Подробное введение в Python

Язык программирования Python 3 — это мощный инструмент для создания программ самого разнообразного назначения, доступный даже для новичков. С его помощью можно решать задачи различных типов.

1. Python 3: преимущества языка

1. Python - интерпретируемый язык программирования. С одной стороны, это позволяет значительно упростить отладку программ, с другой - обуславливает сравнительно низкую скорость выполнения.
2. Динамическая типизация. В python не надо заранее объявлять тип переменной, что очень удобно при разработке.
3. Хорошая поддержка модульности. Вы можете легко написать свой модуль и использовать его в других программах.
4. Встроенная поддержка Unicode в строках. В Python необязательно писать всё на английском языке, в программах вполне может использоваться ваш родной язык.
5. Поддержка объектно-ориентированного программирования. При этом его реализация в python является одной из самых понятных.
6. Автоматическая сборка мусора, отсутствие утечек памяти.
7. Интеграция с C/C++, если возможностей python недостаточно.
8. Понятный и лаконичный синтаксис, способствующий ясному отображению кода. Удобная система функций позволяет при грамотном подходе создавать код, в котором будет легко разобраться другому человеку в случае необходимости. Также вы сможете научиться читать программы и модули, написанные другими людьми.
9. Огромное количество модулей, как входящих в стандартную поставку Python 3, так и сторонних. В некоторых случаях для написания программы достаточно лишь найти подходящие модули и правильно их скомбинировать. Таким образом, вы можете думать о составлении программы на более высоком уровне, работая с уже готовыми элементами, выполняющими различные действия.
10. Кроссплатформенность. Программа, написанная на Python, будет функционировать совершенно одинаково вне зависимости от того, в какой операционной системе она запущена. Отличия возникают лишь в редких случаях, и их легко заранее предусмотреть благодаря наличию подробной документации.

2. Синтаксис

- Конец строки является концом инструкции (точка с запятой не требуется).
- Вложенные инструкции объединяются в блоки по величине отступов. Отступ может быть любым, главное, чтобы в пределах одного вложенного блока отступ был одинаков. И про читаемость кода не забывайте. Отступ в 1 пробел, к примеру, не лучшее решение. Используйте 4 пробела (или знак табуляции, на худой конец).
- Вложенные инструкции в Python записываются в соответствии с одним и тем же шаблоном, когда основная инструкция завершается двоеточием, вслед за которым располагается вложенный блок кода, обычно с отступом под строкой основной инструкции.

```
Основная инструкция:
    Вложенный блок инструкций
```

3. Типы данных в Python

3.1 None (null)

Ключевое слово null обычно используется во многих языках программирования, таких как Java, C++, C# и JavaScript. Это значение, которое присваивается переменной.

Концепция ключевого слова null в том, что она дает переменной нейтральное или "нулевое" поведение.

Эквивалент null в Python: None

Он был разработан таким образом, по двум причинам:

Многие утверждают, что слово null несколько эзотерично. Это не наиболее дружелюбное слово для новичков. Кроме того, None относится именно к требуемой функциональности - это ничего, и не имеет поведения.

Присвоить переменной значение None очень просто:

```
my_none_variable = None
```

Существует много случаев, когда следует использовать None.

Часто вы хотите выполнить действие, которое может работать либо завершиться неудачно. Используя None, вы можете проверить успех действия.

Вот пример:

```
if database_connection is None:
    print('The database could not connect')
else:
    print('The database could connect')
```

Python является объектно-ориентированным, и поэтому None - тоже объект, и имеет свой тип.

```
>>> type(None)
<class 'NoneType'>
```

Проверка на None

Есть (формально) два способа проверить, на равенство None.

Один из способов - с помощью ключевого слова ***is***.

Второй - с помощью == (но никогда не пользуйтесь вторым способом, и я попробую объяснить, почему).

Посмотрим на примеры:

```
null_variable = None
not_null_variable = 'Hello There!'

# The is keyword
if null_variable is None:
    print('null_variable is None')
else:
    print('null_variable is not None')

if not_null_variable is None:
    print('not_null_variable is None')
else:
    print('not_null_variable is not None')


# The == operator
if null_variable == None:
    print('null_variable is None')
else:
    print('null_variable is not None')

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

3.2 Числа: целые, вещественные, комплексные

Числа в Python 3: целые, вещественные, комплексные.

Работа с числами и операции над ними:

Целые числа (int)

Числа в Python 3 ничем не отличаются от обычных чисел. Они поддерживают набор самых обычных математических операций:

x + y	Сложение
x - y	Вычитание
x * y	Умножение
x / y	Деление
x // y	Получение целой части от деления
x % y	Остаток от деления
-x	Смена знака числа
abs(x)	Модуль числа
divmod(x, y)	Пара (x // y, x % y)
x ** y	Возведение в степень
pow(x, y[, z])	x^y по модулю (если модуль задан)

Также нужно отметить, что целые числа в python 3, в отличие от многих других языков, поддерживают длинную арифметику (однако, это требует больше памяти).

