

Язык программирования Python



- Python – высокоуровневый язык программирования общего назначения с динамической строгой типизацией и автоматическим управлением памятью, ориентированный на повышение производительности разработчика, читаемости кода и его качества, а также на обеспечение кроссплатформенности написанных на нём программ.

Установка

1. Установить Python (<https://www.python.org/>). В OS X и Unix уже установлен.
2. Только для Windows. При установке выбрать опцию с заданием переменных окружения (лучше установить пакет [Anaconda](#)).

Среда разработки

1. Jupyter Notebook из пакета [Anaconda](#).
2. Текстовые редакторы: [Visual Studio Code](#), Sublime, Atom, Notepad++ и др.
3. IDE: [PyCharm \(Community Edition\)](#), Visual Studio, Spider и др.

Язык программирования Python

6

Кто использует Python

- Google
- YouTube
- Dropbox
- BitTorrent
- iRobot
- Netflix
- Intel, Cisco
- NASA

Сильные стороны

- Качество программного кода
- Продуктивность труда
- Кроссплатформенность
- Библиотеки

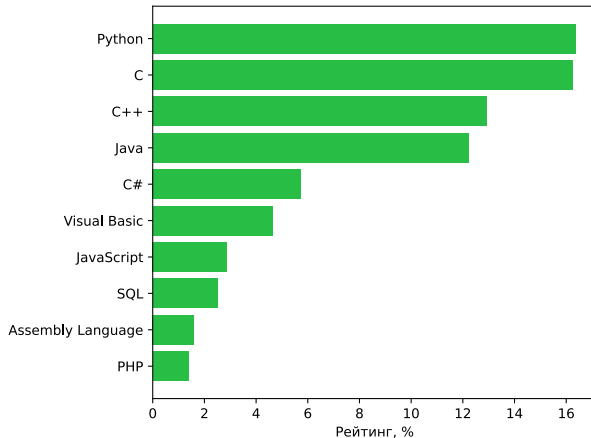


Рисунок 1 – Индекс [TIOBE](#) на январь 2023 года

Логический тип данных

- Для логического типа данных `bool` можно объявлять логические переменные, инициализируя их логическими значениями или присваивая им результат вычисления логических выражений.
- Логических констант в Python две: `True` (истина) и `False` (ложь).

```
1 >>> x = True
2
3 >>> y = False
4
5 >>> z = 2 > -1
6
7 >>> print(x, y, z)
8 (True, False, True)
```

Операции сравнения

Логические выражения являются аналогом математической формулы, результатом его вычисления будет одна из двух логических констант – **True** или **False** .

Операция	Описание
$x < y$	Меньше
$x \leq y$	Меньше или равно
$x > y$	Больше
$x \geq y$	Больше или равно
$x == y$	Равно
$x != y$	Не равно

- Приоритет операций сравнения ниже, чем у арифметических операций:

```
1 >>> print(2 + 3 * 5 > 7 / 2 + 3.5)
2 True
```

Операции сравнения

11

Операции сравнения в Python сравнивают относительные величины своих операндов и возвращают результат логического типа, значение которого обычно используется для выбора следующего действия в более крупном операторе или программе:

```
1 >>> 1 < 2      # Меньше
2 True
3
4 >>> 2.0 >= 1    # Больше или равно: int приводится к float
5 True
6
7 >>> 3.0 == 3.0  # Значения равны
8 True
9
10 >>> 3.0 != 3.0 # Значения не равны
11 False
```

Логическая операция not

13

- Унарная логическая операция `not` называется логическим отрицанием («НЕ», инверсия) и указывается перед логическим выражением для получения его противоположного значения.
- Приоритет операции `not` ниже, чем у операций сравнения, поэтому следующее за ней логическое выражение можно не заключать в скобки:

```
1 >>> print(3 + 5 >= 8)
2 True
3
4 >>> print(not 3 + 5 >= 8)
5 False
```

Логическая операция and

14

- Бинарная логическая операция **and** называется логическим умножением (логическое «И»).
- Результатом операции **and** будет **True** только тогда, когда оба логических выражения будут иметь значения **True**, а в прочих случаях результатом будет **False**.

```
1 >>> print(9 + 3 < 10 and 2 + 2 == 4)
2 False
3
4 >>> print(4 + 2 != 0 and 10 * 2 == 20)
5 True
```

Логическая операция or

- Бинарная логическая операция `or` называется логическим сложением (логическое «ИЛИ»).
- Результатом операции `or` будет `False` только тогда, когда оба логических выражения будут иметь значения `False`, а в прочих случаях результатом будет `True`.

```
1 >>> print(9 + 3 < 10 or 2 + 2 == 4)
2 True
3
4 >>> print(4 + 2 < 0 or 10 * 2 >= 100)
5 False
```


