скобках на несколько строк.
- Разделитель инструкций отсутствует (символ **;** игнорируется)
- Отсутствует функция **main**: инструкции начинают выполняться с начала файла.

```
print("Hello world")
print("Hello, this is a long ",
      "very long message to display!")
```

Переменные

Отсутствует явное объявление переменной.

Переменная появляется в момент первого присвоения новому идентификатору.

Допускается присвоение любых данных любому идентификатору. Переменная может поменять свой тип в момент присвоения для сохранения данных другого типа.

Операторы сравнения (==) и присвоения (=) такие же как в C. Однако, допускается множественное присвоение нескольким переменным сразу.

```
v = 10
a, b, c = 1, 2, 4
```

Переменные

Каждая переменная в конкретный момент времени принадлежит к определенному типу:

- **bool**: принимает значения **True** и **False**.
- **int**: принимает целочисленные значения.
- **float**: принимает вещественные значения
- **str**: хранит строку — последовательность символов.
- **NoneType**: может принимать только значение **None**.

Вызов **type()** позволяет определить тип переменной.

```
b = True
a = 4
f = 2.4e+9
n = None
```

Литералы

Каждому типу данных соответствует свой литерал для записи констант.

```
  # bool:
True, False
  # int:
234
0o24
0xAF14
0b11010010
  # float:
1.25
6.02e23
```

```
  # str:
"Hello world"
'Hello, world'
"I can 'quote' anything!"
'By "mixing" quotes!'

'''This is a string
that takes
several lines of text'''
```

Преобразования типов

Типы данных можно конструировать на основе других типов. Тип **str** можно сконструировать из чего угодно:

```
v = 12
s = str(v)
```

int и **float** не преобразовываются друг в друга автоматически, однако могут участвовать вместе в арифметических операциях:

```
i = 12
f = 4.22
```

```
af = f * i
ai = int(f * i)
```

При конструировании **bool** получается **False**, если аргумент — False, None, 0, либо пустое множество (например пустая строка). Иначе получается **True**.

```
bool(1)     # True
bool(None) # False
```

```
bool("")    # False
bool(" ")   # True
```

Унарные операторы

- **-** (минус) — обращение знака числа
- **~** (тильда) — обращение всех битов целого числа
- **not** — логическое НЕ

Арифметические операторы

- **=** (равно) — присвоение. Всегда происходит справа налево.
- **+**, **-**, ***** — сложение, вычитание и умножение чисел.
- ***/*** (косая черта) — деление чисел. Результатом всегда будет вещественное число, независимо от операндов.
- ***//*** (двойная косая черта) — деление чисел нацело. Результатом всегда будет число без дробной части.
- % — остаток от деления целого числа на целое число.
- ****** (двойная звездочка) — возведение в степень.

Побитовые операторы

- **&** (амперсанд) — Поразрядное И.

```
0xFFFF & 0x011A      // 0x011A
0x00FF & 0x011A      // 0x001A
```

- **|** (вертикальная черта) — Поразрядное ИЛИ.

```
0x0000 | 0x011A      // 0x011A
0x00FF | 0x011A      // 0x01FF
```

- **^** (крышка) — Поразрядное Исключающее ИЛИ (XOR).

```
0x0001 ^ 0x0101      // 0x0100
0x0100 ^ 0x0101      // 0x0001
```

Побитовые операторы

- **<<** — Сдвиг влево

```
0x0001 << 4      // 0x0010
0x00FF << 8      // 0xFF00
```

- **>>** — Сдвиг вправо

```
0x0100 >> 6      // 0x0004
0x00FF >> 4      // 0x000F
```

Логические операторы

- **and** — логическое И

```
False and False   # False
False and True    # False
True  and True    # True
```

- **or** — логическое ИЛИ

```
False or False   # False
False or True    # True
True  or True    # True
```

Если любой из операндов не относится к типу **bool**, он будет автоматически преобразован в него с использованием **bool()**.

Операторы сравнения

- **==** (двойное равенство) — строгое равенство
- **!=** — не равно
- **<, >** — строгое неравенство
- **<=, >=** — нестрогое неравенство

Для объединения неравенств можно использовать логические операторы, комбинируя отдельные сравнения целиком. Допускаются двойные неравенства.

```
a = 44
10 < a <= 44                    # True
a < 100 or a > 200              # True
b = 12
(b > 0 and b < 20) or a <100    # True
```

Приоритет операторов

Каждая операция обладает своим встроенным приоритетом. Например, умножение всегда делается раньше, чем сложение.

Приоритет операций можно изменить, используя скобки, так как они обладают наивысшим приоритетом.

```
10 + 2 * 4          // 18
(10 + 2) * 4        // 48
```

Допускается переопределение операторов для создаваемых классов. При этом приоритет переопределенных операторов не меняется.

