

An abstract graphic in the top-left corner consisting of a cluster of overlapping squares and circles in various shades of blue and white, creating a sense of depth and movement.

Python



Сферы применения Python

В порядке убывания охвата области и популярности языка в ней:

1. WEB (Django, Flask, aiohttp)
2. Data mining/нейросети (SciPy, NumPy)
3. Тестирование (PyTest)
4. Автоматизация (скрипты)
5. Системные утилиты (sys)
6. Desktop-приложения (PyQT)
7. Мобильные приложения (Kivy)



История языка

Язык программирования Python начал свою историю ещё в 1980-х годах, когда идеей о его создании загорелся Гвидо ван Россум - нидерландский программист. В декабре 1989 года он приступил к написанию языка Python в центре математики и информатики в Нидерландах. К 1991 была готова 1 версия интерпретатора. За основу был взят язык abc.

HOW TO RETURN words document:

PUT {} **IN** collection

FOR line **IN** document:

FOR word **IN** split line:

IF word not.in collection:

INSERT word **IN** collection

RETURN collection



Особенности языка

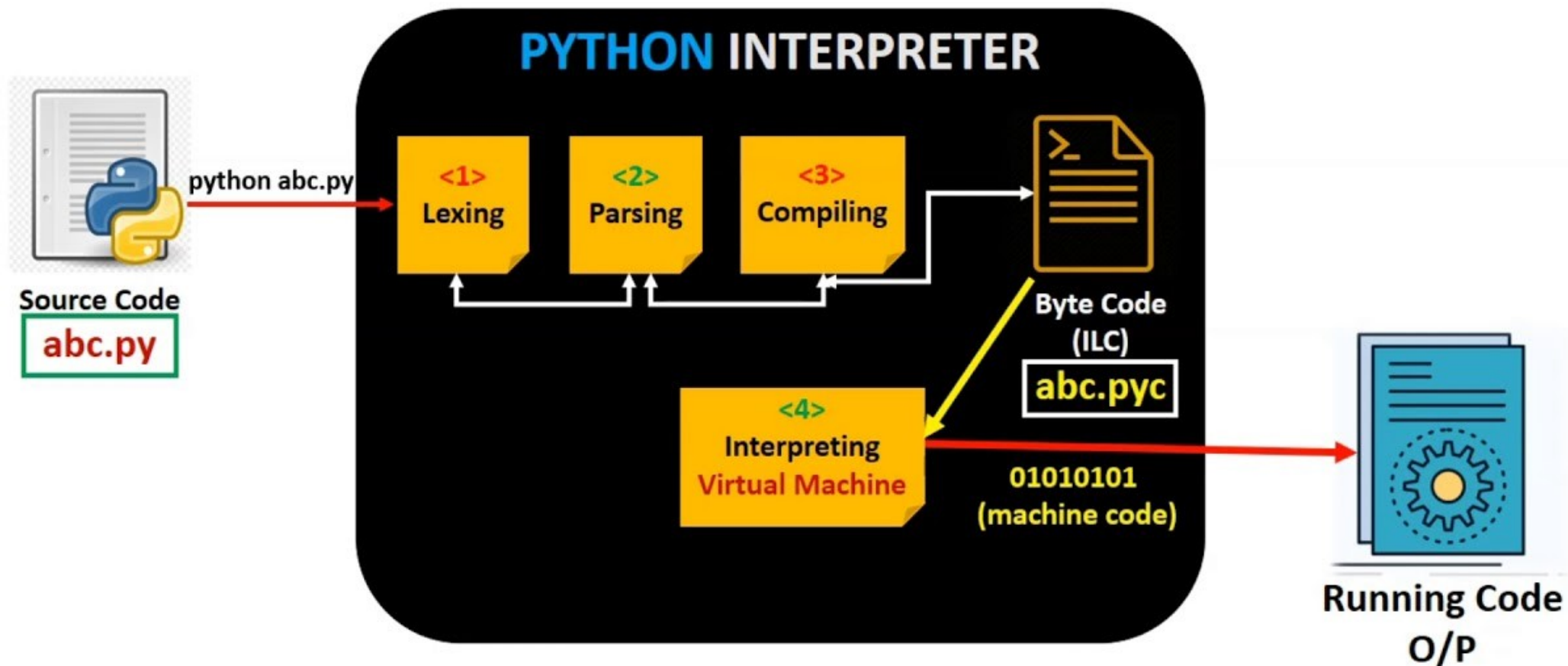
- Интерпретируемый язык высокого уровня
- Динамическая типизация
- Автоматический сборщик мусора
- Поддержка различных парадигм программирования
включая функциональный и объектно-ориентированный
подход



Как работает Python ?

1. Программа читается парсером и происходит анализ лексики. Где parser - это анализатор синтаксиса. В итоге получается набор лексем для дальнейшей обработки.
2. Затем парсером из инструкций происходит генерация структуры и формирования дерева синтаксического разбора- AST (Abstract Syntax Tree).
3. После этого компилятор преобразует AST в байт-код и отдает его на выполнение интерпретатору.

