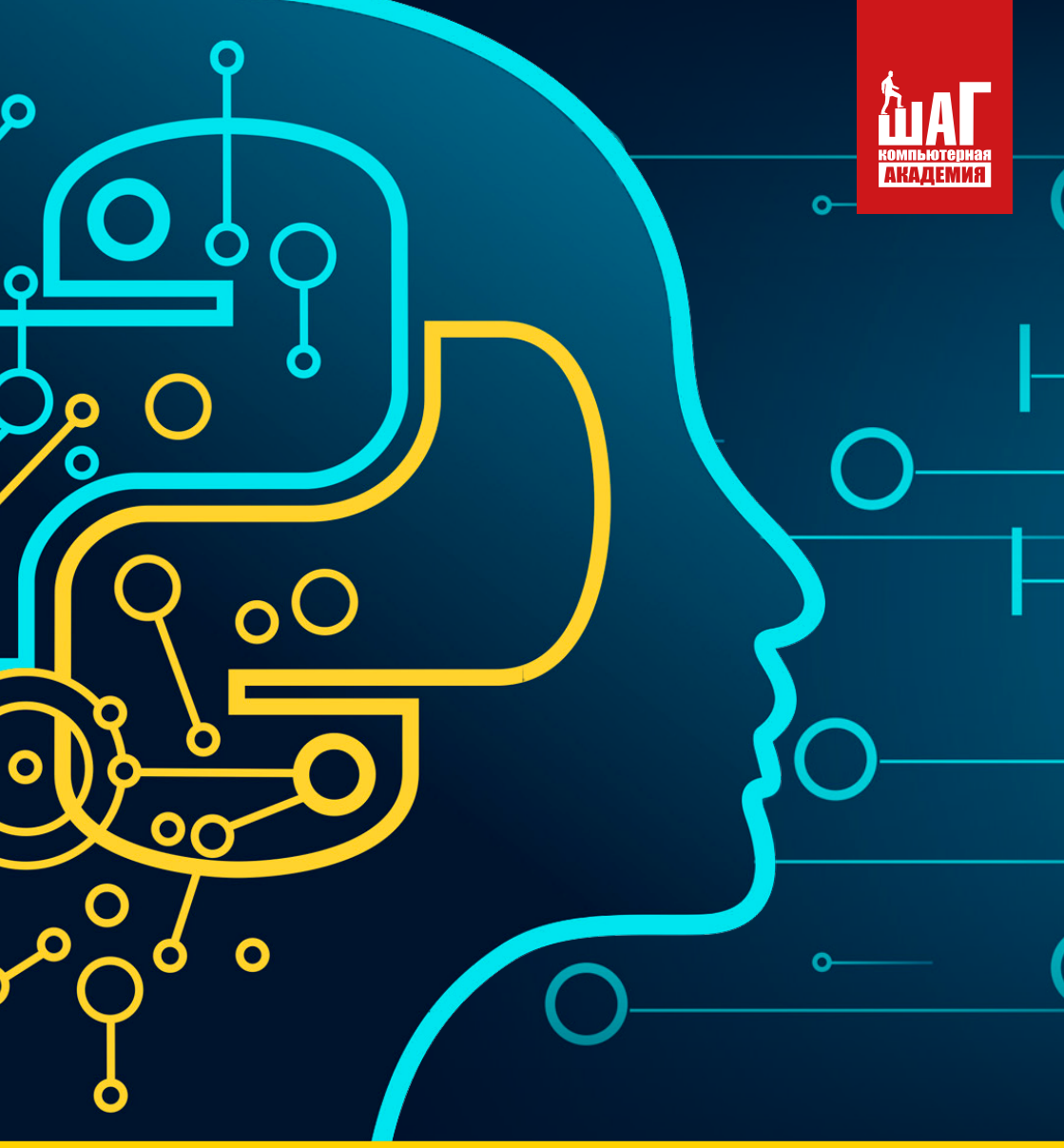

ШАГ
компьютерная
АКАДЕМИЯ

Основы программирования на языке Python

Урок 3

Циклы в языке Python. Логические и побитовые операции

Contents

Циклы в языке Python

Создание циклов с помощью while

Согласны ли вы с утверждением, представленным ниже?

пока есть что делать
делай это

Обратите внимание, что эта запись также означает, что если нечего делать, ничего не произойдет.

В общем, в Python цикл может быть представлен следующим образом:

while conditional_expression:
instruction

Если вы заметили некоторые сходства с инструкцией if, все в порядке. Действительно, синтаксическая разница только одна: вместо слова if вы используете слово while.

Семантическое различие более важно: когда условие выполнено, if выполняет свое действие только один раз; while повторяет выполнение до тех пор, пока условие оценивается как True.

