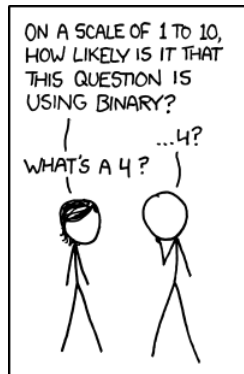


CS51 - Assignment 3

Bitwise Puzzles

Due: February 12th, at 11:59pm



<https://xkcd.com/953/>

For this assignment, we will:

- Become more comfortable with the binary representation of integers.
- Practice binary arithmetic by implementing various bitwise operations from scratch in Python.
- Practice writing Python code that is organized in functions.
- Practice testing your Python code.

You may (and we would strongly encourage you to) work with a partner on this assignment. If you do, you must *both be there* when either of you are working on the project and you should only be coding on *one computer* (e.g., don't divide up the puzzles). Since we'll start the assignment in lab, you should pair up with someone in your lab section (or

1 Background

The goal of this assignment is to increase your familiarity with the binary representation of information, specifically integer numbers. You are asked to complete a series of eight logical puzzles

that apply bitwise operations, that is, operations on the individual bits of the binary representation of integers. These puzzles will be solved in Python, but you will benefit from working on pen and paper first. You should also review the slides on Binary Arithmetic and Representing Information and De Morgan's Laws from the slides on Boolean Algebra.

2 Before you come to lab [1 point]

Read through this entire handout to make sure you understand what you will be implementing this week.

To help get you ready to tackle the puzzles, before you come to lab, you need to come up with one test case (on paper) for each of the eight puzzles. A test case must include the following information:

- A test input to the function and the expected answer (as `ints`).
- The binary representation of the test input and expected answer in two's complement.

For example, for the `bitwise_and` function, a test case could be:

- `bitwise_and(6,5) == 4`
- $6_{10} = 110_2$, $5_{10} = 101_2$, $4_{10} = 100_2$

Bring your test cases to the lab *on a piece of paper with your name on* that you will turn in. We'd encourage you to take a picture so that you have your test cases for testing your functions.

You may NOT use any of the test cases already provided in the `.py` file.

3 The Eight Puzzles

We have provided you with a file, `assignment3.py`, that provides skeleton code for the eight puzzles you will solve with your partner. Start by adding your names on the allocated space.

3.1 Bitwise operators in Python

Python supports six bitwise operators (`<<`, `>>`, `&`, `|`, `~`, and `^`) that are applied on the individual bits of a number written in two's complement binary.

Here is a detailed explanation for each of the bitwise operations that Python supports but don't forget to read the Quick Tips at the bottom of the section, too.

- `x << y`: Left shift. Note that because Python doesn't have a fixed number of bits for integers, we don't discard the first `y` bits. Instead, we fill out `y` zeros on the right. This is the same as multiplying `x` by `2**y`. For example, if `x=47`, then `x << 2 == 47 * (2**2) == 188`. If we see it in binary, that would be shifting the binary number `0101111` by adding two zeros to the right, resulting in `010111100`. Similarly for negative numbers, if `x=-47`, then `x << 2 == (-47) * (2**2) == -188`. If we see it in binary (two's complement), that would be shifting the binary number `1010001` by adding two zeros to the right, resulting in `101000100`.
- `x >> y`: Arithmetic right shift. Returns `x` with the bits shifted to the right by `y` places. The last `y` bits are discarded and `y` bits matching the sign bit (1 for negative, 0 for positive) are added to the left. This is the same as the *floor division* of `x` by `2**y`. For example, for `x=47`, shifting `x >> 2 == 47 // (2**2) = 11`. Or seen in binary, this would be shifting the number `0101111` two bits to the right, discarding the last two digits (11) and adding two 0s (because it's a positive number) at the beginning resulting in `0001011` or `1011` for short. Similarly for negative numbers, if `x=-47`, then `x >> 2 == (-47) // (2**2) == -12`. If we see it in two's complement, that would be shifting the binary number `1010001` and discarding the last two bits (01) and adding two ones to the left, resulting in `1110100` or `10100` for short in two's complement.
- `x & y`: Bitwise and. Each bit of the output is 1 if the corresponding bit of `x` AND of `y` is 1, otherwise it's 0. For example, if `x=3` and `y=5`, then `x & y = 1` because `x` in binary is `0011` and `y` in binary is `0101`. Thus, applying the and operator on each bit, we get in binary `0001`.
- `x | y`: Bitwise or. Each bit of the output is 0 if the corresponding bit of `x` AND of `y` is 0, otherwise it's 1. For example, if `x=3` and `y=5`, then `x | y = 7`. Remember that `x` in binary is `0011` and `y` in binary is `0101`. Thus, applying the or operator on each bit, we get in binary `0111`.
- `~x` Returns the complement of `x`, that the number you get by switching each 1 for a 0 and each 0 for a 1. This is the same as `-x-1`. For example, if `x=47`, then `~x` results to `-48`.
- `x ^ y`: Bitwise xor. Each bit of the output is 1 if the corresponding bit at `x` and `y` are different and 0 if they are the same. For example, if `x=3` and `y=5`, then `x ^ y = 6`. Remember that `x` in binary is `0011` and `y` in binary is `0101`. Thus, applying the xor operator on each bit, we get in binary `0110`.