Simulating Python Interpreter

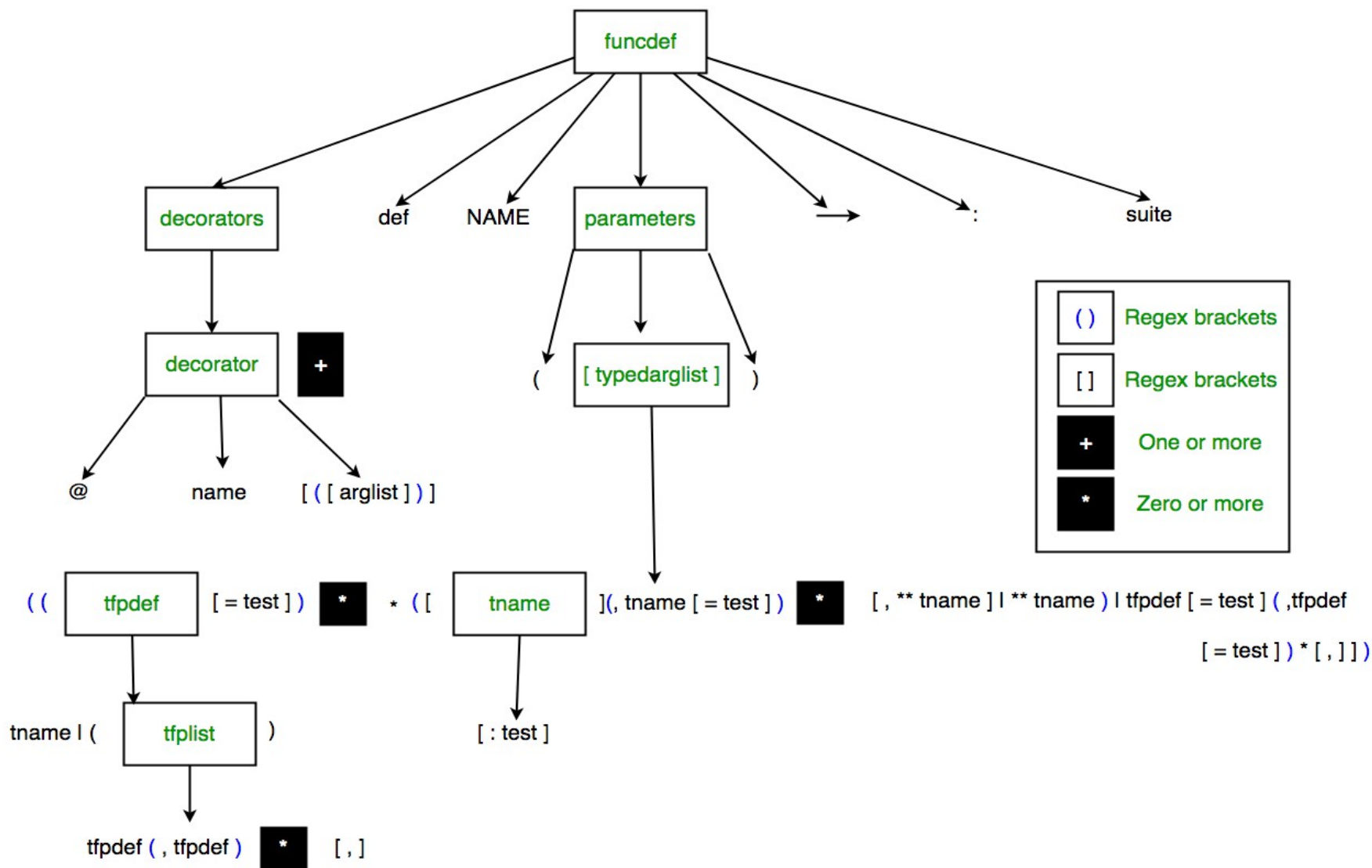




Рассмотрим интерпритацию данного кода.

