Битовые операции И, ИЛИ, НЕ, XOR. Сдвиговые операторы

Курс по Python: https://stepik.org/course/100707

Смотреть материал на видео

На этом занятии речь пойдет о битовых операциях над целыми числами. Начнем с самой простой битовой операции НЕ, которая выполняет инверсию бит в соответствии с такой табличкой:

x	НЕ
0	1
1	0

Например, возьмем целое число:

```
a = 121
```

Чтобы посмотреть его битовое (двоичное) представление, можно воспользоваться функцией bin():

```
bin(a)
```

На выходе будет строка с битами этого десятичного числа. А теперь выполним инверсию его бит. Для этого в Python используется оператор ~ (тильда), которая записывается перед числом или переменной:

```
b = ~a
```

В результате переменная b стала отрицательной и принимает значение -122. Почему значение стало отрицательным и уменьшилось на -1? Смотрите, любое число кодируется набором бит. В Python – это довольно длинная последовательность:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0

Самый старший бит отвечает за знак числа: 0 – положительное; 1 – отрицательное. Так вот, если взять десятичное число 0:

```
d = 0
```

и инвертировать все его биты:

```
~d
```

то мы получим число, состоящее из всех битовых единиц, а это есть не что иное, как десятичное значение -1. Мы здесь наглядно видим, как операция отрицания превращает число 0 в отрицательное и уменьшает его на -1. Так происходит со всеми целыми положительными числами:

```
d = 10
~d
```

И с отрицательными:

```
d = -10
~d
```

За счет инвертирования всех бит числа.

Битовая операция И

Следующая битовая операция И применяется уже к двум операндам (переменным или числам) со следующей таблицей истинности:

x1	x2	И

0	0	0
0	1	0
1	0	0
1	1	1

и реализуется через оператор & (амперсанд). Например, представим, что у нас есть два числа (две переменные):

```
flags = 5
mask = 4
```

Выполним для них битовую операцию И:

```
res = flags & mask
```

Переменная res ссылается на значение 4. Давайте посмотрим, почему получился такой результат. В соответствии с таблицей истинности результирующие биты будут следующими (перечислить). В итоге, переменная res получается равной 4.

flags	0	0	0	0	0	1	0	1
mask	0	0	0	0	0	1	0	0
res	0	0	0	0	0	1	0	0

Ну хорошо, разобрались, но зачем все это надо? Смотрите, если нам нужно проверить включен ли какой-либо бит числа (то есть установлен в 1), то мы можем это сделать с помощью битовой операции И, следующим образом:

```
if flags & mask == mask:
        print("Включен 2-й бит числа"
else:
        print("2-й бит выключен")
```

То есть, из-за использования операции И мы в переменной flags обнуляем все биты, кроме тех, что совпадают с включенными битами в переменной mask. Так как mask содержит только один включенный бит – второй, то именно проверку на включенность этого бита мы и делаем в данном случае.

Чтобы убедиться, что все работает, присвоим переменной flags значение 1:

```
flags = 1
```

и запустим программу. Теперь, видим сообщение, что 2-й бит выключен, что, в общем то, верно. Вот так, с помощью битовой операции И можно проверять включены ли определенные биты в переменных.

Также битовую операцию И используют для выключения определенных битов числа. Делается это, следующим образом:

```
flags = 13
mask = 5
flags = flags & ~mask
```

Что происходит здесь? Сначала выполняется инверсия бит маски, так как операция НЕ имеет более высокий приоритет, чем операция И. Получаем, что маска состоит из вот таких бит. Затем, идет операция поразрядное И, и там где в маске стоят 1 биты переменной flags не меняются, остаются прежними, а там где стоят в маске 0 – эти биты в переменной flags тоже становятся равными 0. За счет этого происходит выключение 2-го и 0-го битов переменной flags.

Кстати, последнюю строчку можно переписать и короче:

```
flags &= ~mask
```

Это будет одно и то же.

