**Just a Little BIT**

Welcome to an intro level explanation of bitwise operations in Python!

Bitwise operations might seem a little esoteric and tricky at first, but you'll get the hang of them pretty quickly.

*Bitwise operations* are operations that directly manipulate *bits*. In all computers, numbers are represented with bits, a series of zeros and ones. In fact, pretty much everything in a computer is represented by bits. This course will introduce you to the basic bitwise operations and then show you what you can do with them.

Bitwise operators often tend to puzzle and mystify new programmers, so don't worry if you are a little bit confused at first. To be honest, you aren't really going to see bitwise operators in everyday programming. However, they do pop up from time to time, and when they do, you should have a general idea of what is going on.

|  |
| --- |
| print 5 >> 4 # Right Shift  print 5 << 1 # Left Shift  print 8 & 5 # Bitwise AND  print 9 | 4 # Bitwise OR  print 12 ^ 42 # Bitwise XOR  print ~88 # Bitwise NOT |
| 0  10  0  13  38  -89 |

# Lesson I0: The Base 2 Number System

When we count, we usually do it in base 10. That means that each place in a number can hold one of ten values, 0-9. In binary we count in base two, where each place can hold one of two values: 0 or 1. The counting pattern is the same as in base 10 except when you carry over to a new column, you have to carry over every time a place goes higher than one (as opposed to higher than 9 in base 10).

For example, the numbers one and zero are the same in base 10 and base 2. But in base 2, once you get to the number 2 you have to carry over the one, resulting in the representation "10". Adding one again results in "11" (3) and adding one again results in "100" (4).

Contrary to counting in base 10, where each decimal place represents a power of 10, each place in a binary number represents a power of two (or a **bit**). The rightmost bit is the 1's bit (two to the zero power), the next bit is the 2's bit (two to the first), then 4, 8, 16, 32, and so on.

The binary number '1010' is 10 in base 2 because the 8's bit and the 2's bit are "on":

8's bit 4's bit 2's bit 1's bit 1 0 1 0 8 + 0 + 2 + 0 = 10

In Python, you can write numbers in binary format by starting the number with 0b. When doing so, the numbers can be operated on like any other number!

|  |
| --- |
| print 0b1, #1  print 0b10, #2  print 0b11, #3  print 0b100, #4  print 0b101, #5  print 0b110, #6  print 0b111 #7  print "\*\*\*\*\*\*"  print 0b1 + 0b11  print 0b11 \* 0b11 |
| 1 2 3 4 5 6 7  \*\*\*\*\*\*  4  9 |

# I Can Count to 1100!

All right! Time to practice counting in binary.

To make sure you've got the hang of it, fill out the rest of the numbers all the way up to twelve. Please **do not** use the str() method or any other outside functions.

Here are a few numbers that will be good to know going forward -

2 \*\* 0 = 1 2 \*\* 1 = 2 2 \*\* 2 = 4 2 \*\* 3 = 8 2 \*\* 4 = 16 2 \*\* 5 = 32 2 \*\* 6 = 64 2 \*\* 7 = 128 2 \*\* 8 = 256 2 \*\* 9 = 512 2 \*\* 10 = 1024

You may recognize these numbers. Do you have a 32 or 64 bit system? Does your computer have a 256GB hard drive? Computers think in binary!

|  |
| --- |
| one = 0b1  two = 0b10  three = 0b11  four = 0b100  five = 0b101  six = 0b110  seven = 0b111  eight = 0b1000  nine = 0b1001  ten = 0b1010  eleven = 0b1011  twelve = 0b1100 |

**The bin() Function**

Excellent! The biggest hurdle you have to jump over in order to understand bitwise operators is learning how to count in base 2. Hopefully the lesson should be easier for you from here on out.

There are Python functions that can aid you with bitwise operations. In order to print a number in its binary representation, you can use the bin() function. bin() takes an integer as input and returns the binary representation of that integer in a string. (Keep in mind that after using the binfunction, you can no longer operate on the value like a number.)

You can also represent numbers in base 8 and base 16 using the oct() and hex()functions. (We won't be dealing with those here, however.)

**Instructions**

**1.**

We've provided an example of the bin function in the editor. Go ahead and use print and bin() to print out the binary representations of the numbers 2 through 5, each on its own line.

|  |
| --- |
| print bin(1)  for i in range(2,6):  print bin(i) |
| 0b1  0b10  0b11  0b100  0b101 |

**int()'s Second Parameter**

Python has an int() function that you've seen a bit of already. It can turn non-integer input into an integer, like this:

int("42") # ==> 42

What you might not know is that the intfunction actually has an optional second parameter.

int("110", 2) # ==> 6

When given a string containing a number and the base that number is in, the function will return the value of that number converted to base ten.