```
>>> 255 + 34
289
>>> 5 * 2
10
>>> 20 / 3
6.666666666666667
>>> 20 // 3
6
>>> 20 % 3
2
>>> 3 ** 4
81
>>> pow(3, 4)
81
>>> pow(3, 4, 27)
0
>>> 3 ** 150
369988485035126972924700782451696644186473100389722973815184405301748249
```

Битовые операции

Над целыми числами также можно производить битовые операции

x | y	Побитовое *или*
x ^ y	Побитовое *исключающее или*
x & y	Побитовое *и*
x << n	Битовый сдвиг влево
x >> y	Битовый сдвиг вправо
~x	Инверсия битов

Дополнительные методы

int.bit_length() - количество бит, необходимых для представления числа в двоичном виде, без учёта знака и лидирующих нулей.

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

int.to_bytes(length, byteorder, *, signed=False) - возвращает строку байтов, представляющих это число.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() // 8) + 1, byteorder='little')
b'\xe8\x03'
```

classmethod **int.from_bytes(bytes, byteorder, *, signed=False)** - возвращает число из данной строки байтов.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
```

```
16711680
```

Системы счисления

Те, у кого в школе была информатика, знают, что числа могут быть представлены не только в десятичной системе счисления. К примеру, в компьютере используется двоичный код, и, к примеру, число 19 в двоичной системе счисления будет выглядеть как 10011. Также иногда нужно переводить числа из одной системы счисления в другую. Python для этого предоставляет несколько функций:

int([object], [основание системы счисления]) - преобразование к целому числу в десятичной системе счисления. По умолчанию система счисления десятичная, но можно задать любое основание от 2 до 36 включительно.

bin(x) - преобразование целого числа в двоичную строку.

hex(x) - преобразование целого числа в шестнадцатеричную строку.

oct(x) - преобразование целого числа в восьмеричную строку.

Примеры:

```
>>> a = int('19') # Переводим строку в число
>>> b = int('19.5')  # Строка не является целым числом
Traceback (most recent call last):
  File "", line 1, in
ValueError: invalid literal for int() with base 10: '19.5'
>>> c = int(19.5)  # Применённая к числу с плавающей точкой,
отсекает дробную часть
>>> print(a, c)
19 19
>>> bin(19)
'0b10011'
>>> oct(19)
'0o23'
>>> hex(19)
'0x13'
>>> 0b10011  # Так тоже можно записывать числовые константы
19
>>> int('10011', 2)
19
>>> int('0b10011', 2)
19
```

Вещественные числа (float)

Вещественные числа поддерживают те же операции, что и целые. Однако (из-за представления чисел в компьютере) вещественные числа неточны, и это может привести к ошибкам:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
0.9999999999999999
```

Для высокой точности используют другие объекты (например Decimal и Fraction)).

Также вещественные числа не поддерживают длинную арифметику:

```
>>> a = 3 ** 1000
>>> a + 0.1
Traceback (most recent call last):
  File "", line 1, in
OverflowError: int too large to convert to float
```

Простенькие примеры работы с числами:

```
>>> c = 150
>>> d = 12.9
>>> c + d
162.9
>>> p = abs(d - c)   # Модуль числа
>>> print(p)
137.1
>>> round(p)   # Округление
137
```

Дополнительные методы

float.as_integer_ratio() - пара целых чисел, чьё отношение равно этому числу.
float.is_integer() - является ли значение целым числом.
float.hex() - переводит float в hex (шестнадцатеричную систему счисления).
classmethod **float.fromhex**(s) - float из шестнадцатеричной строки.

```
>>> (10.5).hex()
'0x1.5000000000000p+3'
>>> float.fromhex('0x1.5000000000000p+3')
10.5
```

Помимо стандартных выражений для работы с числами (а в Python их не так уж и много), в составе Python есть несколько полезных модулей.