Примечание: все правила, касающиеся отступов, применимы и здесь. Мы вскоре расскажем о них.

Посмотрите на алгоритм ниже:

```
while conditional_expression:
    instruction_one
    instruction_two
    instruction_three
```

```
    :
    :
    instruction_n
```

Теперь важно помнить, что:

- внутри цикла while, вы должны (как и в случае с if) сделать одинаковый отступ в коде для всех инструкций;
- инструкция или набор инструкций, выполняемых внутри цикла while, называется телом цикла;
- если условие является ложным False (*равно нулю*) уже при первой проверке, тело не выполняется ни разу (обратите внимание на аналогию отсутствия необходимости делать что-либо, если нечего делать);
- тело должно иметь возможность изменять значение условия, потому что, если условие вначале истинно True, тело может непрерывно выполняться до бесконечности — обратите внимание, что выполнение действия обычно уменьшает количество действий).

Бесконечный цикл

Бесконечный цикл, также называемый вечным циклом, представляет собой последовательность инструкций в программе, которые повторяются бесконечно (зациклены).

Вот пример цикла, который не может завершить свое выполнение:

```
while True:
    print("Я застрял внутри цикла.")
```

Этот цикл будет бесконечно выводить на экран «Я застрял внутри цикла».

Если вы хотите получить больше опыта, увидев, как ведет себя бесконечный цикл, запустите IDLE, создайте новый файл, скопируйте и вставьте приведенный выше код, сохраните свой файл и запустите программу. То, что вы увидите, это бесконечная последовательность вывода строки «Я застрял внутри цикла» в окне консоли Python. Чтобы завершить вашу программу, просто нажмите Ctrl-C (или Ctrl-Break на некоторых компьютерах). Это вызовет так называемое исключение KeyboardInterrupt и позволит вашей программе выйти из цикла. Мы поговорим об этом позже.

Давайте вернемся к наброску алгоритма, который мы вам недавно показали. Мы собираемся показать вам, как использовать этот недавно изученный цикл, чтобы найти наибольшее число из большого набора введенных данных.

Проанализируйте программу внимательно. Найдите тело цикла и выясните, как происходит выход из тела:

```
# мы будем хранить здесь текущее наибольшее число
largest_number = -999999999

# ввод первого значения
number = int(input("Введите число или напишите -1,
    чтобы остановить программу: "))

# если число не равно -1, мы продолжаем
while number != -1:

    # число больше, чем largest_number?
    if number > largest_number:
```

```
        # да, обновляем largest_number
        largest_number = number

    # ввод следующего число
    number = int(input("Введите число или напишите -1,
        чтобы остановить программу: "))

# вывод наибольшего числа
print("Наибольшее число:", largest_number)
```

Проверьте, как этот код реализует алгоритм, который мы показали вам ранее.

Цикл while: примеры

Давайте посмотрим на другой пример, использующий цикл while. Просмотрите комментариям, чтобы понять идею и решение.

```
# Программа, которая читает последовательность чисел
# и подсчитывает, сколько чисел четный и сколько
# нечетных.
# Программа завершается, когда введен ноль.
odd_numbers = 0
even_numbers = 0

# чтение первого числа number =
# int(input("Введите число или напишите 0,
# чтобы остановить программу: "))
# 0 прекращает выполнение while number != 0:
    # проверка является ли число нечетным
    # if number % 2 == 1:
        # увеличение счетчика odd_numbers
        odd_numbers += 1
    else:
```

```
        # увеличение счетчика even_numbers
        even_numbers += 1
    # чтение следующего число
    number = int(input("Введите число или напишите 0,
        чтобы остановить программу: "))
# вывод результатов
print("Количество нечетных чисел:", odd_numbers)
print("Количество четных чисел:", even_numbers)
```

Некоторые выражения могут быть упрощены без изменения поведения программы.

Попытайтесь вспомнить, как Python интерпретирует истинность условия, и обратите внимание, что эти две формы эквивалентны:

```
while number != 0: и while number:.
```

Условие, которое проверяет, является ли число нечетным, также может быть написано в этих эквивалентных вариантах:

```
if number % 2 == 1: и if number % 2:.
```

Использование переменной counter для выхода из цикла

Посмотрите на фрагмент кода ниже:

```
counter = 5
while counter != 0:
    print("Внутри цикла.", counter)
    counter -= 1
print("Вне цикла.", counter)
```

Этот код предназначен для вывода строки «Внутри цикла» и значения, хранящееся в переменной counter в течение данного цикла, ровно пять раз. Как только условие не выполнено (переменная counter достигла 0), цикл завершается, и появляется сообщение «Вне цикла», а также выводится значение, хранящееся в counter.

Но есть одна вещь, которую можно записать более компактно — условие цикла while.

