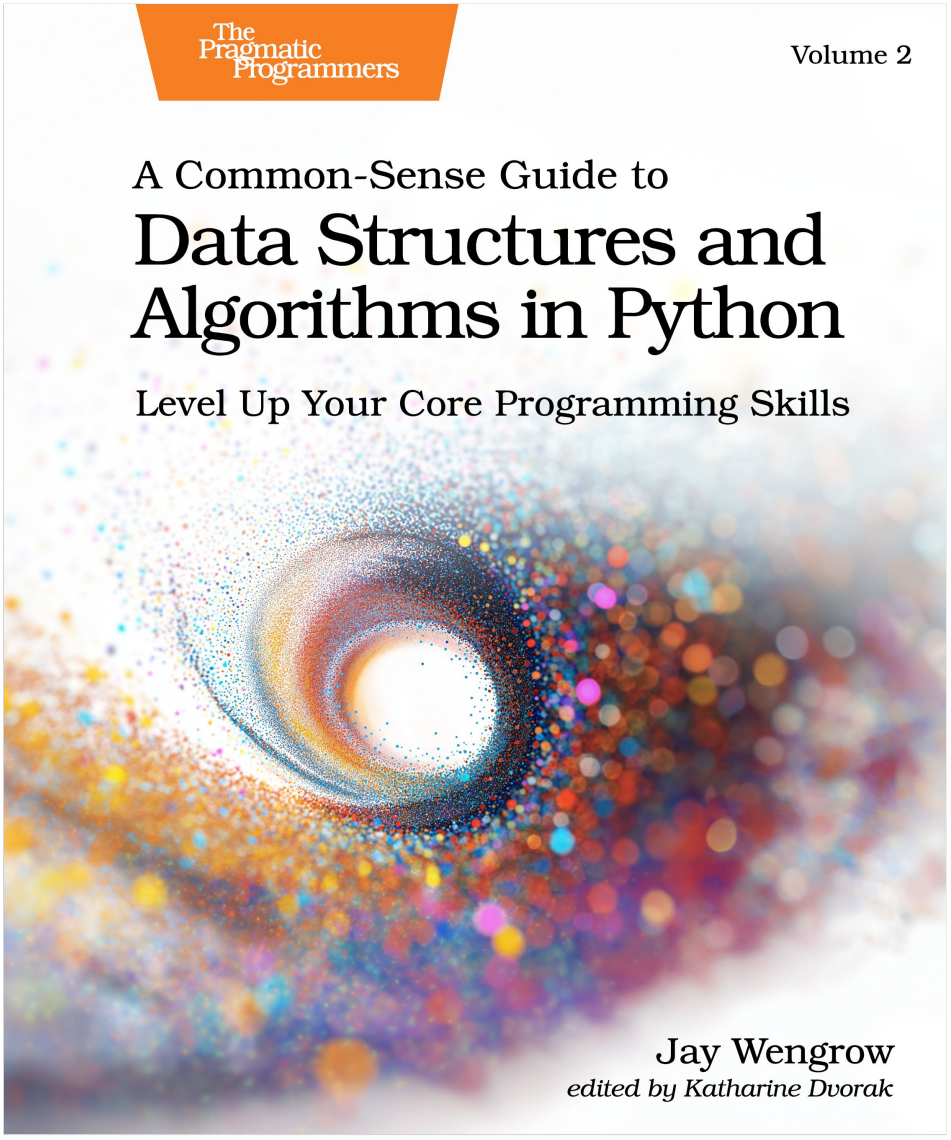

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Here's a fascinating question: how *does* a computer use bits to represent a negative number anyway? A common approach is a system known as *two's complement*. In this article, I outline how it works.

As you know, the following binary numbers represent the eight numbers 0 through 7:

```
       4  2  1  Places
--------------
0:     0  0  0
1:     0  0  1
2:     0  1  0
3:     0  1  1
4:     1  0  0
5:     1  0  1
6:     1  1  0
7:     1  1  1
```

The left-most digit place is the 4's place. With two's complement, instead of the left-most place being what you'd expect, it is instead the *negative* of that place. In this example, this means that the left-most place is the *negative 4's* place. The other digit places, though, are what you'd expect.

While two's complement might sound weird at first, it's actually incredibly useful. With the left-most place being the negative 4's place, we can still use the three-digit places to represent eight different numbers. However, instead of the numbers being `0` through `7` as above, we can now represent `-4` through `3`, as follows:

```
        -4  2  1  Places
-----------------
-4:      1  0  0
-3:      1  0  1
-2:      1  1  0
-1:      1  1  1
 0:      0  0  0
 1:      0  0  1
 2:      0  1  0
 3:      0  1  1
```

In this example, you can see that `110` in binary represents the decimal number `-2`. This is because there is one `-4` and one `2`. And `-4 + 2 = -2`.

Python does indeed use the two's complement system. However, this begs the question: why does the Python expression `0b111` return `7`? Shouldn't it return `-1`, as in the previous table?

The answer is that when we enter the Python binary string `0b111`, Python, under the hood, pads the left of the string with at least one `0` bit before computing the expression. (Actually, Python pads the string with an arbitrary number of `0` bits. You can't be sure how many `0` bits there will be, but it really doesn't matter.) So, our binary string of `111` is actually `0111`. Even with two's complement, this translates to:

```
       -8  4  2  1  Places
------------------
7:      0  1  1  1
```

That is, since there's a `0` bit in that left-most "negative" place, our number ends up being a positive number in the end.

With this in mind, you can now understand why the ~ Python operator always returns a negative decimal number. For example, let's analyze the expression `~7`.

Now, `7` in binary is `111`. However—and this is key—Python, as I said before, automatically pads each binary string with an arbitrary number of `0` bits on the left-most side of the string. So, in Python-Land, `7`, in binary, is actually something like `0111`.

And here's the clincher. When we *invert* `0111`, we get the result of `1000`. And with two's complement, this is `-8`:

```
        -8  4  2  1  Places
--------------------
-8:      1  0  0  0
```

And so, `~7` yields `-8`.

In fact, this pattern holds true for all numbers. That is, when we invert any number N, the result will be `-(N + 1)`. So, `~50` yields `-51`. And `~99` yields `-100`.

I'll point out one last thing, and it's pretty cool. Remember when I said that Python automatically pads a binary string with an *arbitrary* number of `0` bits on the left? This means that it might pad it with a single `0` bit, or it might pad it with eleven `0` bits. Now, we know that for positive numbers this doesn't matter, since `00000000001` is the same as `0001` and `01` and `1`.

What's fascinating is that the number of padded `0` bits doesn't matter for inverted numbers either. That is, whether we invert `0001` (so that it becomes `1110`), invert `001` (so that it becomes `110`), or invert `01` (so that it becomes `10`), with two's complement we'll get the same result. This is because only the left-most digit becomes the "negative" place, and so everything works out. See for yourself in the following examples.

Here's the binary number 1110:

```
        -8  4  2  1  Places
--------------------
-2:      1  1  1  0
```

That is, -8 + 4 + 2 = -2.

Now, here's the binary number 110:

```
        -4  2  1  Places
-----------------
-2:      1  1  0
```

And here's the binary number 10:

```
        -2  1  Places
--------------
-2:      1  0
```

And that's how the two's complement number system works. Now you know!