Приведение к логическому типу

- Конструктор типа `bool(x)` может использоваться для приведения любого значения к логическому типу (если это значение можно интерпретировать как логический тип). Если аргумент `*x*` ложь или опущен, то будет возвращено значение `False`.

```
1 >>> print(bool(1), bool(-1.0))
2 True True
3
4 >>> print(bool(0), bool(0.0))
5 False False
6
7 >>> print(bool())
8 False
```

Сцепленные операции сравнения

18

В Python также есть возможность выстраивать цепочки из нескольких операторов сравнения для выполнения проверок вхождения в диапазон.

Сцепленные сравнения являются краткой записью для более массивных булевых выражений.

```
1 >>> x < y > z
2 False
3
4 >>> x < y and y > z
5 False
6
7 >>> 3 < 6 < 9.0 < 12
8 True
9
10 >>> 3 > 6 > 9 > 12
11 False
```

Сравнение чисел с плавающей точкой

20

Числа с плавающей точкой в логических операциях сравнения могут вести себя неожиданным образом, требуя определенных преобразований для содержательного сравнения:

```
1 >>> 0.1 + 1.1 == 1.2 # Кажется, что должно быть True
2 False
3
4 >>> 0.1 + 1.1 # Близко к 1.2, но неточно: ограниченная точность
5 1.2000000000000002
6
7 >>> round(0.1 + 1.1, 1) == round(1.2, 1) # Работает нормально при округлении
8 True
```

Оператор if

- Оператор `if` используется для проверки условий: **если** условие верно, то выполняется блок выражений (называемый «if-блок»), **иначе** выполняется другой блок выражений (называемый «else-блок»).
- Блок «else» является необязательным.

```
1 >>> if 1:  
2 ...     print('true')  
3 ...  
4 true
```

Напоминаем, что 1 – это `True`, таким образом, проверка в операторе всегда проходит. Для обработки ложного результата понадобится добавить часть `else` :

```
1 >>> if not 1:  
2 ...     print('true')  
3 ... else:  
4 ...     print('false')  
5 ...  
6 false
```

Множественное ветвление

Рассмотрим пример оператора `if`, содержащего все необязательные части:

```
1 >>> number = 23
2
3 >>> guess = int(input('Введите целое число: '))
4
5 >>> if guess == number:
6 ...     print('Поздравляю, Вы угадали,') # Здесь начинается новый блок
7 ...     print('хотя и не выиграли никакого приза!') # Конец нового блока
8 ... elif guess < number:
9 ...     print('Нет, загаданное число немного больше этого.') # Ещё один блок
10 ... # Внутри блока Вы можете выполнять любые операторы
11 ... else:
12 ...     print('Нет, загаданное число немного меньше этого.')
```

- Интерпретатор выполняет операторы, вложенные внутрь первой проверки, которая вернет `True`, или часть `else`, если все проверки вернули `False`.
- Части `elif` и `else` могут быть опущены, а в каждой части допускается указывать более одного вложенного оператора.
- Части `if`, `elif` и `else` связываются друг с другом одинаковыми отступами.

Ограничения блоков: правила отступов

- Python определяет границы блоков автоматически по **отступам** строк.
- Все операторы с отступами на одинаковое расстояние вправо принадлежат тому же самому блоку кода.
- Блок заканчивается тогда, когда встречается конец файла или строка с меньшим отступом, и более глубоко вложенный блок просто смещается дальше вправо, чем операторы во включающем блоке.

```
1 x = 10
2
3 if x:
4     y = 20
5
6     if y:
7         print("block 2")
8
9     print("block 1")
10
11 print("block 0")
```

Блок 0 (block 0)

Строка заголовка

Блок 1 (block 1)

Строка заголовка

Блок 2 (block 2)

Блок 1 (block 1)

Блок 0 (block 0)

Тернарное выражение if/else

26

- Зачастую элементы, использованные в операторе `if`, достаточно просты, так что распространение такого оператора на четыре строки выглядит излишеством.
- В других случаях конструкцию подобного рода может понадобиться вложить в более крупный оператор, а не присваивать ее результат какой-то переменной.
- По указанным причинам в Python был добавлен новый формат условного выражения, который позволяет определить то же самое в одном действии (тернарный оператор).

```
1 if x:  
2     a = y  
3 else:  
4     a = z
```

```
a = y if x else z
```

Тернарное выражение if/else

27

```
1 >>> x = 'f' if 10 else 't' # Числа, не равные нулю, истина
2
3 >>> print(x)
4 f
5
6 >>> x = 'f' if 0 else 't' # Ноль - это ложь
7
8 >>> print(x)
9 t
```

Использовать тернарный оператор нужно крайне умеренно и только в тех случаях, когда все его составные части относительно просты, иначе предпочтительнее использовать форму полного оператора `if` для облегчения его будущего модифицирования.