Quick tips:

- Python does not have a fixed number of bits for integers. (It used to, and many languages, such as C++ and Java, do fix the number of bits for integers, typically to 32.)
- For the bitwise arithmetic to work, when `x` and `y` don't have the same number of bits, Python prepends the smaller one with 0s if it is a positive number, and with 1s if it is a negative number.
- If you have a positive number `x`, you can see its binary representation by calling `bin(x)`. For example, if `x=47`, then `bin(47)` would return `0b101111`. Ignore the `0b` part which represents a positive binary number. What you got is that 47 in binary is `101111`.

- Unfortunately, the trick to get the two's complement representation of a negative number is slightly more complicated. If you have a negative number `x`, you can see its binary representation in two's complement by calling `bin(x & 0b1111111111111111)`. For example, if `x=-47`, then `bin(x & 0b1111111111111111)` would return `0b111111111010001`. Ignore the `0b`. What you got is that `-47` in binary two's complement is `1010001`. Remember, sign extension for two's complement prepends the number with `1s`.
- If we have an integer `x`, we can extract its number of bits by calling `x.bit_length()`. The number you get back does not contain the sign.

3.2 Puzzle Requirements

- The puzzles need to be completed using only straight-line code (no loops or conditionals, that is, no `for` loops, `while` loops, or `if/else` statements) and no function calls beyond calling `bit_length()`.
- We have also imposed constraints on which of the bitwise operators you can use to solve each puzzle. Within a function that corresponds to a puzzle, you are allowed to use local variables if your solution is complex and cannot be contained in a single line. Each function's *docstring* (documentation string), whose start and end are indicated with three straight single quotes, explains what each function does, which operators you are allowed to use, and how many of them at most.

4 Testing

An important part of learning to program is learning how to test your code and ensure that it is behaving as expected. One of the most effective ways to do this is to write test cases that explicitly call your functions and compare the output to an expected answer (calculated manually).

At the bottom of the `.py` file there are tests for each of the puzzles. We have already implemented one test for you for each of the puzzles. You can immediately use these to test your functions, e.g., you can run your code interactively by typing the following in the **Terminal** window of VS Code:

```
% python3 -i assignment3.py
```

This will run your Python program and then leave you in the Python Shell with the functions loaded. You can then call any of the test functions:

```
>>> test_bitwise_and1()
True
```

TODO: For each of the puzzles, you must implement 2 additional test cases. There are functions already defined in the starter file, but you need to replace the `return False` with your test cases.

There is a function at the very end called `test_all_functions` which you can call. It will call all of the test cases and print out whether or not each function passes the test cases. For each puzzle, if all of the test passes you'll see a smiley face and if any of the three tests don't pass, you'll see a frowny face. (Note that this assumes that you have properly added all of your test cases).

When you're all done with your eights puzzles *and* your test cases, you should be able to run the `test_all_functions` function and see eight lines printed out, all with smiley faces :)

NOTE: When you submit your code, it should not have any print statements or functions calls outside of functions.

5 Feedback

Create a file named `feedback3.txt` that answers the questions:

- How long did you spend on this assignment?
- Any comments or feedback? What things did you find interesting, challenging, or boring?

6 When you're done

Submit your `assignment3.py` and `feedback3.txt` online using Gradescope. Be sure you're uploading them to the right assignment. Only one submission will be made for both partners. Note that you can resubmit the same assignment as many times as you would like up until the deadline.

7 Grading

	points
Before you come to lab	1
<code>bitwise_and</code>	0.75
<code>bitwise_nor</code>	0.75
<code>bitwise_xor</code>	0.75
<code>are_equal</code>	0.75
<code>negate</code>	0.75
<code>is_even</code>	0.75
<code>absolute</code>	1.75
<code>least_significant_bit</code>	0.75
Test functions	1.5
Submitted properly	0.5
Total	10