Ch02-2-BitwiseOperators

September 10, 2025

1 2. Bitwise Operators

- https://wiki.python.org/moin/BitwiseOperators

1.1 Topics

- Number systems
- Binary representation of positive integers
- Twos Complement for Negative Integers
- Bitwise operators
- Examples

1.2 Number systems

- there are several number systems based on the base
  - base is number of unique digits number system uses to represent numbers
- binary (base 2), octal (base 8), decimal (base 10), hexadecimal (base 16), etc.

1.2.1 Decimal number system

- also called Hindu-Arabic number system
- most commonly used number system that uses base 10
  - has 10 digits or numerals to represent numbers: 0..9
  - e.g. 1, 79, 1024, 12345, etc.
- numerals representing numbers have different place values depending on position:
  - ones (10^0), tens(10^1), hundreds(10^2), thousands(10^3), ten thousands(10^4), etc.
  - e.g. $543.21 = (5 \times 10^2) + (4 \times 10^1) + (3 \times 10^0) + (2 \times 10^{-1}) + (1 \times 10^{-2})$

1.2.2 Binary number system

- digital computers work with binary number system
- decimal number system and any text and symbols must be converted into binary for the computer systems to process, store and transmit
- typically, programming language like C/C++ uses 32-bit or 64-bit depending on the architecture of the system to represent binary numbers
- Python however uses "INFINITE" number of bits to represent Integers in binary

1.3 Number system conversion

- since computers understand only binary, everything (data, code) must be converted into binary
- all characaters (alphabets and symbols) are given decimal codes for electronic communication
  - these codes are called ASCII (American Standard Code for Information Interchange)
  - A -> 65; Z -> 90; a -> 97; z -> 122, * -> 42, etc.
  - see ASCII chart: https://en.cppreference.com/w/c/language/ascii

1.3.1 Converting decmial to binary number

- Twos-complement for positive integers
- algorithm steps:
  1. repeteadly divide the decimal number by base 2 until the quotient becomes 0
     - note remainder for each division
  2. collect all the remainders in reverse order
     - the first remainder is the last (least significant) digit in binary
- example 1: what is decimal $(10)_{10}$ in binary $(?)_2$?
  - step 1:

```
10 / 2 : quotient: 5, remainder: 0
5 / 2  : quotient 2, remainder: 1
2 / 2  : quotient: 1, remainder: 0
1 / 2  : quotient: 0, remainder: 1
```

  - step 2:
    * collect remainders from bottom up: 1010
  - so, $(10)_{10} = (1010)_2$
- example 2: what is decimal $(13)_{10}$ in $(?)_2$?
  - step 1:

```
13 / 2 : quotient: 6, remainder: 1
6 / 2  : quotient 3, remainder: 0
3 / 2  : quotient: 1, remainder: 1
1 / 2  : quotient: 0, remainder: 1
```

  - step 2:
    * collect remainders from bottom up: 1101
  - so, $(13)_{10} = (1101)_2$

1.3.2 Converting binary to decimal number

- once the computer does the computation in binary, it needs to convert the results back to decimal number system for humans to understand
- algorithm steps:
  1. multiply each binary digit by its place value in binary

2. sum all the products

- example 1: what is binary $(1010)_2$ in decimal $(?)_{10}$?
  - step 1:
    * $0 \times 2^0 = 0$
    * $1 \times 2^1 = 2$
    * $0 \times 2^2 = 0$
    * $1 \times 2^3 = 8$
  - step 2:
    * $0 + 2 + 0 + 8 = 10$
  - so, $(1010)_2 = (10)_{10}$
- example 2: what is binary $(1101)_2$ in decimal $(?)_{10}$?
  - step 1:
    * $1 \times 2^0 = 1$
    * $0 \times 2^1 = 0$
    * $1 \times 2^2 = 4$
    * $1 \times 2^3 = 8$
  - step 2:
    * $1 + 0 + 4 + 8 = 13$
  - so, $(1101)_2 = (13)_{10}$
- we got the same decimal vales we started from in previous examples

1.4 Twos Complement for Negative (signed) integers

- most common method of storing negative numbers on computers is a mathematical operation called Two's complement
- Two's complement of an N-bit number is defined as its complement with respect to 2^N
  - the sum of a number and its two's complement is 2^N
- e.g.: for the 3-bit binary number 010_2, the two's complement is 110_2, because $010_2 + 110_2 = 1000_2 = 2^3_{10}$
- Two's complement of N-bit number can be found by flipping each bit and adding one to it
- e.g. Find two's complement of 010 (3-bit integer)
- Algorithm steps:
  1. flipped each bit; 0 is flipped to 1 and 1 flipped to 0
     ```
     010 -> 101
     ```
  2. add 1 to the flipped binary

