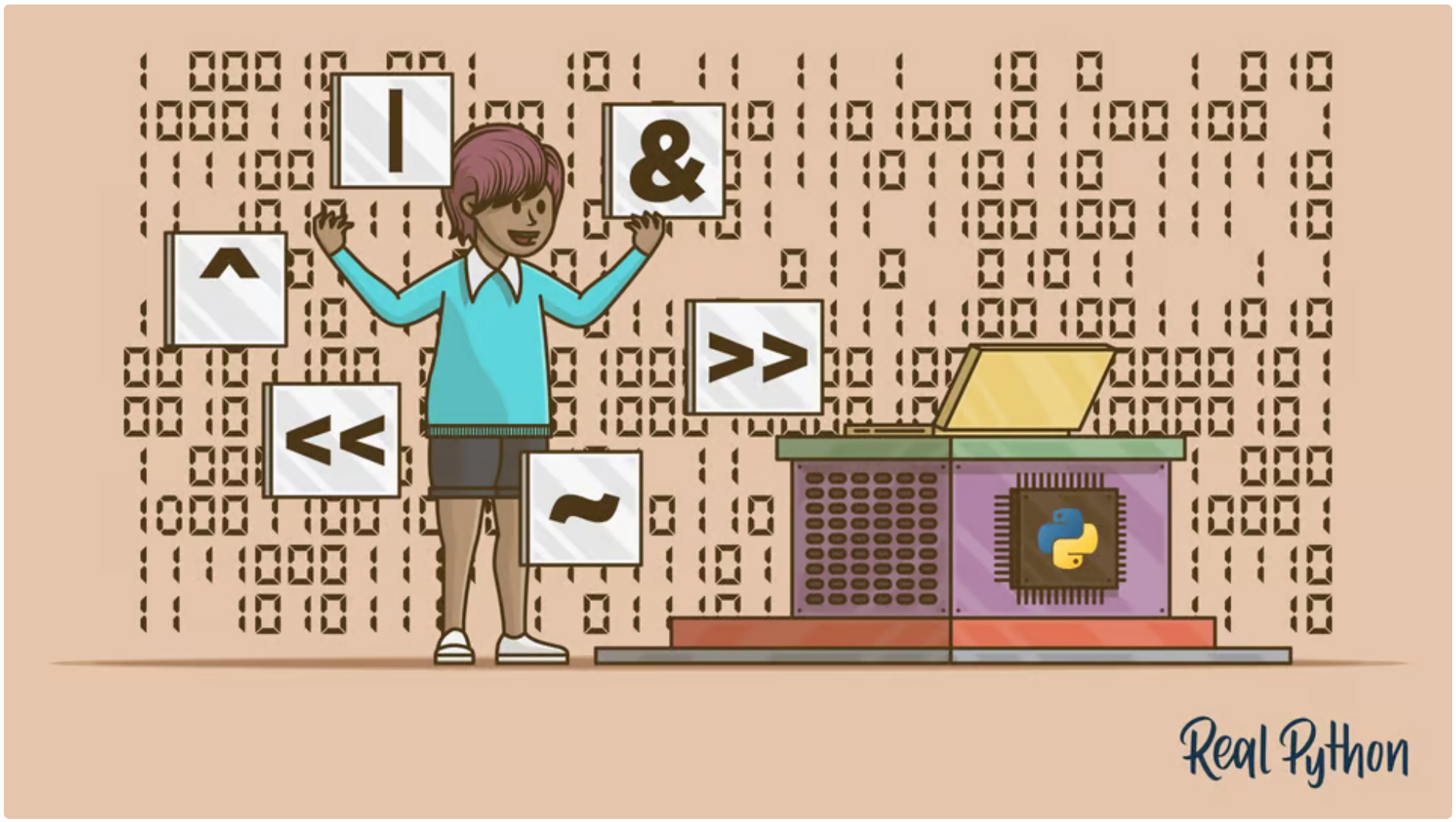


Bitwise Operators in Python

by Bartosz Zaczyński Dec 09, 2020 9 Comments intermediate python

Mark as Completed

Share Share Email

Table of Contents

Help

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Binary, Bytes, and Bitwise Operators in Python**

Computers store all kinds of information as a stream of binary digits called **bits**. Whether you're working with text, images, or videos, they all boil down to ones and zeros. Python's **bitwise operators** let you manipulate those individual bits of data at the most granular level.