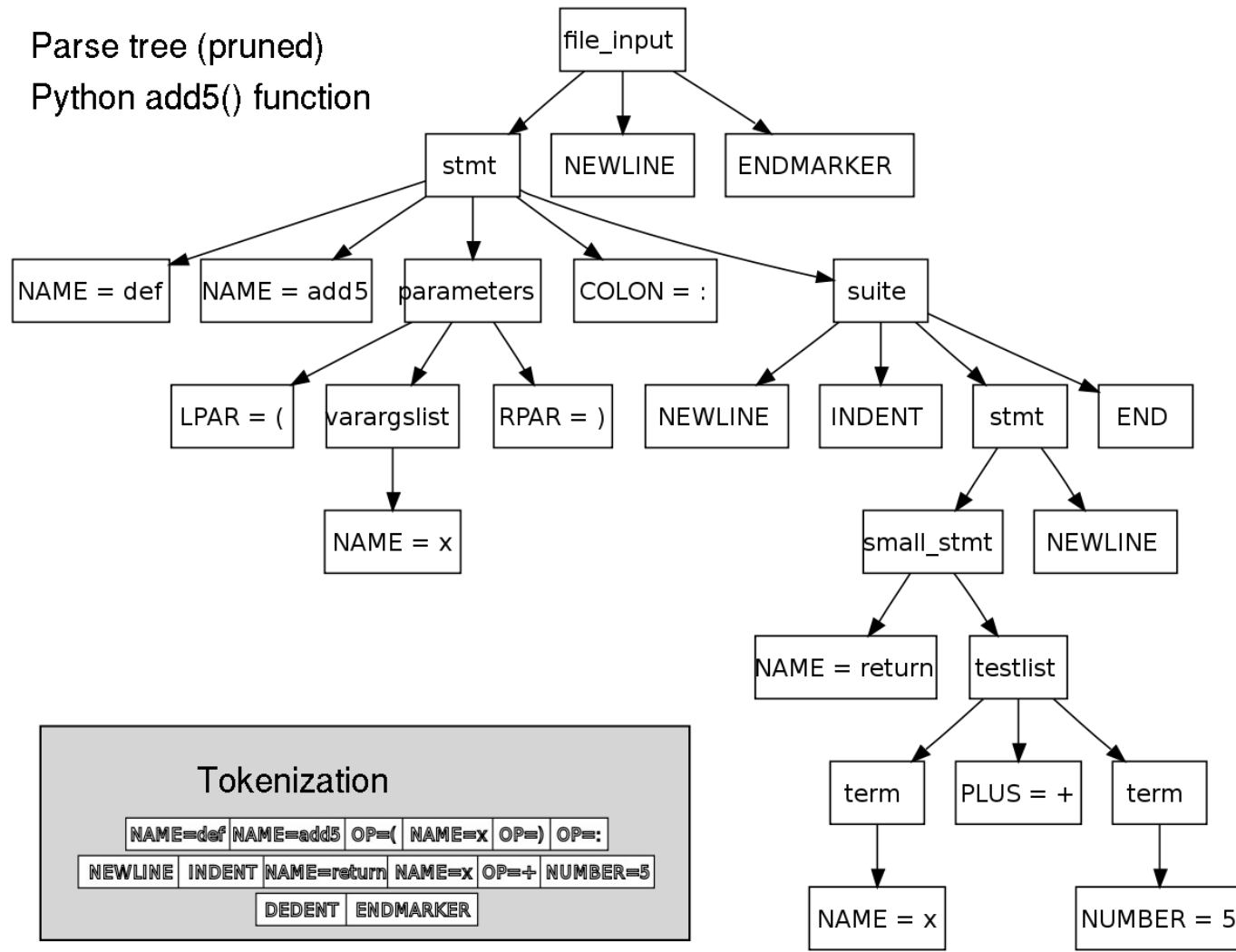
```
def summ5 (x) :  
    return x + 5
```

Грамматика функции




Дерево синтаксического разбора

Parse tree (pruned)
Python add5() function



Внутренние структуры хранения AST дерева

```
▼ Module: {} 2 keys
  ▼ body: [] 1 item
    ▼ 0: {} 1 key
      ▼ FunctionDef: {} 6 keys
        ► args: {} 1 key
        ► body: [] 1 item
        decorator_list: [] 0 items
        name: "summ5"
        returns: null
        type_comment: null
        type_ignores: [] 0 items
```



На стадии компиляции наш код превращается в байт код. В нашем случае бай код представлен мнемоническими именами. Затем он выполняется на виртуальной машине.

2	0 LOAD_FAST	0 (x)
	2 LOAD_CONST	1 (5)
	4 BINARY_ADD	
	6 RETURN_VALUE	



Для чего нам нужно знать про синтаксический разбор?

Если мы допускаем ошибки в грамматике кода то получаем синтаксическую ошибку.

SyntaxError: invalid syntax



Основы синтаксиса Python

Программа - это заданная последовательность инструкций. Инструкции выполняются сверху вниз.

* Конец строки является концом инструкции (**точка с запятой** не требуется).


* Вложенные инструкции объединяются в блоки по величине отступов. Отступ может быть любым. Отступ одинаков в пределах вложенного блока. В Python принят отступ в 4 пробела.



Формальные языки

Любой формальный язык, в том числе и Python, имеет три самые важные составляющие:

- * Типы и операторы над типами
- * Данные
- * Конструкции

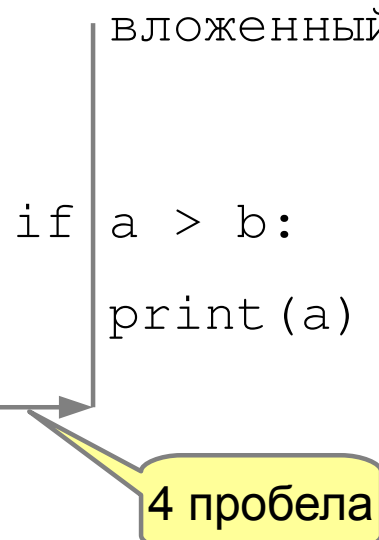


Некоторые операторы языка (if, for, try и т.д.) требуют вложенные инструкции. Они в Python записываются в соответствии с одним и тем же шаблоном. Когда основная инструкция завершается двоеточием, за ней идет вложенный блок кода с отступом.