Операторы цикла

- Алгоритмы решения многих задач требуют некоторого количества повторений своих отдельных частей.
- Такие повторяющиеся участки называют циклическими, а операторы языка Python, реализующие соответствующие повторения – **операторами цикла**.
- Цикл состоит из **заголовка цикла** и **тела цикла**.
- Заголовок определяет условие прекращения (или выполнения) цикла, а тело цикла содержит операторы, которые нужно повторять.

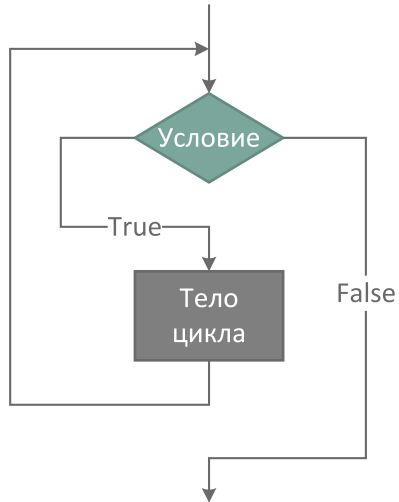
Операторы цикла в Python:

1. Цикл `while`
2. Цикл `for`

Оператор цикла while

30

- Оператор `while` многократно повторяет блок операторов до тех пор, пока проверка в заголовочной части оценивается как истина.
- Управление продолжает возвращаться к началу оператора, пока проверка не даст ложное значение. Когда результат проверки становится ложным, управление переходит на оператор, следующий после блока `while`.
- Если проверка оценивается в ложное значение с самого начала, тогда тело цикла никогда не выполнится и оператор `while` пропускается.



Оператор цикла while

Общий формат цикла `while` :

```
while expression:  # Проверка цикла
    operator_1      # Тело цикла
    operator_2
    ...
    operator_n
```

Цикл `while` можно использовать:

- в математических итерационных алгоритмах для проведения вычислений с заданной точностью;
- при вводе данных, когда их количество заранее неизвестно, а условие завершения ввода определено некоторым введенным значением;
- при поиске нужного элемента в какой-либо структуре данных.

Примеры использования цикла while

32

Вывод чисел в диапазоне $[0, 10)$ с шагом 1:

```
1 >>> a, b = 0, 10
2
3 >>> while a < b:
4 ...     print(a, end='')
5 ...     a += 1
6 0123456789
```

Примеры использования цикла while

33

```
1 >>> number = 24
2
3 >>> running = True
4
5 >>> while running:
6 ...     guess = int(input('Введите целое число: '))
7 ...
8 ...     if guess == number:
9 ...         print('Поздравляем, Вы угадали!')
10 ...         running = False # это останавливает цикл while
11 ...
12 ...     elif guess < number:
13 ...         print('Нет, загаданное число немного больше этого')
14 ...
15 ...     else:
16 ...         print('Нет, загаданное число немного меньше этого')
17 ...
18 >>> print('Завершение.') # другие операторы программы
19 Введите целое число: 12
20 Введите целое число: 24
21 Поздравляем, Вы угадали!
22 Завершение.
```

Оператор цикла for

34

- Оператор `for...in` также является оператором цикла, который осуществляет итерацию по **последовательности** объектов, т.е. проходит через каждый элемент в последовательности.
- **Последовательность** – это упорядоченный или неупорядоченный набор элементов.
- Во многих случаях в заголовке цикла `for` используется функция `range()`, которая является генератором арифметических прогрессий:

```
1 >>> for i in range(10): # 10 не включительно
2 ...     print(i, end=' ')
3 0 1 2 3 4 5 6 7 8 9
```

Оператор цикла for

35

Функция `range()` – генератор арифметических прогрессий:

```
1 >>> for i in range(1, 11): # можно задать начальное значение
2 ...     print(i, end=' ')
3 1 2 3 4 5 6 7 8 9 10
4
5 >>> for i in range(0, 11, 2): # также можно менять шаг
6 ...     print(i, end=' ')
7 0 2 4 6 8 10
```

Вложенные циклы for

36

Операторы цикла `for` могут быть вложены друг в друга на произвольную глубину:

```
1 >>> for i in range(3):
2 ...     for j in range(3):
3 ...         if i != j:
4 ...             print(i, j, round(1 / (i + j), 2))
5 ...
6 0 1 1.0
7 0 2 0.5
8 1 0 1.0
9 1 2 0.33
10 2 0 0.5
11 2 1 0.33
```