**Instructions**

**1.**

In the console are several different ways that you can use the intfunction's second parameter.On line 7, use int to print the base 10 equivalent of the binary number 11001001.

Hint

Use the examples on lines 1 – 4 as a guide!

|  |
| --- |
| print int("1",2)  print int("10",2)  print int("111",2)  print int("0b100",2)  print int(bin(5),2)  # Print out the decimal equivalent of the binary 11001001.  print int("11001001",2) |
| 1  2  7  4  5  201 |

**Slide to the Left! Slide to the Right!**

The next two operations we are going to talk about are the left and right shift bitwise operators. These operators work by shifting the bits of a number over by a designated number of slots.

The block below shows how these operators work on the bit level. Note that in the diagram, the shift is always a positive integer:

# Left Bit Shift (<<) 0b000001 << 2 == 0b000100 (1 << 2 = 4) 0b000101 << 3 == 0b101000 (5 << 3 = 40) # Right Bit Shift (>>) 0b0010100 >> 3 == 0b000010 (20 >> 3 = 2) 0b0000010 >> 2 == 0b000000 (2 >> 2 = 0)

Shift operations are similar to rounding down after dividing and multiplying by 2 (respectively) for every time you shift, but it's often easier just to think of it as shifting all the 1s and 0s left or right by the specified number of slots.

Note that you can only do bitwise operations on an **integer**. Trying to do them on strings or floats will result in nonsensical output!

**Instructions**

**1.**

Shift the variable shift\_right to the right twice (>> 2) and shift the variable shift\_left to the left twice (<< 2). Try to guess what the printed output will be!

|  |
| --- |
| shift\_right = 0b1100  shift\_left = 0b1  # Your code here!  shift\_right = shift\_right >> 2  shift\_left = shift\_left << 2  print bin(shift\_right)  print bin(shift\_left) |
| 0b11  0b100 |

**A BIT of This AND That**

The bitwise AND (&) operator compares two numbers on a bit level and returns a number where the bits of that number are turned on if the corresponding bits of **both**numbers are 1. For example:

a: 00101010 42 b: 00001111 15 =================== a & b: 00001010 10

As you can see, the 2's bit and the 8's bit are the only bits that are on in both a and b, so a & b only contains those bits. Note that using the & operator can only result in a number that is less than or equal to the smaller of the two values.

So remember, for every given bit in a and b:

0 & 0 = 0 0 & 1 = 0 1 & 0 = 0 1 & 1 = 1

Therefore,

0b111 (7) & 0b1010 (10) = 0b10

which equals two.

**Instructions**

**1.**

print out the result of calling bin()on 0b1110 & 0b101.

See if you can guess what the output will be!

|  |
| --- |
| **print bin(0b1110 & 0b101)** |
| 0b100 |

**A BIT of This OR That**

The bitwise OR (|) operator compares two numbers on a bit level and returns a number where the bits of that number are turned on if **either** of the corresponding bits of either number are 1. For example:

a: 00101010 42 b: 00001111 15 ================ a | b: 00101111 47

Note that the bitwise | operator can only create results that are greater than or equal to the larger of the two integer inputs.

So remember, for every given bit in a and b:

0 | 0 = 0 0 | 1 = 1 1 | 0 = 1 1 | 1 = 1

Meaning

110 (6) | 1010 (10) = 1110 (14)

**Instructions**

**1.**

For practice, print out the result of using | on 0b1110 and 0b101 as a binary string. Try to do it on your own without using the | operator if you can help it.

Hint

This is pretty similar to what you did in the previous exercise! You're just using | instead of &.

|  |
| --- |
| **print bin(0b1110 | 0b101)** |
| 0b1111 |

**This XOR That?**

The XOR (^) or *exclusive or* operator compares two numbers on a bit level and returns a number where the bits of that number are turned on if **either** of the corresponding bits of the two numbers are 1, **but not both**.

a: 00101010 42 b: 00001111 15 ================ a ^ b: 00100101 37

Keep in mind that if a bit is off in both numbers, it stays off in the result. Note that XOR-ing a number with itself will always result in 0.