основная инструкция:

вложенный блок

```
if a > b:  
    print(a)
```



4 пробела




Несколько случайных случаев

Иногда возможно записать несколько инструкций в одной строке, разделяя их точкой с запятой:


```
>>> a = 1; b = 2; print(a, b)
```

Но не делайте это слишком часто! Помните об удобочитаемости. А лучше вообще так не делайте.



Допустимо записывать одну инструкцию в нескольких строках.
Достаточно ее заключить в пару круглых, квадратных или фигурных скобок:

```
if (a == 1 and b == 2 and  
    c == 3 and d == 4): # Не забываем про двоеточие  
    print('spam' * 3)
```



Тело составной инструкции может располагаться в той же строке, что и тело основной, если тело составной инструкции не содержит составных инструкций. Пример:

```
>>> if x > y: print(x)
```



Начнем изучения языка, рассмотрим:

- Типы данных
- Операторы
- Конструкции ветвления
- Циклы
- Функции



Примитивные типы

1. Целые числа (int)
2. Числа с плавающей запятой (float)
3. Комплексные числа (complex)
4. Строки (str)
5. Массивы байт (bytearray)
6. Логический (bool)
7. NoneType



Системы счисления

- Десятичная

>>> 7 → int

>>> 3.14 → float

- Двоичная

>>> 0b0010 → int

- Восьмеричная

>>> 0o07 → int

- Шестнадцатиричная

>>> 0x0F → int



Операторы в Python для работы с числами

Операторы в Python для работы с числами:

" + "	(сложение)
" - "	(вычитание)
" * "	(умножение)
" / "	(деление)
" // "	(целочисленное деление)
" % "	(деление с остатком)
" ** "	(возведение в степень)

$\sqrt{25}$ эквивалентно `num ** (0.5)` -?

`abs(x)` - модуль числа

`divmod(x, y)` - пара (`x // y`, `x % y`)

`pow(x, y[, z])` x^y по модулю (если модуль задан)



- [illegible]

- >> 9 999 999

- ```
>> 2e400 → inf
```

- ```
>> -2e400    → -inf
```



Юмор из Monty Python

```
>> 3.14 * 10          →      31.40000000000000002
```

```
>> -4 // 3            →      -2
```

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

```
0.9999999999999999
```




Пример использования Decimal

```
>>> 0.1 + 0.1 + 0.1 - 0.3  
5.551115123125783e-17
```

Для высокой точности следует использовать другие объекты (например decimal и fraction)

```
>>> from decimal import Decimal  
>>> q=w=e=r=t=y=u=i=o=p=Decimal('0.1')  
>>> q+w+e+r+t+y+u+i+o+p  
Decimal('1.0')
```



Комплексные числа

- Комплексное число — это любое число в форме $a + bj$, где a и b — действительные числа, а $j*j = -1$.
- Каждое комплексное число $(a + bj)$ имеет действительную часть (a) и мнимую часть (b) .

$$>>n = 4 + 3j \quad \rightarrow \quad (4+3j)$$

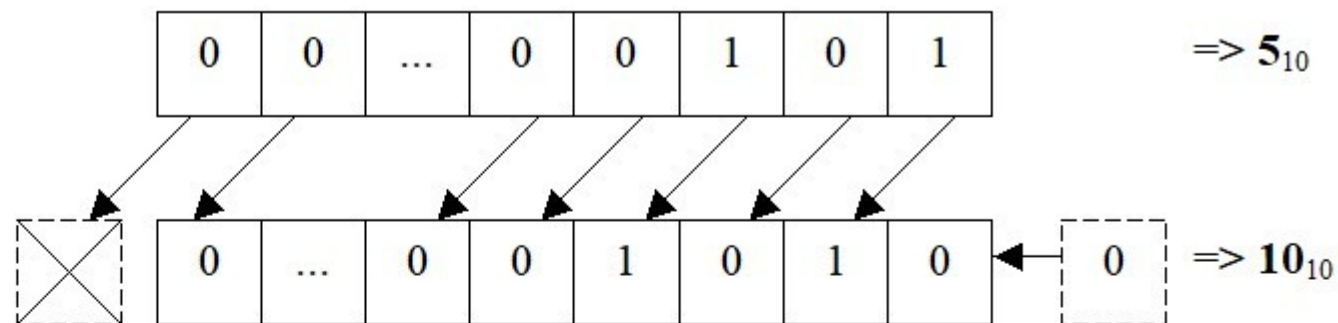


Битовые операторы

- ~ битовый оператор НЕТ (инверсия, наивысший приоритет);
- <<, >> - операторы сдвига влево или сдвига вправо на заданное количество бит;
- & битовый оператор И
- ^ битовое исключающее ИЛИ
- | битовый оператор ИЛИ.