Модуль math предоставляет более сложные математические функции.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(85)
9.219544457292887
```

Модуль random реализует генератор случайных чисел и функции случайного выбора. (подробнее в random.pdf в доп. материалах)

```
>>> import random
```

```
>>> random.random()
0.15651968855132303
```

Комплексные числа (complex)

В Python встроены также и комплексные числа:

```
>>> x = complex(1, 2)
>>> print(x)
(1+2j)
>>> y = complex(3, 4)
>>> print(y)
(3+4j)
>>> z = x + y
>>> print(x)
(1+2j)
>>> print(z)
(4+6j)
>>> z = x * y
>>> print(z)
(-5+10j)
>>> z = x / y
>>> print(z)
(0.44+0.08j)
>>> print(x.conjugate())  # Сопряжённое число
(1-2j)
>>> print(x.imag)  # Мнимая часть
2.0
>>> print(x.real)  # Действительная часть
1.0
>>> print(x > y)  # Комплексные числа нельзя сравнить
Traceback (most recent call last):
  File "", line 1, in
TypeError: unorderable types: complex() > complex()
>>> print(x == y)  # Но можно проверить на равенство
False
>>> abs(3 + 4j)  # Модуль комплексного числа
5.0
>>> pow(3 + 4j, 2)  # Возведение в степень
(-7+24j)
```

Для работы с комплексными числами используется также модуль **cmath**.

3.3 Работа со строками в Python: литералы

Строки в Python - упорядоченные последовательности символов, используемые для хранения и представления текстовой информации, поэтому с помощью строк можно работать со всем, что может быть представлено в текстовой форме.

Литералы строк

Работа со строками в Python очень удобна. Существует несколько литералов строк, которые мы сейчас и рассмотрим.

- **Строки в апострофах и в кавычках**

```
S = 'spam"s'
S = "spam's"
```

Строки в апострофах и в кавычках - одно и то же. Причина наличия двух вариантов в том, чтобы позволить вставлять в литералы строк символы кавычек или апострофов, не используя экранирование.

- **Экранированные последовательности - служебные символы**

Экранированные последовательности позволяют вставить символы, которые сложно ввести с клавиатуры.

Экранированная последовательность	Назначение
\n	Перевод строки
\a	Звонок
\b	Забой
\f	Перевод страницы
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\N{id}	Идентификатор ID базы данных Юникода
\uhhhh	16-битовый символ Юникода в 16-ричном представлении
\Uhhhh…	32-битовый символ Юникода в 32-ричном представлении
\xhh	16-ричное значение символа
\ooo	8-ричное значение символа
\0	Символ Null (не является признаком конца строки)

- **"Сырые" строки - подавляют экранирование**

Если перед открывающей кавычкой стоит символ 'r' (в любом регистре), то механизм экранирования отключается.

```
S = r'C:\newt.txt'
```

Но, несмотря на назначение, "сырая" строка не может заканчиваться символом обратного слэша. Пути решения:

```
S = r'\n\n\\'[:-1]
S = r'\n\n' + '\\'
S = '\\n\\n'
```

- **Строки в тройных апострофах или кавычках**

Главное достоинство строк в тройных кавычках в том, что их можно использовать для записи многострочных блоков текста. Внутри такой строки возможно присутствие кавычек и апострофов, главное, чтобы не было трех кавычек подряд.

```
>>> c = '''это очень большая
... строка, многострочный
... блок текста'''
>>> c
'это очень большая\nстрока, многострочный\nблок текста'
>>> print(c)
это очень большая
строка, многострочный
блок текста
```

Функции и методы строк

Базовые операции

- Конкатенация (сложение)

```
>>> S1 = 'spam'
>>> S2 = 'eggs'
>>> print(S1 + S2)
'spameggs'
```

- Дублирование строки

```
>>> print('spam' * 3)
spamspamspam
```

- Длина строки (функция len)

```
>>> len('spam')
4
```

- Доступ по индексу

```
>>> S = 'spam'
>>> S[0]
's'
>>> S[2]
'a'
>>> S[-2]
'a'
```

Как видно из примера, в Python возможен и доступ по отрицательному индексу, при этом отсчет идет от конца строки.

- Извлечение среза
  Оператор извлечения среза: [X:Y]. X – начало среза, а Y – окончание; символ с номером Y в срез не входит. По умолчанию первый индекс равен 0, а второй - длине строки.

```
>>> s = 'spameggs'
>>> s[3:5]
'me'
>>> s[2:-2]
'ameg'
>>> s[:6]
'spameg'
>>> s[1:]
'pameggs'
>>> s[:]
'spameggs'
```

Кроме того, можно задать шаг, с которым нужно извлекать срез.

```
>>> s[::-1]
'sggemaps'
>>> s[3:5:-1]
''
>>> s[2::2]
'aeg'
```

Другие функции и методы строк

При вызове методов необходимо помнить, что строки в Python относятся к категории неизменяемых последовательностей, то есть все функции и методы могут лишь создавать новую строку.