You can use bitwise operators to implement algorithms such as compression, encryption, and error detection as well as to control physical devices in your Raspberry Pi project or elsewhere. Often, Python isolates you from the underlying bits with high-level abstractions. You're more likely to find the overloaded flavors of bitwise operators in practice. But when you work with them in their original form, you'll be surprised by their quirks!

In this tutorial, you'll learn how to:

- Use Python **bitwise operators** to manipulate individual bits
- Read and write binary data in a **platform-agnostic** way
- Use **bitmasks** to pack information on a single byte
- **Overload** Python bitwise operators in custom data types
- Hide **secret messages** in digital images

To get the complete source code of the digital watermarking example, and to extract a secret treat hidden in an image, click the link below:

Get the Source Code: Click here to get the source code you'll use to learn about Python's bitwise operators in this tutorial.

Overview of Python's Bitwise Operators

Python comes with a few different kinds of operators, such as the arithmetic, logical, and comparison operators. You can think of them as functions that take advantage of a more compact **prefix** and **infix** syntax.

> **Note:** Python does not include **postfix** operators like the increment (`i++`) or decrement (`i--`) operators available in C.

Bitwise operators look virtually the same across different programming languages:

Operator	Example	Meaning
`&`	`a & b`	Bitwise AND
`|`	`a | b`	Bitwise OR
`^`	`a ^ b`	Bitwise XOR (exclusive OR)
`~`	`~a`	Bitwise NOT
`<<`	`a << n`	Bitwise left shift
`>>`	`a >> n`	Bitwise right shift

As you can see, they're denoted with strange-looking symbols instead of words. This makes them stand out in Python as slightly less verbose than you might be used to seeing. You probably wouldn't be able to figure out their meaning just by looking at them.

> **Note:** If you're coming from another programming language such as Java, then you'll immediately notice that Python is missing the **unsigned right shift operator** denoted by three greater-than signs (`>>>`).
>
> This has to do with how Python represents integers internally. Since integers in Python can have an infinite number of bits, the sign bit doesn't have a fixed position. In fact, there's no sign bit at all in Python!

Most of the bitwise operators are **binary**, which means that they expect two operands to work with, typically referred to as the **left operand** and the **right operand**. Bitwise NOT (~) is the only **unary** bitwise operator since it expects just one operand.

All binary bitwise operators have a corresponding **compound operator** that performs an augmented assignment:

Operator	Example	Equivalent to
`&=`	`a &= b`	`a = a & b`
`|=`	`a |= b`	`a = a | b`
`^=`	`a ^= b`	`a = a ^ b`
`<<=`	`a <<= n`	`a = a << n`
`>>=`	`a >>= n`	`a = a >> n`

These are shorthand notations for updating the left operand in place.

That's all there is to Python's bitwise operator syntax! Now you're ready to take a closer look at each of the operators to understand where they're most useful and how you can use them. First, you'll get a quick refresher on the binary system before looking at two categories of bitwise operators: the bitwise **logical** operators and the bitwise **shift** operators.

Binary System in Five Minutes

Before moving on, take a moment to brush up your knowledge of the **binary system**, which is essential to understanding bitwise operators. If you're already comfortable with it, then go ahead and jump to the Bitwise Logical Operators section below.

Why Use Binary?

There are an infinite number of ways to represent numbers. Since ancient times, people have developed different notations, such as Roman numerals and Egyptian hieroglyphs. Most modern civilizations use positional notation, which is efficient, flexible, and well suited for doing arithmetic.

A notable feature of any positional system is its **base**, which represents the number of digits available. People naturally favor the **base-ten** numeral system, also known as the **decimal system**, because it plays nicely with counting on fingers.

Computers, on the other hand, treat data as a bunch of numbers expressed in the **base-two** numeral system, more commonly known as the **binary** system. Such numbers are composed of only two digits, zero and one.

> **Note:** In math books, the base of a numeric literal is commonly denoted with a subscript that appears slightly below the baseline, such as 42_{10}.

For example, the binary number 10011100_2 is equivalent to 156_{10} in the base-ten system. Because there are ten numerals in the decimal system—zero through nine—it usually takes fewer digits to write the same number in base ten than in base two.

> **Note:** You can't tell a numeral system just by looking at a given number's digits.
>
> For example, the decimal number 101_{10} happens to use only binary digits. But it represents a completely different value than its binary counterpart, 101_2, which is equivalent to 5_{10}.