Вы видите разницу?

```
counter = 5
while counter:
    print("Внутри цикла.", counter)
    counter -= 1
print("Вне цикла.", counter)
```

Более компактно, чем раньше? Немного. Более разборчиво? Спорно.

ПОМНИТЕ!

Не считайте своей обязанностью писать свои программы так, чтобы они всегда были самыми короткими и самыми компактными. Читаемость может быть более важным фактором. Пусть ваш код будет готов к новому программисту.

Создание циклов с помощью for

Другой тип цикла, доступный в Python, исходит из наблюдения, что иногда важнее считать «обороты» цикла, чем проверять условия. Представьте, что тело цикла должно быть выполнено ровно сто раз. Если вы хотите использовать цикл while, это может выглядеть так:

```
i = 0
while i < 100:
    # do_something()
    i += 1
```

Было бы неплохо, если бы кто-то мог сделать этот скучный подсчет вместо вас. Это возможно?

Конечно, есть — есть специальный цикл для таких задач, и он назван for.

На самом деле цикл for предназначен для выполнения более сложных задач — он может «просматривать» большие коллекции данных элемент за элементом. Мы покажем вам, как это сделать в ближайшее время, но сейчас мы представим более простой вариант его применения.

Посмотрите на фрагмент кода:

```
for i in range(100):
    # do_something()
    pass
```

Есть несколько новых элементов. Позвольте рассказать вам о них:

- ключевое слово for открывает цикл for;

  Примечание: *после него нет условия; вам не нужно думать об условиях, поскольку они проверяются внутри, без какого-либо вмешательства;*

- любая переменная после ключевого слова for является управляющей переменной цикла; он считает обороты цикла и делает это автоматически;

- ключевое слово in вводит элемент синтаксиса, описывающий диапазон возможных значений, назначаемых управляющей переменной;
- функция range() (это особенная функция) отвечает за генерацию всех желаемых значений управляющей переменной; в нашем примере функция создаст (мы можем даже сказать, что она будет снабжать цикл) последующими значениями из следующего набора: 0, 1, 2 .. 97, 98, 99; примечание: в этом случае функция range() начинает свою работу с 0 и завершает ее за один шаг (одно целое число) перед значением своего аргумента;
- обратите внимание на ключевое слово pass внутри тела цикла — вообще ничего не делает; это пустая инструкция — мы поместили ее здесь, потому что синтаксис цикла for требует по крайней мере одну инструкцию внутри тела (кстати, if, elif, else и while требуют то же самое)

Наши следующие примеры будут немного скромнее в количестве повторений цикла.

Посмотрите на фрагмент ниже. Можете ли вы предсказать его результат?

```
for i in range(10):
print("Значение i в настоящее время", i)
```

Запустите код, чтобы проверить, были ли вы правы.

Примечание:

- *цикл был выполнен десять раз (это аргумент функции range())*

- *последнее значение управляющей переменной равно 9 (не 10, поскольку начинается с 0, а не с 1)*

Вызов функции range() может быть снабжен двумя аргументами, а не одним:

```
for i in range(2, 8):
    print("Значение i в настоящее время ", i)
```

В этом случае первый аргумент определяет начальное (первое) значение управляющей переменной.

Последний аргумент показывает первое значение, которое не будет назначено управляющей переменной.

Примечание: *функция range() принимает в качестве аргументов только целые числа и генерирует последовательности целых чисел.*

Можете ли вы угадать вывод программы? Запустите ее, чтобы проверить, были ли вы правы.

Первое показанное значение равно 2 (взято из первого аргумента range()).

Последнее — 7 (хотя второй аргумент range() это 8).

Подробная информация о цикле for и функции range() с тремя аргументами

Функция range() также может принимать три аргумента — взгляните на код.

Третий аргумент — это инкремент, т.е. значение, добавляемое для управления переменной при каждом обороте цикла (как вы можете предположить, значение инкремента по умолчанию равно 1).

Можете ли вы сказать нам, сколько строк появится в консоли и какие значения они будут содержать?

Запустите программу, чтобы узнать, были ли вы правы.

```
for i in range(2, 8, 3):
    print("Значение i в настоящее время", i)
```

Вы должны увидеть следующие строки в окне консоли:

```
Значение i в настоящее время 2
Значение i в настоящее время 5
```

Знаете почему? Первый аргумент, переданный функции range(), сообщает нам, каково начальное число последовательности (т.е. 2 при выводе). Второй аргумент сообщает функции, где остановить последовательность (функция генерирует числа вплоть до числа, указанного вторым аргументом, но не включает его). Наконец, третий аргумент указывает шаг, который фактически означает разницу между каждым числом в последовательности чисел, сгенерированных функцией.