```
>>> s = 'spam'
>>> s[1] = 'b'
Traceback (most recent call last):
  File "", line 1, in
    s[1] = 'b'
TypeError: 'str' object does not support item assignment
>>> s = s[0] + 'b' + s[2:]
>>> s
'sbam'
```

Поэтому все строковые методы возвращают новую строку, которую потом следует присвоить переменной.

Таблица "Функции и методы строк"

Функция или метод	Назначение
S = 'str'; S = "str"; S = '''str'''; S = """str"""	Литералы строк

Функция или метод	Назначение
S = "s\np\ta\nbbb"	Экранированные последовательности
S = r"C:\temp\new"	Неформатированные строки (подавляют экранирование)
S = b"byte"	Строка байтов
S1 + S2	Конкатенация (сложение строк)
S1 * 3	Повторение строки
S[i]	Обращение по индексу
S[i:j:step]	Извлечение среза
len(S)	Длина строки
S.find(str, [start],[end])	Поиск подстроки в строке. Возвращает номер первого вхождения или -1
S.rfind(str, [start],[end])	Поиск подстроки в строке. Возвращает номер последнего вхождения или -1
S.index(str, [start],[end])	Поиск подстроки в строке. Возвращает номер первого вхождения или вызывает ValueError
S.rindex(str, [start],[end])	Поиск подстроки в строке. Возвращает номер последнего вхождения или вызывает ValueError
S.replace(шаблон, замена[, maxcount])	Замена шаблона на замену. maxcount ограничивает количество замен
S.split(символ)	Разбиение строки по разделителю
S.isdigit()	Состоит ли строка из цифр
S.isalpha()	Состоит ли строка из букв
S.isalnum()	Состоит ли строка из цифр или букв
S.islower()	Состоит ли строка из символов в нижнем регистре
S.isupper()	Состоит ли строка из символов в верхнем регистре
S.isspace()	Состоит ли строка из неотображаемых символов (пробел, символ перевода страницы ('\f'), "новая строка" ('\n'), "перевод каретки" ('\r'), "горизонтальная табуляция" ('\t') и "вертикальная табуляция" ('\v'))

Функция или метод	Назначение
S.istitle()	Начинаются ли слова в строке с заглавной буквы
S.upper()	Преобразование строки к верхнему регистру
S.lower()	Преобразование строки к нижнему регистру
S.startswith(str)	Начинается ли строка S с шаблона str
S.endswith(str)	Заканчивается ли строка S шаблоном str
S.join(список)	Сборка строки из списка с разделителем S
ord(символ)	Символ в его код ASCII
chr(число)	Код ASCII в символ
S.capitalize()	Переводит первый символ строки в верхний регистр, а все остальные в нижний
S.center(width, [fill])	Возвращает отцентрованную строку, по краям которой стоит символ fill (пробел по умолчанию)
S.count(str, [start],[end])	Возвращает количество непересекающихся вхождений подстроки в диапазоне [начало, конец] (0 и длина строки по умолчанию)
S.expandtabs([tabsize])	Возвращает копию строки, в которой все символы табуляции заменяются одним или несколькими пробелами, в зависимости от текущего столбца. Если TabSize не указан, размер табуляции полагается равным 8 пробелам
S.lstrip([chars])	Удаление пробельных символов в начале строки
S.rstrip([chars])	Удаление пробельных символов в конце строки
S.strip([chars])	Удаление пробельных символов в начале и в конце строки
S.partition(шаблон)	Возвращает кортеж, содержащий часть перед первым шаблоном, сам шаблон, и часть после шаблона. Если шаблон не найден, возвращается кортеж, содержащий саму строку, а затем две пустых строки
S.rpartition(sep)	Возвращает кортеж, содержащий часть перед последним шаблоном, сам шаблон, и часть после шаблона. Если шаблон не найден, возвращается кортеж, содержащий две пустых строки, а затем саму строку

Функция или метод	Назначение
S.swapcase()	Переводит символы нижнего регистра в верхний, а верхнего – в нижний
S.title()	Первую букву каждого слова переводит в верхний регистр, а все остальные в нижний
S.zfill(width)	Делает длину строки не меньшей width, по необходимости заполняя первые символы нулями
S.ljust(width, fillchar=" ")	Делает длину строки не меньшей width, по необходимости заполняя последние символы символом fillchar
S.rjust(width, fillchar=" ")	Делает длину строки не меньшей width, по необходимости заполняя первые символы символом fillchar
S.format(*args, **kwargs)	Форматирование строки