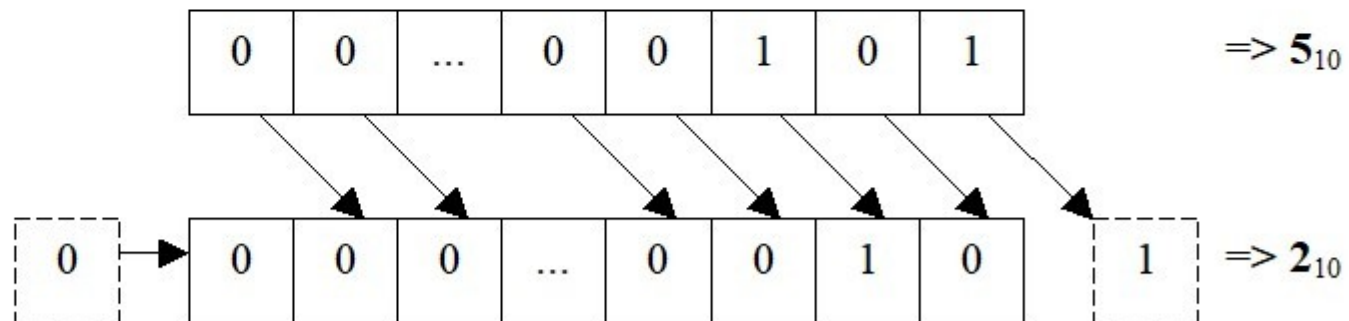
Пример: Операторы сдвига влево \ll , вправо \gg

$x = 5$
 $y = 5 \ll 1 \# y = 10$



a)

$x = 5$
 $y = x \gg 1 \# y = 2$



b)

Операторы сравнения

"=="	(равно)
">="	(больше или равно)
"<="	(меньше или равно)
"!="	(не равно)
"<"	(меньше)
">"	(больше)

Примечание: Когда мы хотим сравнить что две переменные равны то мы делаем так:

```
>> weight_one = 100
```

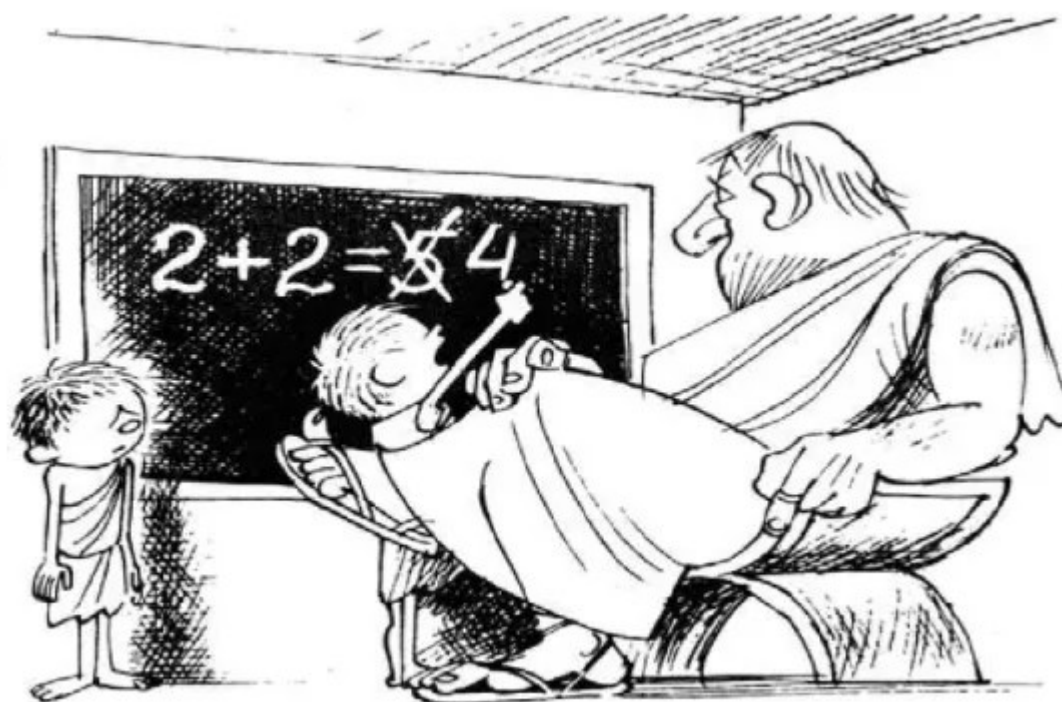
```
>> weight_two = 100
```

```
>> weight_one == weight_two      →      true
```

```
>> weight_one != 90              →      true
```

```
>> weight_one = weight_two      ←      не правильно !!
```

Питон мне друг но истина дороже.