2 (начальное число) → 5 (2 увеличенное на 3 равно 5 – число в диапазоне от 2 до 8) → 8 (5 увеличенное на 3 равно 8 — число не в диапазоне от 2 до 8, поскольку параметр остановки не входит в последовательность чисел, генерируемых функцией.)

> **Примечание:** *если набор, сгенерированный функцией range(), пуст, цикл не будет выполнять свое тело вообще.*

Точно так же как здесь — не будет никакого вывода:

```
for i in range(1, 1):
    print("Значение i в настоящее время", i)
```

> **Примечание:** *набор, сгенерированный range(), должен быть отсортирован в порядке возрастания. Нет способа заставить range() создать набор в другом виде. Это означает, что второй аргумент range() должен быть больше первого.*

Таким образом, в этой случае также не будет никакого вывода:

```
for i in range(2, 1):
    print("Значение i в настоящее время", i)
```

Давайте посмотрим на короткую программу, задача которой написать несколько первых степеней числа два:

```
pow = 1
for exp in range(16):
    print("2 в", exp, "степени равно", pow)
    pow *= 2
```

Переменная exp используется в качестве управляющей для цикла и указывает значение показателя степени. Само возведение в степень заменяется умножением на два. Так как 2^0 равно 1, то 2×1 равно 2^1, 2×2^1 равно 2^2 и так далее. Каков максимальный показатель степени, для которого наша программа все еще выводит результат?

Запустите код и проверьте, соответствует ли результат вашим ожиданиям.

Управление циклами на языке Python

Операторы break и continue

До сих пор мы рассматривали тело цикла как неделимую последовательность инструкций, которые выполняются полностью на каждом обороте цикла. Однако, как разработчик, вы можете столкнуться со следующими вариантами:

- оказывается, что нет необходимости продолжать цикл в целом; вам следует воздерживаться от дальнейшего выполнения тела цикла и идти дальше;
- оказывается, что вам нужно начать следующий оборот цикла без завершения выполнения текущего оборота.

Python предоставляет две специальные инструкции для реализации обеих этих задач. Скажем ради точности, что их существование в языке не является необходимым — опытный программист может написать любой алгоритм без этих инструкций. Такие дополнения, которые не улучшают выразительные возможности языка, а лишь упрощают работу разработчика, иногда называют синтаксическим сахаром.

Такие инструкции включают:

- break — немедленно выходит из цикла и безоговорочно завершает работу цикла; программа начинает выполнять ближайшую инструкцию после тела цикла;

- continue — ведет себя так, как будто программа внезапно достигла конца тела; начинается следующий оборот, и условное выражение немедленно проверяется.

Оба эти слова являются ключевыми.

Теперь мы покажем вам два простых примера, чтобы проиллюстрировать, как работают эти две инструкции. Посмотрите на код ниже. Запустите программу и проанализируйте вывод. Модифицируйте код и экспериментируйте.

```
# break - пример
print("Инструкция break:")
for i in range(1, 6):
    if I == 3:
        break
    print("Внутри цикла.", i)
print("Вне цикла.")

# continue - пример
print("\nИнструкция contivue:")
for i in range(1, 6):
    if i == 3:
        continue
    print("Внутри цикла.", i)
print("Вне цикла.")
```

Операторы break и continue: примеры

Вернемся к нашей программе, которая распознает наибольшие из введенных чисел. Мы конвертируем ее дважды, используя инструкции break и continue.

Проанализируйте код и определите, будете ли вы использовать оба варианта и как.

Вариант break:

```
largestNumber = -99999999
counter = 0
while True:
    number = int(input("Введите число и напишите -1
             для завершения программы: "))
    if number == -1:
        break
    counter += 1
    if number > largestNumber:
        largestNumber = number
if counter != 0:
    print("Наибольшее число", largestNumber)
else:
    print("Вы не ввели число.")
```

Запустите код, протестируйте и экспериментируйте с ним.

А теперь вариант continue:

```
largestNumber = -99999999
counter = 0
number = int(input("Введите число и напишите -1
         для завершения программы: "))
while number != -1:
    if number == -1:
        continue
    counter += 1
    if number > largestNumber:
        largestNumber = number
    number = int(input("Введите число и напишите -1
             для завершения программы: "))
if counter:
    print("Наиболшее число", largestNumber)
else:
    print("Вы не ввели число.")
```

Опять же — запустите, протестируйте и поэкспериментируйте.

Цикл while и ветвь else

Оба цикла, while и for, имеют одну интересную (и редко используемую) особенность.

Мы покажем вам, как она работает — постарайтесь решить сами, пригодно ли она для использования и можете ли вы жить без нее.

Другими словами, попытайтесь убедить себя, является ли эта функция ценной и полезной или просто синтаксическим сахаром.