Составные типы данных

В Python отсутствуют классические массивы, равно как и указатели и возможность работать с динамической памятью напрямую. Вместо этого стандартная библиотека Python содержит ряд классов для хранения произвольного количества элементов различных типов. Данные классы можно использовать вместо массивов.

- **Тьюпл** или **кортеж** (**tuple**) — это неизменяемое хранилище данных. Отдельные переменные внутри кортежа различаются по индексам.
- **Список** (**list**) — это хранилище данных, аналогичное кортежу, но способное менять свое содержимое.
- **Словарь** (**dict**) — это хранилище данных, в котором доступ к элементам осуществляется по ключу, а не по индексу.

Кортеж

Создается при помощи круглых скобок, в которых через запятую записаны значения входящих в кортеж переменных.

```
T = (1, 2, "24", "hello!")
```

Доступ к элементам кортежа осуществляется по индексу при помощи квадратных скобок. Нумерация может идти как с начала, так и с конца. Для кортежа из N элементов справедливы индексы [0 .. N-1] и [-N .. -1]

```
print(T[0])    # 1
print(T[-1])   # hello!
```

Элементами кортежа могут быть любые типы данных, в том числе другие кортежи:

```
T = ((1,1), (1,2), (2,2))
print(T[0])      # (1,1)
print(T[1][1])   # 2
```

Кортеж

Длину кортежа можно получить при помощи встроенной функции **len**:

```
T = (1, 2, "24", "hello!")
print(len(T))      # 4
```

Для выбора нескольких идущих подряд элементов используется двоеточие внутри квадратных скобок. При этом первый индекс всегда включается в диапазон, а второй не включается.

Если не указывать первый индекс, то выбор идет с начала. Если не указывать второй — то до конца.

```
print(T[:2])       # (1, 2)
print(T[1:3])      # (2, "24")
print(T[2:])       # ("24", "hello!")
print(T[1:-1])     # (2, "24")
```

Кортеж

Одиночная переменная, заключенная в скобки, не превращается в кортеж:

```
T = (4.25)
print(type(T))      # <class 'float'>
```

Кортеж, состоящий из одной переменной, создается так:

```
T = (1,)
```

Это необходимо, чтобы избежать случайного создания кортежей в математических выражениях.

Пустой кортеж создается так:

```
empty_T = ()
print(len(empty_T))      # 0
```

Список

Создается при помощи квадратных скобок, в которых через запятую записаны значения входящих в список элементов.

```
L = [2, 32, 42, "nop", "Hi there!"]
```

Индексы элементов списка точно такие же, как у кортежа, и доступ к элементам осуществляется точно так же через квадратные скобки.

```
print(L[0])       # 2
print(L[-1])      # Hi there!
```

Элементами списка могут быть любые типы данных, в том числе другие списки и кортежи:

```
L = [(1,1), (1,2), (2,2), ["01", "11", "21", "31"]]
print(L[0])       # (1,1)
print(L[-1][2])   # 21
```

Список

Длину списка можно получить при помощи встроенной функции **len**:

```
L = [2, 32, 42, "nop", "Hi there!"]
print(len(L))      # 5
```

Для выбора нескольких идущих подряд элементов используется двоеточие внутри квадратных скобок. При этом первый индекс всегда включается в диапазон, а второй не включается.

Если не указывать первый индекс, то выбор идёт с начала. Если не указывать второй — то до конца.

```
print(L[:2])       # [2, 32]
print(L[1:3])      # [32, 42]
print(L[2:])       # [42, "nop", "Hi there!"]
print(L[1:-1])     # [32, 42, "nop"]
```

Список

Модификация элементов списка также идет через квадратные скобки:

```
L = [2, 32, 42, "nop", "Hi there!"]
L[1] = -4.41
print(L)    # [2, -4.41, 42, "nop", "Hi there!"]
```

Удаление элемента осуществляет оператор **del**:

```
del L[-1]
print(L)    # [2, -4.41, 42, "nop"]
```

Можно убрать элемент из списка и переложить его в другую переменную используя метод **pop**. Аргумент метода — индекс удаляемого элемента.

```
a = L.pop(3)
print(L)    # [2, -4.41, 42]
print(a)    # nop
```

Список

Добавление элементов в конец списка производится при помощи метода **append**:

```
L = [2, -4.41, 42]
L.append("End")
print(L)                # [2, -4.41, 42, "End"]
```

Вставка метода в произвольную позицию в списке осуществляется методом **insert**. Аргументы метода — позиция перед которой нужно вставить значение и само значение.

```
L = [2, -4.41, 42]
L.insert(0, 10)
print(L)                # [10, 2, -4.41, 42, "End"]
```