Логические операторы

- AND** — логическое И
- OR** — логическое ИЛИ
- NOT** — логическое отрицание
- IN** — возвращает истину, если элемент присутствует в последовательности, иначе ложь.
- NOT IN** — возвращает истину если элемента нет в последовательности.
- IS** — проверка идентичности объекта

Python - Logical Operators

- not

x	not x
False	True
True	False

- and

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

- or

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

Operator Priority

<http://indersingh.blogspot.com/>



Применение логических операторов

```
x = 10  
y = 20
```

```
if x > 0 and y > 0:  
    print('Положительные числа')
```

```
if x > 0 or y > 0:  
    print('Хотя бы одно положительное')
```

```
if x > (0 or y) > 0:  
    print('Что будет')
```

```
>> 1 + True    →  ?
```

```
>> 1 + False   →  ?
```

Таблица приоритетов операций

Python Operator Precedence

Precedence	Operator Sign	Operator Name
Highest	**	Exponentiation
	+X, -X, ~X	Unary positive, unary negative, bitwise negation
	*, /, //, %	Multiplication, division, floor, division, modulus
	+, -	Addition, subtraction
	<<, >>	Left-shift, right-shift
	&	Bitwise AND
	^	Bitwise XOR
		Bitwise OR
	==, !=, <, <=, >, >=, is, is not	Comparison, Identity
	not	Boolean NOT
	and	Boolean AND
Lowest	or	Boolean OR



Вывод

Чтобы не запутаться в приоритетах операций ставьте в выражении круглые скобки ()

```
# Тестирование порядка выполнения выражения ( слева направо)
```

```
print(4 * 7 % 3)
```

```
# Результат: 1
```

```
print(2 * (10 % 5))
```

```
# Результат: 0
```



Коллекции

1. Строка (`str`)
2. Список (`list`)
3. Кортеж (`tuple`)
4. Словарь (`dict`)
5. Множество (`set`)



Строки- последовательности символов

Unicode

Строка задается либо парой одинарных " ", либо двойных "" "" или тройных """ """ кавычек. Существенной разницы в Python между одинарными и двойными кавычками нет.

```
>>> name = "" → пустая строка
```

```
>>> name = "" → пустая строка
```

```
>>> print("""
```

```
Usage: thingy [OPTIONS]
```

```
    -h      Display this usage message
```

```
    -H hostname      Hostname to connect to
```

```
""")
```


```
>>> name = "Peter I"
```



Внимание !!! Частая ошибка.

Не забывайте кавычки при задании строки, иначе значение будет интерпретироваться как переменная.

```
>>> str = Hello
```





Максимальная длина строки в Python

Максимальная длина строки зависит от платформы. Обычно это:

- $2^{31} - 1$ — для 32-битной платформы;
- $2^{63} - 1$ — для 64-битной платформы;

Константа `maxsize`, определенная в модуле `sys` :

```
>>> import sys >>> sys.maxsize 2_147_483_647
```

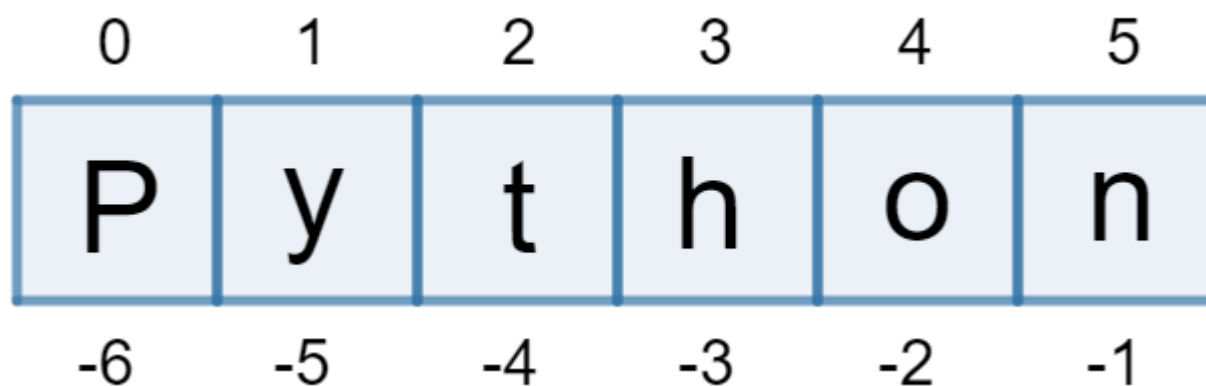
Функция `len()` вычисляет длину строки.

```
>>> len("Hello") → 5
```

Индексация строки

- Для получения символа в строке нужно обратиться по индексу позиции. Индексация строк начинается с 0

```
>>msg          = "Python"  
>>msg[0]       → "P"  
>>msg[-1]      → "n"
```





Операции сложения и умножения строк

Строки можно складывать (конкатенация строк)

```
>>string = "Hello" + " world !" → "Hello world !"
```

Строки можно умножать на целые числа. Происходит повторение строки n раз.

```
>>> "NO!" * 3 → 'NO!NO!NO!'
```



IMMUTABLE

Строка является неизменяемой (immutable) последовательностью СИМВОЛОВ.

```
>>msg = "Hello World!"
```

Попытка записать значение в начало слова, вызовет ошибку!

```
>>msg[0] = "S"
```

```
TypeError: 'str' object does not support item assignment
```

Строки (str). Срезы (slices)

Срезы (slices) – извлечение из данной строки одного символа или некоторого фрагмента (подстроки)

0	1	2	3	4
Н	Е	Л	Л	О
-5	-4	-3	-2	-1

Оператор извлечения среза из строки выглядит так: [X:Y].