Посмотрите на фрагмент ниже.

```
i = 1
while i < 5:
    print(i)
    i += 1
else:
    print("else:", 1)
```

В конце есть что-то странное — ключевое слово else.

Как вы, возможно, подозревали, у циклов тоже может быть ветвь else, как и у if.

Ветвь else цикла всегда выполняется один раз, независимо от того, вошел ли цикл в свое тело или нет.

Можете ли вы угадать вывод? Запустите программу, чтобы проверить, были ли вы правы.

Немного измените фрагмент кода, чтобы у цикла не было возможности выполнить свое тело хотя бы один раз:

```
i = 5
while i < 5:
    print(i)
    i += 1
else:
    print("else:", i)
```

Условие while является ложным False в начале — вы это видите?

Запустите и протестируйте программу и проверьте, была ли выполнена ветвь else.

Циклы for ведут себя немного по-другому — взгляните на фрагмент ниже и запустите его.

```
for i in range(5):
    print(i)
else:
    print("else:", i)
```

Вывод может быть немного удивительным.

Переменная i сохраняет свое последнее значение. Немного измените код, чтобы провести еще один эксперимент.

```
i = 111
for i in range(2, 1):
    print(i)
else:
    print("else:", i)
```

Можете ли вы угадать результат?

Тело цикла здесь не будет выполнено вообще.

Примечание: *мы назначили переменную i перед циклом.*

Запустите программу и проверьте ее результат.

Когда тело цикла не выполняется, переменная управления сохраняет значение, которое было до цикла.

Примечание: *если управляющая переменная не существует до начала цикла, она не будет существовать, когда выполнение достигнет ветви else.*

Как вы относитесь к этому варианту else?

Теперь мы расскажем вам о некоторых других типах переменных. Наши текущие переменные могут хранить только одно значение за раз, но есть переменные, которые могут делать гораздо больше — они могут хранить столько значений, сколько вы хотите.

Основные моменты

1. В Python есть два типа циклов: while и for:
   - цикл while выполняет оператор или множество операторов, пока заданное логическое условие истинно (true), например:

```
# Пример 1
while True:
    print("Застрял в бесконечном цикле.")

# Пример 2
counter = 5
while counter > 2:
    print(counter)
    counter -= 1
```

- цикл for выполняет множество операторов много раз; он используется для перебора последовательности (например, списка, словаря, набора чисел или множества — вы скоро узнаете о них) или других итерируемых объектов (например, строк). Вы можете использовать цикл for для перебора последовательности чисел, используя встроенную функцию range. Посмотрите на примеры ниже:

```
# Пример 1
word = "Python"
for letter in word:
    print(letter, end="*")

# Пример 2
for i in range(1, 10):
    if i % 2 == 0:
        print(i)
```

2. Вы можете использовать операторы break и continue для изменения потока выполнения цикла:

- Вы используете break для выхода из цикла, например:

```
text = "OpenEDG Python Institute"

for letter in text:
    if letter == "P":
        break
    print(letter, end="")
```

- Вы используете continue, чтобы пропустить текущую итерацию и продолжить следующую итерацию, например:

```
text = "pyxpyxpyx"
for letter in text:
if letter == "x":
continue
print(letter, end="")
```

3. Циклы while и for также могут содержать выражение else в Python. Выражение else выполняется после того, как цикл завершает свое выполнение до тех пор, пока он не был прерван break, например:

```
n = 0
while n != 3:
    print(n)
    n += 1
else:
print(n, "else")
    print()
for i in range(0, 3):
    print(i)
else:
    print(i, "else")
```

4. Функция range() генерирует последовательность чисел. Она принимает целые числа и возвращает объекты диапазона. Синтаксис range() выглядит следующим образом: range(start, stop, step), где:

- start — необязательный параметр, задающий начальное число последовательности (по умолчанию 0)
- stop — необязательный параметр, указывающий конец сгенерированной последовательности (он не включен),

- и step — необязательный параметр, определяющий разницу между числами в последовательности (по умолчанию 1).

  Пример кода:

```
for i in range(3):
    print(i, end=" ") # выводит: 0 1 2
for i in range(6, 1, -2):
    print(i, end=" ") # выводит: 6, 4, 2
```

Логические и побитовые операции на языке Python

Компьютерная логика

Вы заметили, что условия, которые мы использовали до сих пор, были очень простыми? Условия, которые мы используем в реальной жизни, намного сложнее. Давайте посмотрим на это предложение: «если у нас будет немного свободного времени и погода будет хорошая, мы пойдем гулять».

Мы использовали союз «и» (and), что означает, что прогулка зависит от одновременного выполнения этих двух условий. На языке логики такая связь условий называется конъюнкцией. А теперь еще один пример:

Если ты будешь в торговом центре или я буду в торговом центре, кто-то из нас купит подарок для мамы.