Словарь

Создается при помощи фигурных скобок. Каждый элемент словаря — это пара «Ключ»-«Значение», разделенные двоеточием.

```
D = {"one": 1, "two": 2}
```

Доступ к элементам словаря осуществляется путем указания значения ключа в квадратных скобках.

```
print(D["one"])    # 1
```

Запись значения в словарь также происходит при помощи оператора квадратных скобок. При этом, если на момент записи значение с таким ключом не существовало в словаре, оно будет создано, а если существовало — то будет перезаписано.

```
D["two"] = 22
D["three"] = 3
print(D)           # {"one": 1, "two": 22, "three": 3}
```

Общее для составных типов

Количество элементов можно получить при помощи встроенного оператора **len**:

```
D = {"one": 1, "two": 2}
len(D)      # 2
```

```
L = [1, 2, 4, 8, 16, 32]
len(L)      # 6
```

Проверить, входит ли заданное значение в хранилище, можно при помощи оператора **in**. Для списков и кортежей следует указывать значение, а для словарей — значение ключа.

```
print("one" in D)  # True
print(2 in D)      # False
```

```
print(2 in L)      # True
print("8" in L)    # False
```

Строка также является составным типом, хранящим символы по индексам, причем строки в Python нельзя изменять.

Близкая аналогия: строка — это кортеж из символов.

Ленивое копирование

Присвоение между составными типами (в том числе между строками!) просто присваивает тем же данным новый идентификатор.

```
L1 = [1, 2, 3, 4]
L2 = L1
L1[0] = 0
print(L2)     # [0, 2, 3, 4]
```

При выборке диапазона данных при помощи двоеточия внутри квадратных скобок происходит копирование указанной части данных в новое хранилище. Можно использовать это для полного копирования данных:

```
L1 = [1, 2, 3, 4]
L2 = L1[:]
L1[0] = 0
print(L2)     # [1, 2, 3, 4]
```

Оператор условного перехода

Оператор **if** используется для определения, будет ли выполнен следующий за ним операторный блок.

Операторный блок определяется по **уровню отступа**, то есть по количеству пробельных символов в начале строки, а также требует символ **:** (двоеточие) на предыдущей строке. Операторный блок заканчивается там, где встречается строка с тем же уровнем отступа, что и инструкция с двоеточием ранее:

```
a = 10
if a > 0:
    print(a)
```

Для записи пустого операторного блока используется пустой оператор: **pass**

```
if a == 0:
    pass
```

Оператор условного перехода

Вместе с оператором **if** может использоваться оператор **else**. Он следует сразу за операторным блоком **if** на том же уровне отступа, что и **if**, и за ним следует альтернативный операторный блок, который будет выполнен если условие после **if** не выполнено.

После выполнения **if-else**, сценарий продолжит выполняться в обычном режиме.

В общем случае **else** может отсутствовать.

```
a = 10
b = 0
if a > 0:
    a = 0
else:
    b = a
print("done")
```

Циклы

Оператор **while** определяет условие, выполнение которого заставляет программу войти в следующий за ним операторный блок.

После окончания операторного блока, происходит переход обратно к оператору **while**, и вычисления повторяются.

Одно выполнение операторного блока цикла носит название **итерация**.

Операторный блок определяется так же, как и у оператора **if**: при помощи отступа.

```
a = 0
while a < 10:
    a += 1
b = a
```

Циклы

Внутри цикла доступны операторы **break** и **continue**.

break завершает цикл и переводит выполнение программы к первой строке после операторного блока.

continue переводит выполнение сразу к началу следующей итерации.

```
a = 1
while True      # Сюда ведет оператор continue
    if a > 10:
        continue
    if a == 0:
        break
    a = a + 3
# Сюда ведет оператор break
```

Циклы

Оператор **for** используется совместно с оператором **in** для записи цикла, в котором переменная последовательно принимает значения из переданного набора.

В качестве набора значений может выступать любая подходящая переменная: кортеж, список или строка.

```
T = (1, 2, 3, 4, 5)
sum = 0
for a in T:
    sum += a
print(sum)          # 15
```

```
for a in "Hello":
    print(a)
# H
# e
# l
# l
# o
```

Циклы

Оператор **for** часто используется вместе с функцией **range**, которая создает арифметическую прогрессию, значения из которой пробегает используемая в **for** переменная.

- **range(start, end, step)** — создает арифметическую прогрессию начиная со start и до end, не включая end, с шагом step. Аргументы start, end и step — целые числа.
- **range(start, end)** — создает арифметическую прогрессию в диапазоне [start, end) с шагом 1.
- **range(end)** — создает арифметическую прогрессию в диапазоне [0, end) с шагом 1

Использование **range** позволяет обходить списки и кортежи в ситуациях, когда нужно обработать часть кортежа, либо требуется относительное перемещение между элементами.