X – индекс **начала среза**,

Y – индекс **окончания среза** (символ с номером Y в срез не входит).

```
>>> s = 'hello'
```

```
>>> s[1:4]    ИЛИ
```

```
'ell'
```

```
>>>
```

```
>>> s = 'hello'
```

```
>>> s[-4:-1]
```

```
'ell'
```

```
>>>
```




Срезы (slice)

#Пустое значение в начале обозначает позицию 0 индекса

```
>>>str = "Hello !"  
>>>str[:3]    # Соберем срез по индексам 0,1,2  
"Hel"  
>>>str[None:3] # Аналогично
```

#Пустое значение в конце обозначает позицию по концу строки

```
>>str[3:]     # Соберем срез по индексам от 2 до 6  
>>"lo !"
```



Проверка вхождения значения в последовательность.

```
>>> "P" in "Python"  
True
```

```
>>> "world" in "Hello world"  
True
```



Сравнение строк при помощи == и !=

```
>>> language = 'chinese'
>>> print(language == 'chinese')    → True
>>> print(language != 'chinese')    → False
```

```
>>> 'chinese' > 'italiano'
```

Ответ: ?

Массивы байт.

Bytearray в python – массив байт. От строк отличается только тем, что является изменяемым.

```
>>> b = bytearray(b'hello world!')
>>> b
bytearray(b'hello world!')
>>> b[0]
104
>>> b[0] = b'h'
Traceback (most recent call last):
  File "", line 1, in
    b[0] = b'h'
TypeError: an integer is required
>>> b[0] = 105
>>> b
bytearray(b'iello world!')
```



Список (mutable)

Создать список можно двумя способами:

Вызывать функцию `list()`

```
lst = list()
```

Использовать квадратные скобки

```
lst = [] → Задали пустой список
```

Пример:

```
lst = list([1, 4, 5])
```

```
lst = list("hello")
```

```
lst = [1, 4, 5]
```




Элементы списка разных типов

Пример:

```
>>> lst = [10, True, [1,2], "#ffffff"]
```

```
>>> type(lst)
```

```
<class 'list'>
```

Кортеж

```
# пустой кортеж
```

```
empty_tuple = ()
```

```
#упаковка кортежа из 4-х элементов разных типов
```

```
four_el_tuple = (36.6, 'Normal', None, False)
```

```
type(four_el_tuple)
```

```
<class 'tuple'>
```

```
>>> four_el_tuple[0] → 36.6
```

Определение словаря

```
dict = {k:v}
```

Словарь задается парой **ключ: значение**,

```
dic = {  
    <key>: <value>,  
    <key>: <value>,  
    .  
    .  
    .  
    <key>: <value>  
}
```

Пример 1:

```
person = {  
    'name': 'Маша',  
    'login': 'masha',  
    'age': 25,  
    'email': 'masha@yandex.ru',  
    'password': 'fhei23jj~'  
}  
  
print (type (person) )  
<class 'dict'>
```

Множества

Мы также можем создать множество с элементами разных типов. Например:

```
>>> mixed_set = {2.0, "Nicholas", (1, 2, 3)}  
>>> print(mixed_set)  
{'Nicholas', 2.0, (1, 2, 3)}
```



Конструкции ветвления if

```
str = "Hello"
```

```
if str:
```

```
    print("Не пустая строка!")
```



Конструкции ветвления if/else

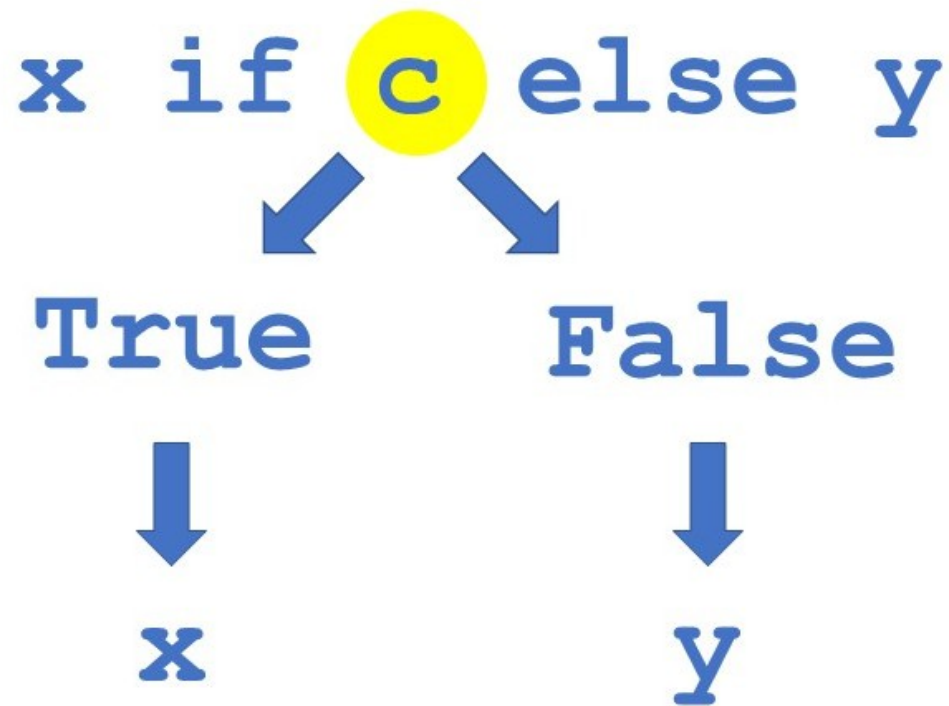
```
age = input("Enter you age")
age = int(age)
if age <= 18:
    print("Доступ запрещен!")
else:
    print("Доступ разрешен!")
```