The binary system requires more storage space than the decimal system but is much less complicated to implement in hardware. While you need more building blocks, they're easier to make, and there are fewer types of them. That's like breaking down your code into more modular and reusable pieces.

More importantly, however, the binary system is perfect for electronic devices, which translate digits into different **voltage levels**. Because voltage likes to drift up and down due to various kinds of noise, you want to keep sufficient distance between consecutive voltages. Otherwise, the signal might end up distorted.

By employing only two states, you make the system more reliable and resistant to noise. Alternatively, you could jack up the voltage, but that would also increase the **power consumption**, which you definitely want to avoid.

How Does Binary Work?

Imagine for a moment that you had only two fingers to count on. You could count a zero, a one, and a two. But when you ran out of fingers, you'd need to note how many times you had already counted to two and then start over until you reached two again:

Decimal	Fingers	Eights	Fours	Twos	Ones	Binary
0_{10}	✊	0	0	0	0	0_2
1_{10}	☝	0	0	0	1	1_2
2_{10}	✌	0	0	1	0	10_2

Decimal	Fingers	Eights	Fours	Twos	Ones	Binary
3_{10}	✌+☝	0	0	1	1	11_2
4_{10}	✌✌	0	1	0	0	100_2
5_{10}	✌✌+☝	0	1	0	1	101_2
6_{10}	✌✌+✌	0	1	1	0	110_2
7_{10}	✌✌+✌+☝	0	1	1	1	111_2
8_{10}	✌✌✌✌	1	0	0	0	1000_2
9_{10}	✌✌✌✌+☝	1	0	0	1	1001_2
10_{10}	✌✌✌✌+✌	1	0	1	0	1010_2
11_{10}	✌✌✌✌+✌+☝	1	0	1	1	1011_2
12_{10}	✌✌✌✌+✌✌	1	1	0	0	1100_2
13_{10}	✌✌✌✌+✌✌+☝	1	1	0	1	1101_2

Every time you wrote down another pair of fingers, you'd also need to group them by powers of two, which is the base of the system. For example, to count up to thirteen, you would have to use both of your fingers six times and then use one more finger. Your fingers could be arranged as one *eight*, one *four*, and one *one*.

These powers of two correspond to digit positions in a binary number and tell you exactly which bits to switch on. They grow right to left, starting at the **least-significant bit**, which determines if the number is even or odd.

Positional notation is like the odometer in your car: Once a digit in a particular position reaches its maximum value, which is one in the binary system, it rolls over to zero and the one carries over to the left. This can have a cascading effect if there are already some ones to the left of the digit.

How Computers Use Binary

Now that you know the basic principles of the binary system and *why* computers use it, you're ready to learn *how* they represent data with it.

Before any piece of information can be reproduced in digital form, you have to break it down into numbers and then convert them to the binary system. For example, plain text can be thought of as a string of characters. You could assign an arbitrary number to each character or pick an existing character encoding such as ASCII, ISO-8859-1, or UTF-8.

In Python, strings are represented as arrays of Unicode code points. To reveal their ordinal values, call `ord()` on each of the characters:

Python

```
>>> [ord(character) for character in "€uro"]
[8364, 117, 114, 111]
```

The resulting numbers uniquely identify the text characters within the Unicode space, but they're shown in decimal form. You want to rewrite them using binary digits:

Character	Decimal Code Point	Binary Code Point
€	8364_{10}	10000010101100_2

Character	Decimal Code Point	Binary Code Point
u	117_{10}	1110101_2
r	114_{10}	1110010_2
o	111_{10}	1101111_2

Notice that **bit-length**, which is the number of binary digits, varies greatly across the characters. The euro sign (€) requires fourteen bits, while the rest of the characters can comfortably fit on seven bits.

Note: Here's how you can check the bit-length of any integer number in Python:

Python

```
>>> (42).bit_length()
6
```

Without a pair of parentheses around the number, it would be treated as a floating-point literal with a decimal point.

Variable bit-lengths are problematic. If you were to put those binary numbers next to one another on an optical disc, for example, then you'd end up with a long stream of bits without clear boundaries between the characters:

$1000001010110011101011110010110111 1_2$

One way of knowing how to interpret this information is to designate fixed-length bit patterns for all characters. In modern computing, the smallest unit of information, called an **octet** or a **byte**, comprises eight bits that can store 256 distinct values.