Появление слова «или» (or) означает, что покупка зависит хотя бы от одного из этих условий. В логике такое соединение называется дизъюнкцией.

Понятно, что в Python должны быть операторы для построения конъюнкций и дизъюнкций. Без них выразительная сила языка была бы существенно ослаблена. Они называются логическими операторами.

and

Одним из логических операторов конъюнкции в Python является слово and. Это бинарный оператор с приоритетом ниже, чем у операторов сравнения. Это позволяет

нам программировать сложные условия без использования скобок, подобные этому:

```
counter > 0 and value == 100
```

Результат, который выдает оператор and, может быть определен на основе таблицы истинности.

Если мы рассмотрим конъюнкцию A and B, множество возможных значений аргументов и соответствующих значений конъюнкции выглядит следующим образом:

Аргумент A	Аргумент B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

or

Оператор дизъюнкции — это слово or. Это бинарный оператор с более низким приоритетом, чем and (так же, как + по сравнению с *). Его таблица истинности выглядит следующим образом:

Аргумент A	Аргумент B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

not

Кроме того, есть еще один оператор, который можно применять для построения условий. Это унарный опе-

ратор, выполняющий логическое отрицание. Его работа простая: он превращает истину в ложь, а ложь в истину.

Этот оператор записывается как слово not, и его приоритет очень высок: такой же, как для унарных + и -. Его таблица истинности проста:

Аргумент	not Аргумент
False	True
True	False

Логические выражения

Давайте создадим переменную с именем var и присвоим ей 1. Следующие условия попарно эквивалентны:

```
print(var > 0)
print(not (var <= 0))
print(var != 0)
print(not (var == 0))
```

Возможно, вы знакомы с законами де Моргана. Они говорят, что:

- Отрицание конъюнкции есть дизъюнкция отрицаний.
- Отрицание дизъюнкции есть конъюнкция отрицаний.

Давайте напишем то же самое, используя Python:

```
not (p and q) == (not p) or (not q)
not (p or q) == (not p) and (not q)
```

Обратите внимание, как круглые скобки использовались для написания выражений — мы размещаем их, чтобы улучшить читаемость.

Следует добавить, что ни один из этих операторов с двумя аргументами не может использоваться в сокращенной форме, известной как op=. Это исключение стоит запомнить.

Логические значения и одиночные биты

Логические операторы принимают свои аргументы целиком независимо от того, сколько битов они содержат. Операторы знают только значение: ноль (когда все биты сброшены) означает False; не ноль (если есть хотя бы один бит) означает True.

Результатом их операций является одно из следующих значений: False или True. Это означает, что этот фрагмент будет присваивать значение True переменной j, если i не равен нулю; в противном случае будет False.

```
i = 1
j = not not i
```

Побитовые операторы

Однако есть четыре оператора, которые позволяют вам управлять одиночными битами данных. Они называются побитовыми операторами.

Они охватывают все операции, которые мы упоминали ранее в логическом контексте, и один дополнительный оператор. Это оператор xor (как исключающее или) и обозначается как ^ (карет).

Вот все из них:

- & (*амперсанд*) — побитовая конъюнкция;
- | (*вертикальная черта*) — побитовая дизъюнкция;

- ~ (*тильда*) — побитовое отрицание;
- ^ (*карет*) — побитовое исключающее или (xor).

Побитовые операции (&, |, и ^)

Arg A	Arg B	Arg B & Arg B	Arg A | Arg B	Arg A ^ Arg B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Побитовые операции (~)

Arg	~Arg
0	1
1	0

Давайте разъясним проще:

- & требует ровно две 1, чтобы дать 1 как результат;
- | требует по крайней мере одну 1, чтобы дать 1 как результат;
- ^ требует ровно одну 1, чтобы дать 1 как результат;

Добавим важное замечание: аргументы этих операторов должны быть целыми числами; мы не должны использовать здесь плавающие запятые.

Важна разница в работе логических и побитовых операторов: логические операторы не проникают в битовый уровень своего аргумента. Их интересует только конечное целочисленное значение.

Побитовые операторы строже: они работают с каждым битом отдельно. Если предположить, что целочис-

ленная переменная занимает 64 бита (что является обычным в современных компьютерных системах), вы можете представить себе побитовую операцию как 64-кратную оценку логического оператора для каждой пары битов аргументов. Эта аналогия явно несовершенна, так как в реальном мире все эти 64 операции выполняются одновременно.