Конструкции ветвления if/elif/else

```
age = input("Enter you age")
age = int(age)
if age <= 16:
    print("Школьник!")
elif 16 < age <= 25:
    print("Студент")
elif 25 < age <= 40:
    print("Сотрудник компании")
else:
    print("Еще не существует!")
```


Ternary Operator





Тернарный оператор

```
x = 1
```

```
y = 2
```

```
maximum = x if x > y else y
```

Pattern Matching in Python 3.10



```
match status:
    case 200:
        print("OK")
    case 301 | 302:
        print("Redirect")
    case 404:
        print("Not Found")
```



Пример:

```
color = "RED"

match color:
    case "RED":
        print("Флаг красный")
    case "GREEN":
        print("Трава зеленая")
    case "BLUE":
        print("Небо синее")
```



Цикл for

Общая конструкция:

for цель **in** объект:

операторы

if проверка: break # выход из цикла

if проверка: continue # переход в начало цикла

else:

Операторы # ветка else выполняется если не было выхода с
помощью оператора break



Итерация списка с использованием for

```
input_list = [10, "S", 15, "A", 1]
for x in input_list:
    print(x)
```

Вывод:

10

"S"

15

"A"

1



Функция range()

Функция `range()` применяется для генерации индексов в цикле `for`. Генерирует диапазон чисел в зависимости от условия.

```
>>> range(5)
```

```
>>> list(range(5))
```

```
[0, 1, 2, 3, 4]
```

```
>>> list(range(-5, 5))
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```



Пример:

```
for x in range(3):  
    print('result', x )
```




А можно так ?

```
for x in 3:  
    print('result', x )
```



Оператор break

```
>>> for i in 'hello world':  
...     if i == 'o':  
...         break  
...     print(i * 2, end=' ')  
...  
hheel111
```



Оператор continue

```
>>> for i in 'hello world':  
...     if i == 'o':  
...         continue  
...     print(i * 2, end=' ')  
...  
hheel111  wwrrl1dd
```



Волшебное слово else

```
>>> for i in 'hello world':  
...     if i == 'a':  
...         break  
...     else:  
...         print('Буквы а в строке нет')  
...
```

Буквы а в строке нет



Оператор pass

```
for i in 'hello world':
```

```
    ????
```

← Что нужно поставить чтобы for заработал ?



Цикл while

Общая конструкция:

while проверка условия:

операторы

if проверка: break # выход из цикла

if проверка: continue # переход в начало цикла

else:

Операторы # ветка else выполняется если не было выхода с
помощью оператора break



Пример

```
>>> i = 5
>>> while i < 15:
...     print(i)
...     i = i + 2
...
5
7
9
11
13
```



Бесконечный цикл

```
>>> i = 5
>>> while True:
...     print(i)
...     i = i + 2
...     if i == 7: braek
```

Что выведет код ?



Функции

```
# объявление функции my_function()
```

```
def my_function([параметр1, параметр2, ...]):  
    # тело функции
```

```
    # возвращаемое значение  
    return result # необязательно
```

```
# вызов функции  
my_function([аргумент1 , аргумент2, ...] )
```

```
type(my_function)  
<class 'function'>    - еще один тип в Python
```



Пример:

#Определение функции:

```
def summ(x, y):  
    result = x + y  
    return result
```

#ВЫЗОВ функции

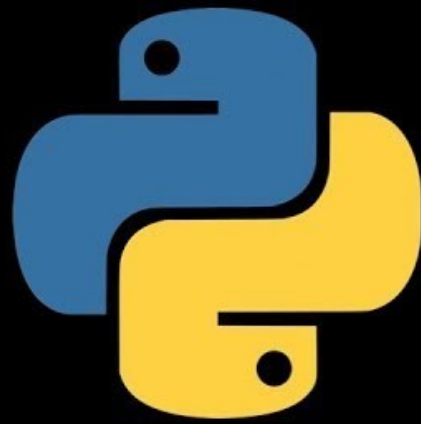
```
a = 100
```

```
b = -50
```

```
answer = summ(a, b)
```

```
print(answer)
```

```
50
```



PYTHON

PROGRAMMING