You can pad your binary code points with leading zeros to express them in terms of bytes:

Character	Decimal Code Point	Binary Code Point
€	8364_{10}	$00100000\ 10101100_2$
u	117_{10}	$00000000\ 01110101_2$
r	114_{10}	$00000000\ 01110010_2$
o	111_{10}	$00000000\ 01101111_2$

Now each character takes two bytes, or 16 bits. In total, your original text almost doubled in size, but at least it's encoded reliably.

You can use Huffman coding to find unambiguous bit patterns for every character in a particular text or use a more suitable character encoding. For example, to save space, UTF-8 intentionally favors Latin letters over symbols that you're less likely to find in an English text:

Python

```
>>> len("€uro".encode("utf-8"))
6
```

Encoded according to the UTF-8 standard, the entire text takes six bytes. Since UTF-8 is a superset of ASCII, the letters `u`, `r`, and `o` occupy one byte each, whereas the euro symbol takes three bytes in this encoding:

Python

```
>>> for char in "€uro":
...     print(char, len(char.encode("utf-8")))
...
€ 3
u 1
r 1
o 1
```

Other types of information can be digitized similarly to text. Raster images are made of pixels, with every pixel having channels that represent color intensities as numbers. Sound waveforms contain numbers corresponding to air pressure at a given sampling interval. Three-dimensional models are built from geometric shapes defined by their vertices, and so forth.

At the end of the day, everything is a number.

Bitwise Logical Operators

You can use bitwise operators to perform Boolean logic on individual bits. That's analogous to using logical operators such as `and`, `or`, and `not`, but on a bit level. The similarities between bitwise and logical operators go beyond that.

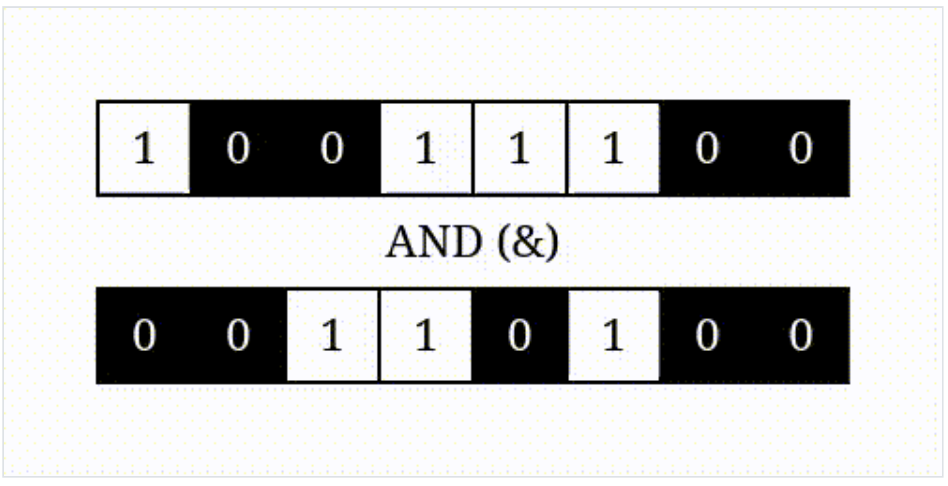
It's possible to evaluate Boolean expressions with bitwise operators instead of logical operators, but such overuse is generally discouraged. If you're interested in the details, then you can expand the box below to find out more.

Evaluating Boolean Expressions With Bitwise Operators Show/Hide

Unless you have a strong reason and know what you're doing, you should use bitwise operators only for controlling bits. It's too easy to get it wrong otherwise. In most cases, you'll want to pass integers as arguments to the bitwise operators.

Bitwise AND

The bitwise AND operator (`&`) performs logical conjunction on the corresponding bits of its operands. For each pair of bits occupying the same position in the two numbers, it returns a one only when both bits are switched on:



The resulting bit pattern is an **intersection** of the operator's arguments. It has two bits turned on in the positions where both operands are ones. In all other places, at least one of the inputs has a zero bit.