Логические и побитовые операции: продолжение

Сейчас мы покажем вам пример различий в операциях между логическими и побитовыми операциями. Предположим, что были выполнены следующие присваивания:

```
i = 15
j = 22
```

Если мы предположим, что целые числа хранятся в 32 битах, побитовое изображение двух переменных будет следующим:

```
i: 00000000000000000000000000001111
j: 00000000000000000000000000010110
```

Дано присваивание:

```
log = i and j
```

Мы имеем дело здесь с логической конъюнкцией. Давайте проследим ход расчетов. Обе переменные i и j не являются нулями, поэтому считается, что они представ-

ляют истину (True). Изучив таблицу истинности для оператора and, мы увидим, что результат будет True. Другие операции не выполняются.

```
log: True
```

Теперь побитовая операция — вот она:

```
bit = i & j
```

Оператор & будет работать с каждой парой соответствующих битов в отдельности, получая значения соответствующих битов результата. Следовательно, результат будет следующим:

i	00000000000000000000000000001111
j	00000000000000000000000000010110
bit = i & j	00000000000000000000000000000110

Эти биты соответствуют целочисленному значению шести.

Давайте теперь посмотрим на операторы отрицания. Сначала логический:

```
logneg = not i
```

Переменная logneg будет установлена в False — больше ничего делать не нужно.

Побитовое отрицание выглядит так:

```
bitneg = ~i
```

Интересный факт: значение переменной bitneg равно -16. Это может показаться странным, но это не так. Если вы хотите узнать больше, вы должны проверить двоич-

ную систему счисления и правила, управляющие числом в дополнительном двоичном коде.

```
i                   00000000000000000000000000001111
bitneg = ~i         11111111111111111111111111110000
```

Каждый из этих операторов с двумя аргументами может использоваться в сокращенной форме. Вот примеры их эквивалентных нотаций:

`x = x & y`	x &= y
`x = x | y`	x |= y
`x = x ^ y`	x ^= y

Как мы работаем с одиночными битами?

Теперь мы покажем вам, для чего вы можете использовать побитовые операторы. Представьте, что вы разработчик и обязаны написать важную часть операционной системы. Вам сказали, что вы можете использовать переменную, назначенную следующим образом:

```
flagRegister = 0x1234
```

Переменная хранит информацию о различных аспектах работы системы. Каждый бит переменной хранит одно значение да/нет. Вам также сказали, что только один из этих битов ваш — третий (помните, что биты нумеруются с нуля, а бит номер ноль — младший, а самый старший — номер 31). Остальные биты не могут изменяться, потому что они предназначены для хранения других данных. Вот ваш бит, помеченный буквой x:

```
flagRegister = 00000000000000000000000000x000
```

Вы можете столкнуться со следующими задачами:

1. **Проверьте состояние вашего бита** — вы хотите узнать значение вашего бита; сравнение всей переменной с нулем ничего не даст, потому что оставшиеся биты могут иметь совершенно непредсказуемые значения, но вы можете использовать следующее свойство конъюнкции:

```
x & 1 = x
x & 0 = 0
```

Если вы примените операцию & к переменной flagRegister вместе со следующим битовым изображением:

```
00000000000000000000000000001000
```

(обратите внимание на 1 в позиции вашего бита), в результате вы получите одну из следующих битовых строк:

- 00000000000000000000000000001000 если ваш бит был установлен в 1
- 00000000000000000000000000000000 если ваш бит был сброшен на 0

Такая последовательность нулей и единиц, задачей которой является получение значения или изменение выбранных битов, называется битовой маской.

Давайте создадим битовую маску, чтобы определить состояние вашего бита. Это должно указывать на третий бит. Этот бит имеет вес 23 = 8. Подходящую маску можно создать с помощью следующего объявления:

```
theMask = 8
```

Вы также можете сделать последовательность инструкций в зависимости от состояния вашего бита:

```
if flagRegister & theMask:
    # мой бит установлен
else:
    # мой бит сброшен
```

2. **Сбросьте свой бит** — вы назначаете ноль биту, в то время как все остальные биты должны оставаться неизменными. Давайте использовать то же свойство конъюнкции, что и раньше, но давайте использовать немного другую маску — как показано ниже:

```
11111111111111111111111111110111
```

Обратите внимание, что маска была создана в результате отрицания всех битов переменной theMask. Сбросить бит просто и это выглядит так (выберите тот, который вам нравится больше):

```
flagRegister = flagRegister & ~theMask

flagregister &= ~theMask
```

3. **Установите свой бит** — вы назначаете 1 для вашего бита, в то время как все остальные биты должны оставаться неизменными. Используйте следующее свойство дизъюнкции:

```
x | 1 = 1
x | 0 = x
```

Теперь вы готовы установить свой бит с помощью одной из следующих инструкций:

```
flagRegister = flagRegister | theMask
flagRegister |= theMask
```

4. **Отрицайте свой бит** — вы заменяете 1 на 0, а 0 на 1. Вы можете использовать интересное свойство оператора xor:

```
x ^ 1 = ~x
x ^ 0 = x
```

и отрицать свой бит со следующими инструкциями:

```
flagRegister = flagRegister ^ theMask
flagRegister ^= theMask
```

Двоичный сдвиг влево и двоичный сдвиг вправо

Python предлагает еще одну операцию, связанную с одиночными битами: сдвиг. Она применяется только к целочисленным значениям, и вы не должны использовать плавающие запятые как аргументы.