Циклы

```
# Создание видеоимпульса единичной амплитуды длиной в
# 60 отсчетов на общей длительности сигнала в 150
# отсчетов
L = []
for a in range(150):
    L.append(0)

for a in range(20, 80):
    L[a] = 1
```

```
# Вывод букв, идущих после символа "пробел"
# во введенном тексте
s = input()
for a in range(len(s)):
    if s[a] == ' ' and a < len(s):
        print('The word starts with ' + s[a+1])
```

Функции

Функция определяется при помощи ключевого слова **def** (от слова definition). Затем после пробела записывается имя функции.

В скобках после имени функции приводится список аргументов. Типы аргументов не конкретизируются. Тип возвращаемого значения нигде не указывается.

Для мгновенного возврата из функции используется **return**, после которого через пробел указывается возвращаемое значение. Если функция завершится без вызова **return**, или **return** будет без возвращаемого значения, результат работы функции — **None**.

Функция является полноценным объектом. Ее можно передать в качестве аргумента в другую функцию или запомнить в переменной.

Функции

```
# Функция рассчитывает сумму натуральных чисел
# от 1 до x
def N_sum(x):
    if x <= 0:
        return
    s = 0
    for a in range(1, x+1):
        s += a
    return s

print(N_sum(4))          # 10
print(N_sum(-2))         # None
```

Функции

Функция может создавать новые переменные, чья область видимости ограничена ее операторным блоком.

Введение новой переменной внутри функции с тем же именем, что и переменная снаружи функции, скроет внешнюю переменную, и она станет недоступна внутри функции.

Функция может обращаться к глобальным переменным. При попытке изменить глобальную переменную, в функции будет создана копия глобальной переменной с тем же именем, и все обращения будут адресованы копии. Глобальная переменная останется нетронутой.

При использовании ключевого слова **global**, глобальная переменная используется как есть, создания копии не происходит. То есть функция может менять значение указанной глобальной переменной.

Функции

```
N = 0

def func():
    N = 123
    print("N=" + str(N))

print(N)        # 0
func()          # N=123
print(N)        # 0
func()          # N=123
```

Функции

```
N = 0

def func():
    global N
    N = 123
    print("N=" + str(N))

print(N)        # 0
func()          # N=123
print(N)        # 123
func()          # N=123
```

Модули

Модули — это файлы python, содержащие программный код. Код определяет внутри себя объекты: функции, переменные.

Ключевое слово **import** позволяет импортировать указанный модуль и подключить доступные в нем объекты в текущий сценарий.

В качестве имени модуля может выступать как модуль, предоставленный библиотекой интерпретатора, так и сценарий, расположенный в той же папке.

В момент обработки команды **import** интерпретатор открывает соответствующий файл и выполняет все записанные в нем команды.

Модули

Использование **import** без дополнительной обвязки импортирует все объекты из модуля. Они становятся доступны через имя модуля и точку:

```
import math
print(math.exp(2.25))  # 9.487735836358526
```

Использование **from** и имен объектов позволяет избежать дополнительного использования имени модуля:

```
from math import exp
print(exp(2.25))       # 9.487735836358526
```

Можно импортировать все объекты для их использования без имени модуля, используя *****, а также дать объектам другое имя:

```
from math import *
from random import random as rnd
print(exp(rnd()))        # Случайное число от 1 до e
```

Стандартная библиотека Python

print — вывод в терминал строковых данных. При необходимости, аргумент **print** преобразуется к строке.

```
n = 33 * 28
print(n)
```

input — ввод строки с клавиатуры в терминале. Аргумент — строка-приглашение перед вводом данных.

```
s = input("Input something")
print("You entered: " + s)
```

open — открытие файла для чтения или записи. Создает файловый дескриптор, через который можно читать из файла или писать в файл.

```
F = open("text.txt", "w")
F.write("Hello!")
F.close()
```

Стандартная библиотека Python

- **cmath** — работа с комплексными числами
- **math** — математические функции и константы
- **random** — генерация псевдослучайных чисел
- **sys** — функции для взаимодействия с операционной системой
- **socket** — работа с сетевыми соединениями
- **time** — системное время
- **zipfile** — работа с zip-архивами

https://docs.python.org/3/library/

Полезные сторонние библиотеки

Полезные пакеты, доступные для установки с помощью pip:

- **matplotlib** — построение графиков
- **PyQt5** — библиотека Qt 5 для использования вместе с Python. Рекомендуется также установить пакет **pyqt5-tools** для установки Qt Designer.
- **prettytable** — вывод таблиц в терминал при помощи псевдографики.
- **pillow** — работа с растровыми изображениями.