Битовая операция ИЛИ

Следующая битовая операция ИЛИ определяется оператором | и ее таблица истинности выглядит следующим образом:

x1	x2	ИЛИ
0	0	0
0	1	1
1	0	1
1	1	1

Обычно ее применяют, когда нужно включить отдельные биты переменной. Рассмотрим такую программу:

```
flags = 8
mask = 5
flags = flags | mask
```

Операция поразрядное ИЛИ, как бы собирает все единички из обеих переменных и получается такое своеобразное сложение (объединение).

flags	0	0	0	0	1	0	0	0
mask	0	0	0	0	0	1	0	0
flags	0	0	0	0	1	1	0	1

Кстати, в этом случае действительно получилось 8+5=13. Но это будет не всегда так, например, если:

```
flags = 9
flags |= mask
```

то результат тоже будет 13, так как операция ИЛИ включает бит вне зависимости был ли он уже включен или нет, все равно на выходе будет единица.

Битовая операция исключающее ИЛИ (XOR)

И последняя базовая операция работы с битами – исключающее ИЛИ (ее еще называют XOR). Она задается оператором ^ и имеет такую таблицу истинности:

x1	x2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Из нее видно, что данная операция позволяет переключать биты числа, то есть, если они были равны 0, то станут 1 и, наоборот, если были 1 – станут 0. Продемонстрируем это на таком примере:

```
flags = 9
mask = 1
flags = flags ^ mask
```

Здесь сначала получим значение 8, так как 0-й бит будет выключен.

flags	0	0	0	0	1	0	0	1
mask	0	0	0	0	0	0	0	1
flags	0	0	0	0	1	0	0	0

Но если повторить последнюю строчку:

```
flags ^= mask
```

то нулевой бит снова включится и переменная flags примет прежнее значение 9. Вот так, с помощью операции XOR можно переключать отдельные биты переменных.

Интересной особенностью операции XOR является отсутствие потерь данных при ее работе. Что это значит? Смотрите, какую бы маску мы не взяли, дважды примененная маска дает исходное число:

```
flags = 9
mask = 111
flags ^= mask
flags ^= mask
```

Этот эффект часто используют для шифрования информации. Например, так работает пароль архиватора zip. В нем пароль – это маска, которая накладывается на заархивированные данные. Чтобы восстановить их, необходимо ввести пароль и повторить наложение этой маски.

Вот мы с вами рассмотрели основные битовые операции, и обратите внимание, что все их можно записывать в таком виде:

Полная форма	Краткая форма	Приоритет
a = a & b	a &= b	2 (И)
a = a | b	a |= b	1 (ИЛИ)
a = a ^ b	a ^= b	1 (XOR)

~a		3 (НЕ)

Операторы смещения бит

И в заключение занятия рассмотрим еще два распространенных битовых оператора:

>> смещение бит вправо
<< смещение бит влево

Предположим, у нас имеется переменная:

```
x = 160
```

Ее битовое представление:

```
bin(x) # 10100000
```

Давайте сдвинем все эти биты вправо на один бит. Сделать это можно, следующим образом:

```
x = x >> 1
```

Теперь битовое представление числа:

```
bin(x) # 1010000
```

А значение равно 80. То есть, сдвигая биты вправо на один бит, мы разделили число на 2. Если сдвинуть на два бита вправо:

```
x = x >> 2
```

то это уже эквивалентно делению на 4. Давайте еще раз сдвинем на бит вправо:

```
x = x >> 1
```

будет пять, но 5 не делится на 2 нацело. Что же получится при смещении бит вправо? Смотрим:

```
x = x >> 1
```

Получаем значение 1. То есть, здесь, как бы дробная часть была отброшена. Действительно, один из битов был просто потерян, это и объясняет данный эффект.

По аналогии можно делать смещения влево:

```
x = x << 1
```

Это уже будет эквивалентно умножению на два. Или, так:

```
x = x << 3
```

Это эквивалентно умножению на 8 (два в кубе). И так далее. Операции смещения бит влево и вправо выполняют целочисленное умножение и деление кратное двум. Причем эти операции умножения и деления работают значительно быстрее, чем традиционные арифметические операции умножения и деления. Поэтому, разработчики различных алгоритмов для маломощных компьютеров стараются составить вычисления так, чтобы они базировались на сдвиговых операциях, исключая прямое умножение и деление.

Надеюсь, из этого занятия вы поняли, как работают битовые операции И, ИЛИ, НЕ, XOR, а также битовые операции сдвигов.

> Курс по Python: https://stepik.org/course/100707

Видео по теме

#1 Первое знакомство с Python Установка на компьютер

#2. Варианты исполнени команд. Переходим в PyCharm

← Предыдущая

Следующая →