     ```
      101
       +1
     -----
      110
     ```

Example 2 - Represent decimal -13 using 32-bit binary - first find the binary of 13 and use Twos complement for negative integers

[3]:
```
# built-in bin function converts integers into binary
bin(13)
# 00000000000000000000000000001101 - 32-bit
```

[3]: '0b1101'

```
[4]: # Python uses -ve sign to represent -ve binary also
bin(-13)
```

[4]: '-0b1101'

1.4.1 Two's complement of -13 with 32-bit is

```
- 13 ->           00000000000000000000000000001101
- flip every bit: 11111111111111111111111111110010
                                                +1
        -------------------------------------------
                  11111111111111111111111111110011
```

1.4.2 bitwise operators

- https://wiki.python.org/moin/BitwiseOperators
- bitwise operators work on binary numbers (bits)
  - integers are implicitly converted into binary and then bitwise operations are applied
- bitwise operations are used in lower-level programming such as device drivers, low-level graphics, communications protocol packet assembly, encoding and decoding data, encryption technologies, etc.
- a lot of integer arithmetic computations can be carried our much more efficiently using bitwise operations

Operator	Symbol	Symbol Name	Syntax	Operation
bitwise left shift	«	left angular bracket	x « y	all bits in x shifted left y bits; multiplication by 2^y
bitwise right shift	»	right angular bracket	x » y	all bits in x shifted right y bits; division by 2^y
bitwise NOT	~	tilde	~x	all bits in x flipped
bitwise AND	&	ampersand	x & y	each bit in x AND each bit in y
bitwise OR	|	pipe	x | y	each bit in x OR each bit in y

Operator	Symbol	Symbol Name	Syntax	Operation
bitwise XOR	^	caret	x ^ y	each bit in x XOR each bit in y

1.4.3 table for bitwise operations

& - bitwise AND

x	y	x & y
1	1	1
1	0	0
0	1	0
0	0	0

| - bitwise OR

x	y	x | y
1	1	1
1	0	1
0	1	1
0	0	0

~ - bitwise NOT

x	~x
1	0
0	1

^ - bitwise XOR

x	y	x ^ y
1	1	0
1	0	1
0	1	1
0	0	0

bitwise left shift examples

[5]:
```
# convert 1 decimal to binary and shift left by 4 bits
1 << 4 # same as 1*2*2*2*2; result is in decimal
```

[5]: 16

Explanation

- Note: in the following examples, binary uses 32-bit to represent decmial
- $1_{10} = 00000000000000000000000000000001_2$
- $1 << 4 = 000000000000000000000000000010000 = 2^4 = 16_{10}$

```
[6]: 3 << 4 # same as 3*2*2*2*2 or 3*2^4
```

[6]: 48

Explanation

- $3_{10} = 00000000000000000000000000000011_2$
- $3 << 4 = 00000000000000000000000000110000_2 = 2^5 + 2^4 = 32 + 16 = 48_{10}$

Bitwise right shit examples

```
[7]: 1024 >> 10 # same as 1024/2/2/2/2/2/2/2/2/2/2
```

[7]: 1

Explanation

- $1024_{10} = 00000000000000000000010000000000_2$
- $1024 >> 10 = 00000000000000000000000000000001_2 = 2^0 = 1_{10}$

Bitwise NOT examples

```
[8]: ~0 # result shown is in decimal!
```

[8]: -1

```
[9]: ~1 # Note: 1 in binary using 32-bit width (31 0s and 1) 00000....1

#result shown is in decimal
```

[9]: -2

Explanation

- $0_{10} = 0000000000000000000000000000000_2$
- $1_{10} = 0000000000000000000000000000001_2$
- $-1_{10} = 1111111111111111111111111111111_2$
- $2_{10} = 0000000000000000000000000000010_2$
- $-2_{10} = 1111111111111111111111111111110_2$
- Note: -ve numbers are stored in Two's complement
    - 2's complement is calculated by flipping each bit and adding 1 to the binary of positive integer

Bitwise AND examples

[10]:
```
1 & 1
```

[10]: 1

[11]:
```
1 & 0
```

[11]: 0

[12]:
```
0 & 1
```

[12]: 0

[13]:
```
0 & 0
```

[13]: 0

1.4.4 Bitwise OR examples

[14]:
```
1 | 1
```

[14]: 1

[15]:
```
1 | 0
```

[15]: 1

[16]:
```
0 | 1
```

[16]: 1

[19]:
```
0 | 0
```

[19]: 0

1.4.5 Bitwise XOR examples

[20]:
```
1 ^ 1
```

[20]: 0

[21]:
```
1 ^ 0
```

[21]: 1

[22]:
```
0 ^ 1
```

[22]: 1

[24]:
```
0 ^ 0
```

[24]: 0

[ ]:
```

```