Arithmetically, this is equivalent to a **product** of two bit values. You can calculate the bitwise AND of numbers *a* and *b* by multiplying their bits at every index *i*:

$$(a \ \& \ b)_i = a_i \times b_i$$

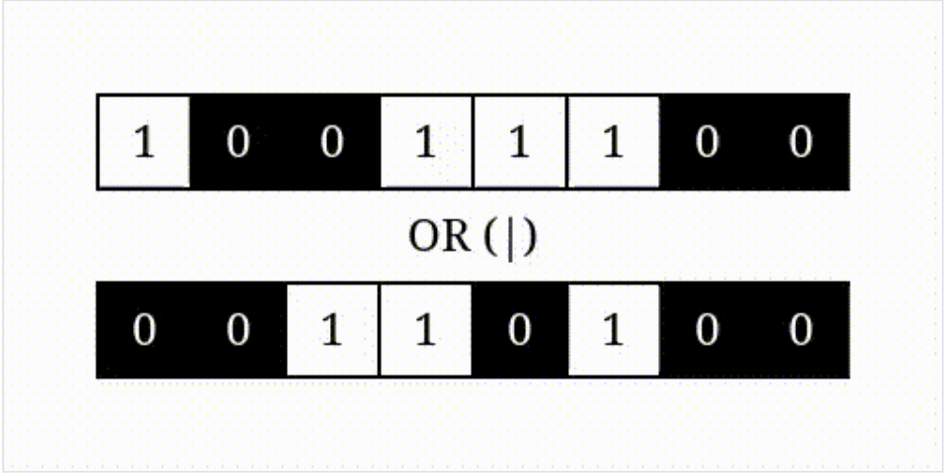
Here's a concrete example:

Expression	Binary Value	Decimal Value
`a`	10011100_2	156_{10}
`b`	110100_2	52_{10}
`a & b`	10100_2	20_{10}

A one multiplied by one gives one, but anything multiplied by zero will always result in zero. Alternatively, you can take the minimum of the two bits in each pair. Notice that when operands have unequal bit-lengths, the shorter one is automatically padded with zeros to the left.

Bitwise OR

The bitwise OR operator (`|`) performs logical disjunction. For each corresponding pair of bits, it returns a one if at least one of them is switched on:



The resulting bit pattern is a **union** of the operator's arguments. It has five bits turned on where either of the operands has a one. Only a combination of two zeros gives a zero in the final output.

The arithmetic behind it is a combination of a **sum** and a **product** of the bit values. To calculate the bitwise OR of numbers *a* and *b*, you need to apply the following formula to their bits at every index *i*:

$$(a \mid b)_i = a_i + b_i - (a_i \times b_i)$$

Here's a tangible example:

Expression	Binary Value	Decimal Value
`a`	10011100_2	156_{10}
`b`	110100_2	52_{10}
`a | b`	10111100_2	188_{10}

It's almost like a sum of two bits but clamped at the higher end so that it never exceeds the value of one. You could also take the maximum of the two bits in each pair to get the same result.

Bitwise XOR

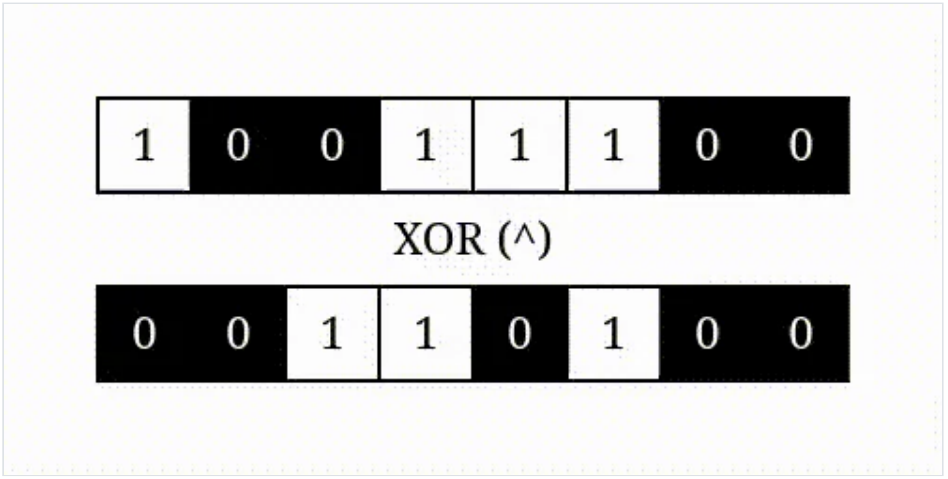
Unlike bitwise AND, OR, and NOT, the bitwise XOR operator (^) doesn't have a logical counterpart in Python. However, you can simulate it by building on top of the existing operators:

Python

```
def xor(a, b):
    return (a and not b) or (not a and b)
```

It evaluates two mutually exclusive conditions and tells you whether exactly one of them is met. For example, a person can be either a minor or an adult, but not both at the same time. Conversely, it's not possible for a person to be neither a minor nor an adult. The choice is mandatory.