So remember, for every given bit in a and b:

0 ^ 0 = 0 0 ^ 1 = 1 1 ^ 0 = 1 1 ^ 1 = 0

Therefore:

111 (7) ^ 1010 (10) = 1101 (13)

**Instructions**

**1.**

For practice, print the result of using ^ on 0b1110 and 0b101 as a binary string. Try to do it on your own without using the ^ operator.

Hint

This is pretty similar to what you did in the previous exercise! You're just using ^ instead of |.

|  |
| --- |
| **print bin(0b1110 ^ 0b101 )** |
| 0b1011 |

# See? This is NOT That Hard!

The bitwise NOT operator (~) just flips all of the bits in a single number. What this actually means to the computer is actually very complicated, so we're not going to get into it. Just know that mathematically, this is equivalent to adding one to the number and then making it negative.

And with that, you've seen all of the basic bitwise operators! We'll see what we can do with these in the next section.

|  |
| --- |
| **print ~1**  **print ~2**  **print ~3**  **print ~42**  **print ~123** |
| -2  -3  -4  -43  -124 |

**The Man Behind the Bit Mask**

A *bit mask* is just a variable that aids you with bitwise operations. A bit mask can help you turn specific bits on, turn others off, or just collect data from an integer about which bits are on or off.

num = 0b1100 mask = 0b0100 desired = num & mask if desired > 0: print "Bit was on"

In the example above, we want to see if the third bit from the right is on.

1. First, we first create a variable numcontaining the number 12, or 0b1100.
2. Next, we create a mask with the third bit on.
3. Then, we use a bitwise-and operation to see if the third bit from the right of num is on.
4. If desired is greater than zero, then the third bit of num must have been one.

**Instructions**

**1.**

Define a function, check\_bit4, with one argument, input, an integer.

It should check to see if the fourth bit from the right is on.

If the bit is on, return "on" (not print!)

If the bit is off, return "off".

Check the Hint for some examples!

Hint

Here are some examples:

check\_bit4(0b1) # ==> "off" check\_bit4(0b11011) # ==> "on" check\_bit4(0b1010) # ==> "on"

You'll need to use a mask where all bits are off except for the fourth bit from the right.

|  |
| --- |
| def check\_bit4(input):  if 0b1000 & input > 0:  return "on"  else:  return "off" |

# Turn It On

You can also use masks to turn a bit in a number on using |. For example, let's say I want to make sure the rightmost bit of number a is turned on. I could do this:

a = 0b110 # 6 mask = 0b1 # 1 desired = a | mask # 0b111, or 7

Using the bitwise | operator will turn a corresponding bit on if it is off and leave it on if it is already on.

**Instructions**

**1.**

In the editor is a variable, a. Use a bitmask and the value a in order to achieve a result where the third bit from the right of a is turned on. Be sure to print your answer as a bin() string!

Hint

You should use | and the variable awith a mask where the third bit from the right, and only the third bit from the right, is on.

|  |
| --- |
| a = 0b10111011  print bin(a | 0b100) |
| 0b10111111 |

**Just Flip Out**

Using the XOR (^) operator is very useful for flipping bits. Using ^ on a bit with the number one will return a result where that bit is flipped.

For example, let's say I want to flip all of the bits in a. I might do this:

a = 0b110 # 6 mask = 0b111 # 7 desired = a ^ mask # 0b1

**Instructions**

**1.**

In the editor is the 8 bit variable a. Use a bitmask and the value a in order to achieve a result where all of the bits in aare flipped. Be sure to print your answer as a bin() string!

Hint

You'll need a mask the same length as a in which all of the bits are turned on (all set to 1).

|  |
| --- |
| a = 0b11101110  print bin(0b11111111 ^ a) |
| 0b10001 |

# Slip and Slide

Finally, you can also use the left shift (<<) and right shift (>>) operators to slide masks into place.

a = 0b101 # Tenth bit mask mask = (0b1 << 9) # One less than ten desired = a ^ mask

Let's say that I want to turn on the 10th bit from the right of the integer a.

Instead of writing out the entire number, we slide a bit over using the << operator.

We use 9 because we only need to slide the mask nine places over from the first bit to reach the tenth bit.

**Instructions**

**1.**

Define a function called flip\_bit that takes the inputs (number, n).

Flip the nth bit (with the ones bit being the first bit) and store it in result.

Return the result of calling bin(result).

Hint

Use the << operator to move your mask into place and the ^ operator to flip your desired bit.

|  |
| --- |
| def flip\_bit(number, n):  result = 0b1 << n-1  return bin(result ^ number) |