Вы уже применяете эту операцию очень часто и совершенно неосознанно. Как вы умножаете любое число на десять? Посмотрите:

```
12345 × 10 = 123450
```

Как вы можете видеть, умножение на десять — это сдвиг всех цифр влево и заполнение полученного пробела нулем.

Деление на десять? Взгляните:

```
12340 ÷ 10 = 1234
```

Деление на десять — это не что иное, как смещение цифр вправо. Подобная операция выполняется компьютером, но с одним отличием: поскольку два является базой для двоичных чисел (не 10), сдвиг значения на один бит влево соответствует умножению его на два; соответственно, сдвиг одного бита вправо подобен делению на два (обратите внимание, что самый правый бит потерян).

Операторы сдвига в Python — это пара орграфов: << и >>, четко указывающих, в каком направлении будет действовать сдвиг.

```
value << bits
value >> bits
```

Левый аргумент этих операторов является целочисленным значением, биты которого сдвинуты. Правильный аргумент определяет размер сдвига.

Это показывает, что эта операция не является коммутативной.

Приоритет этих операторов очень высок. Вы увидите их в обновленной таблице приоритетов, которую мы покажем вам в конце этого раздела.

Посмотрите на сдвиги в коде ниже.

```
var = 17
varRight = var >> 1
varLeft = var << 2
print (var, varLeft, varRight)
```

Последний вызов print() создает следующий вывод:

```
17 68 8
```

Примечание:

- *17 // 2 → 8 (смещение вправо на один бит соответствует целочисленному делению на два);*
- *17 * 4 → 68 (сдвиг влево на два бита аналогичен целочисленному умножению на четыре).*

А вот обновленная таблица приоритетов, содержащая все представленные операторы:

Приоритет	Оператор	Вид
1	~, +, -	унарный
2	**	
3	*, /, //, %	
4	+, -	бинарный
5	<<, >>	
6	<, <=, >, >=	
7	==, !=	
8	&	
9	|	
10	=, +=, -=, *=, /=, %=, &=, ^=, |=, >>=, <<=	

Ключевые моменты

1. **Python поддерживает следующие логические операторы:**

- and → если оба операнда истинны, условие истинно, например, (True and True) это True,
- or → если какой-либо из операндов истинен, условие истинно, например, (True or False) это True,

- not → возвращает false, если результат равен true, и возвращает true, если результат равен false, например, not True это False.

2. **Вы можете использовать побитовые операторы для управления отдельными битами данных. Следующий пример данных:**

- x = 15, что означает 0000 1111 в двоичном виде,
- y = 16, что означает 0001 0000 в двоичном виде.

будет использоваться для иллюстрации значения побитовых операторов в Python. Проанализируйте примеры ниже:

- & выполняет побитовый and, например, x & y = 0, что означает 0000 0000 в двоичном виде,
- | выполняет побитовый or, например, x | y = 31, что означает 0001 1111 в двоичном виде,
- ~ выполняет побитовый not, например, ~ x = 240, что означает 1111 0000 в двоичном виде,
- ^ выполняет побитовый xor, например, x ^ y = 31, что означает 0001 1111 в двоичном виде,
- >> выполняет побитовый сдвиг вправо, например y >> 1 = 8, что означает 0000 1000 в двоичном виде,
- << выполняет побитовый сдвиг влево, например y << 3 =, что означает 1000 0000 в двоичном виде.

Урок 3.

Циклы в языке Python.
Логические и побитовые операции

© STEP IT Academy, www.itstep.org

All rights to protected pictures, audio, and video belong to their authors or legal owners.
Fragments of works are used exclusively in illustration purposes to the extent justified by the purpose as part of an educational process and for educational purposes in accordance with Article 1273 Sec. 4 of the Civil Code of the Russian Federation and Articles 21 and 23 of the Law of Ukraine "On Copyright and Related Rights". The extent and method of cited works are in conformity with the standards, do not conflict with a normal exploitation of the work, and do not prejudice the legitimate interests of the authors and rightholders. Cited fragments of works can be replaced with alternative, non-protected analogs, and as such correspond the criteria of fair use.

All rights reserved. Any reproduction, in whole or in part, is prohibited. Agreement of the use of works and their fragments is carried out with the authors and other right owners. Materials from this document can be used only with resource link.

Liability for unauthorized copying and commercial use of materials is defined according to the current legislation of Ukraine.