The name XOR stands for "exclusive or" since it performs exclusive disjunction on the bit pairs. In other words, every bit pair must contain opposing bit values to produce a one:



Visually, it's a **symmetric difference** of the operator's arguments. There are three bits switched on in the result where both numbers have different bit values. Bits in the remaining positions cancel out because they're the same.

Similarly to the bitwise OR operator, the arithmetic of XOR involves a sum. However, while the bitwise OR clamps values at one, the XOR operator wraps them around with a **sum modulo two**:

$$(a \hat{} b)_i = (a_i + b_i) \bmod 2$$

Modulo is a function of two numbers—the **dividend** and the **divisor**—that performs a division and returns its remainder. In Python, there's a built-in modulo operator denoted with the percent sign (%).

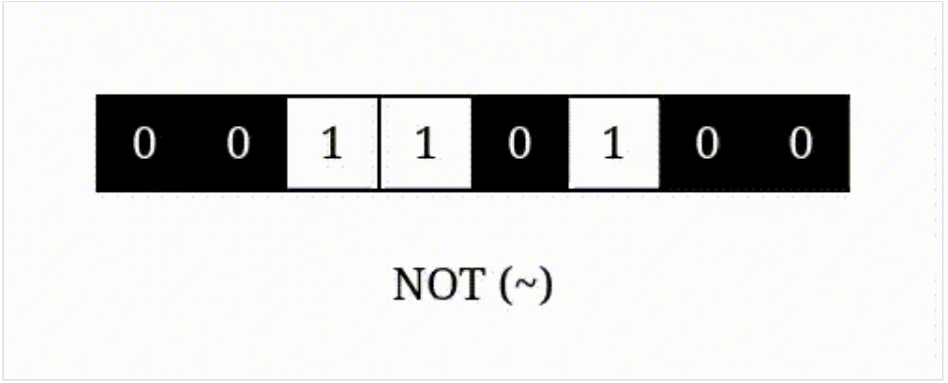
Once again, you can confirm the formula by looking at an example:

Expression	Binary Value	Decimal Value
a	10011100_2	156_{10}
b	110100_2	52_{10}
a ^ b	10101000_2	168_{10}

The sum of two zeros or two ones yields a whole number when divided by two, so the result has a remainder of zero. However, when you divide the sum of two *different* bit values by two, you get a fraction with a remainder of one. A more straightforward formula for the XOR operator is the difference between the maximum and the minimum of both bits in each pair.

Bitwise NOT

The last of the bitwise logical operators is the bitwise NOT operator (~), which expects just one argument, making it the only unary bitwise operator. It performs logical negation on a given number by flipping all of its bits:



The inverted bits are a **complement** to one, which turns zeros into ones and ones into zeros. It can be expressed arithmetically as the **subtraction** of individual bit values from one:

$$\sim a_i = 1 - a_i$$

Here's an example showing one of the numbers used before:

Expression	Binary Value	Decimal Value
a	10011100_2	156_{10}
~a	1100011_2	99_{10}

While the bitwise NOT operator seems to be the most straightforward of them all, you need to exercise extreme caution when using it in Python. Everything you've read so far is based on the assumption that numbers are represented with **unsigned** integers.

> **Note:** Unsigned data types don't let you store negative numbers such as -273 because there's no space for a sign in a regular bit pattern. Trying to do so would result in a compilation error, a runtime exception, or an integer overflow depending on the language used.

Although there are ways to simulate unsigned integers, Python doesn't support them natively. That means all numbers have an implicit sign attached to them whether you specify one or not. This shows when you do a bitwise NOT of any number:

Python

```
>>> ~156
-157
```

Instead of the expected 99_{10}, you get a negative value! The reason for this will become clear once you learn about the various binary number representations. For now, the quick-fix solution is to take advantage of the bitwise AND operator:

Python

```
>>> ~156 & 255
99
```

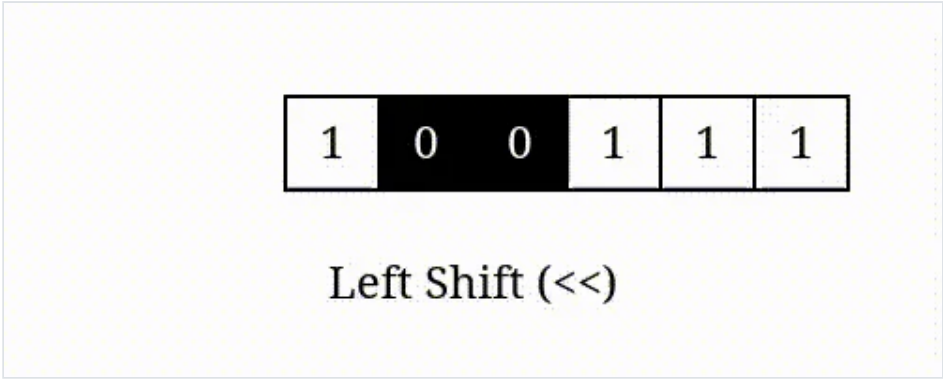
That's a perfect example of a bitmask, which you'll explore in one of the upcoming sections.

Bitwise Shift Operators

Bitwise shift operators are another kind of tool for bit manipulation. They let you move the bits around, which will be handy for creating bitmasks later on. In the past, they were often used to improve the speed of certain mathematical operations.

Left Shift

The bitwise left shift operator (<<) moves the bits of its first operand to the left by the number of places specified in its second operand. It also takes care of inserting enough zero bits to fill the gap that arises on the right edge of the new bit pattern:



Shifting a single bit to the left by one place doubles its value. For example, instead of a two, the bit will indicate a four after the shift. Moving it two places to the left will quadruple the resulting value. When you add up all the bits in a given number, you'll notice that it also gets doubled with every place shifted:

Expression	Binary Value	Decimal Value
`a`	100111_2	39_{10}
`a << 1`	1001110_2	78_{10}
`a << 2`	10011100_2	156_{10}
`a << 3`	100111000_2	312_{10}

In general, shifting bits to the left corresponds to multiplying the number by a **power of two**, with an exponent equal to the number of places shifted:

$$a \ll n = a \times 2^n$$

The left shift used to be a popular **optimization** technique because bit shifting is a single instruction and is cheaper to calculate than exponent or product. Today, however, compilers and interpreters, including Python's, are quite capable of optimizing your code behind the scenes.

> **Note:** Don't use the bit shift operators as a means of premature optimization in Python. You won't see a difference in execution speed, but you'll most definitely make your code less readable.

On paper, the bit pattern resulting from a left shift becomes longer by as many places as you shift it. That's also true for Python in general because of how it handles integers. However, in most practical cases, you'll want to constrain the length of a bit pattern to be a multiple of eight, which is the standard byte length.

For example, if you're working with a single byte, then shifting it to the left should discard all the bits that go beyond its left boundary:

It's sort of like looking at an unbounded stream of bits through a fixed-length window. There are a few tricks that let you do this in Python. For example, you can apply a bitmask with the bitwise AND operator:

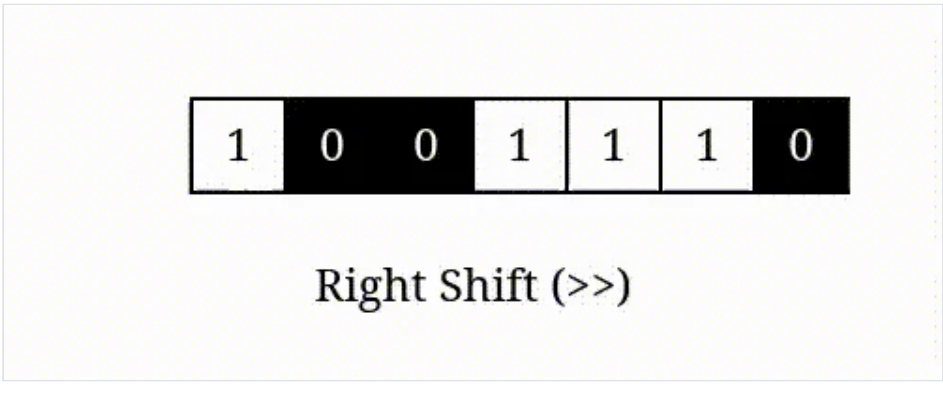
Python

```
>>> 39 << 3
312
>>> (39 << 3) & 255
56
```

Shifting 39_{10} by three places to the left returns a number higher than the maximum value that you can store on a single byte. It takes nine bits, whereas a byte has only eight. To chop off that one extra bit on the left, you can apply a bitmask with the appropriate value. If you'd like to keep more or fewer bits, then you'll need to modify the mask value accordingly.