Оператор break

Оператор **break** выполняет немедленный выход из цикла, т.е. остановку выполнения команд, даже если условие выполнения цикла еще не приняло значение **False** или последовательность элементов не закончилась.

```
1 >>> while True:
2 ...     name = input('Enter name: ')
3 ...     if name == 'stop':
4 ...         break
5 ...     age = input('Enter age: ')
6 ...     print('Hello', name, '=>', int(age) * 2)
7 ...
8 Enter name: John
9 Enter age: 35
10 Hello John => 70
11 Enter name: Julia
12 Enter age: 24
13 Hello Julia => 48
14 Enter name: stop
```

Оператор continue

39

Оператор `continue` используется для немедленного перехода в начало цикла.

```
1 >>> x = 10
2
3 >>> while x:
4 ...     x -= 1
5 ...     # Нечетное? Тогда пропустить
6 ...     if x % 2 != 0:
7 ...         continue
8 ...     print(x, end=' ')
9 8 6 4 2 0
```

```
1 >>> x = 10
2
3 >>> while x:
4 ...     x -= 1
5 ...     # Четное? Тогда выводим
6 ...     if x % 2 == 0:
7 ...         print(x, end=' ')
8 8 6 4 2 0
```

Оператор pass

40

- Оператор `pass` – это заполнитель, обозначающий отсутствие действий, используемый в ситуациях, когда синтаксис требует оператора, но нет возможности выполнить что-либо полезное.
- Данный оператор часто применяется для кодирования пустого тела для составного оператора.

К примеру, с помощью `pass` можно написать бесконечный цикл, который на каждом проходе ничего не делает:

```
while True:  
    pass # Для прекращения работы нажмите <Ctrl+C>!
```

Конструкция else цикла

Полная форма записи циклов `while` и `for` выглядит следующим образом:

```
1 while condition():
2     operators
3
4     # Выход с пропуском else
5     if exit_test():
6         break
7
8     # Переход к заголовку цикла
9     if skip_test():
10        continue
11
12 # Выполняется, если не было break
13 else:
14     operators
```

```
1 for x in collection:
2     operators
3
4     # Выход с пропуском else
5     if exit_test():
6         break
7
8     # Переход к заголовку цикла
9     if skip_test():
10        continue
11
12 # Выполняется, если не было break
13 else:
14     operators
```

Если циклы `while` или `for` прервать оператором `break`, соответствующие им блоки `else` выполняться не будут.

Конструкция else цикла

42

В приведенном примере выполняется проверка, является ли положительное целое число y простым, за счет поиска сомножителей больше 1:

```
1  x = y // 2  # Для y > 1
2
3  while x > 1:
4      if y % x == 0: # Остаток от деления
5          print(y, "has factor", x)  # Имеет сомножитель
6          break # Пропуск else
7
8      x -= 1
9
10 else: # Нормальный выход
11     print(y, "is prime") # Является простым
```

Пример 1 – Fizz Buzz

Дано: Положительное целое число (`int`)

«Fizz buzz» – это игра со словами. Необходимо написать программу, которая принимает положительное целое число и возвращает следующие значения:

1. «Fizz Buzz», если число делится на 3 и 5;
2. «Fizz», если число делится на 3;
3. «Buzz», если число делится на 5;
4. Исходное число в остальных случаях.

Пример:

$x = 15$	результат:	«Fizz Buzz»
$x = 6$	результат:	«Fizz»
$x = 5$	результат:	«Buzz»
$x = 7$	результат:	7

Пример 1 – Fizz Buzz

45

```
1 x = int(input('Введите положительное целое число: '))
2
3 if not x % 3 and not x % 5:
4     print('Fizz Buzz')
5
6 elif not x % 3:
7     print('Fizz')
8
9 elif not x % 5:
10    print('Buzz')
11
12 else:
13    print(x)
```

Пример 2 – Вычисление факториала числа n

46

Необходимо написать программу для вычисления $n!$:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n - 1) \cdot n$$

где n – положительное целое число.

Пример 2 – Вычисление факториала числа n

47

С использованием цикла `while` :

```
1  n = int(input('Введите положительное целое число: '))
2
3  i = 1
4  f = 1
5
6  while i <= n:
7      f *= i
8      i += 1
9
10 print('n! =', f)
```

Пример 2 – Вычисление факториала числа n

48

С использованием цикла `for` :

```
1  n = int(input('Введите положительное целое число: '))
2
3  f = 1
4
5  for i in range(1, n + 1):
6      f *= i
7
8  print('n! =', f)
```