Right Shift

The bitwise right shift operator (>>) is analogous to the left one, but instead of moving bits to the left, it pushes them to the right by the specified number of places. The rightmost bits always get dropped:



Every time you shift a bit to the right by one position, you halve its underlying value. Moving the same bit by two places to the right produces a quarter of the original value, and so on. When you add up all the individual bits, you'll see that the same rule applies to the number they represent:

Expression	Binary Value	Decimal Value
`a`	10011101_2	157_{10}
`a >> 1`	1001110_2	78_{10}
`a >> 2`	100111_2	39_{10}
`a >> 3`	10011_2	19_{10}

Halving an odd number such as 157_{10} would produce a fraction. To get rid of it, the right shift operator automatically floors the result. It's virtually the same as a **floor division** by a power of two:

$$a \gg n = \left\lfloor \frac{a}{2^n} \right\rfloor$$

Again, the exponent corresponds to the number of places shifted to the right. In Python, you can leverage a dedicated operator to perform a floor division:

Python

```
>>> 5 >> 1  # Bitwise right shift
2
>>> 5 // 2  # Floor division (integer division)
2
>>> 5 / 2   # Floating-point division
2.5
```

The bitwise right shift operator and the floor division operator both work the same way, even for negative numbers. However, the floor division lets you choose any divisor and not just a power of two. Using the bitwise right shift was a common way of improving the performance of some arithmetic divisions.

Note: You might be wondering what happens when you run out of bits to shift. For example, when you try pushing by more places than there are bits in a number:

Python

```
>>> 2 >> 5
0
```

Once there are no more bits switched on, you're stuck with a value of zero. Zero divided by anything will always return zero. However, things get trickier when you right shift a negative number because the implicit sign bit gets in the way:

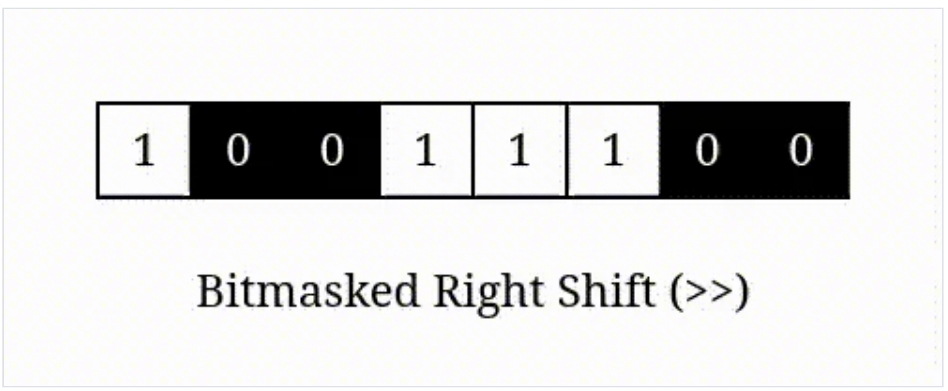
Python

```
>>> -2 >> 5
-1
```

The rule of thumb is that, regardless of the sign, the result will be the same as a floor division by some power of two. The floor of a small negative fraction is always minus one, and that's what you'll get. Read on for a more detailed explanation.

Just like with the left shift operator, the bit pattern changes its size after a right shift. While moving bits to the right makes the binary sequence shorter, it usually won't matter because you can put as many zeros in front of a bit sequence as you like without changing the value. For example, 101_2 is the same as 0101_2, and so is 00000101_2, provided that you're dealing with nonnegative numbers.

Sometimes you'll want to keep a given bit-length after doing a right shift to align it against another value or to fit in somewhere. You can do that by applying a bitmask:



It carves out only those bits you're interested in and fills the bit pattern with leading zeros if necessary.

The handling of negative numbers in Python is slightly different from the traditional approach to bitwise shifting. In the next section, you'll examine this in more detail.

Arithmetic vs Logical Shift

You can further categorize the bitwise shift operators as **arithmetic** and **logical** shift operators. While Python only lets you do the arithmetic shift, it's worthwhile to know how other programming languages implement the bitwise shift operators to avoid confusion and surprises.

This distinction comes from the way they handle the **sign bit**, which ordinarily lies at the far left edge of a signed binary sequence. In practice, it's relevant only to the right shift operator, which can cause a number to flip its sign, leading to integer overflow.

> **Note:** Java and JavaScript, for example, distinguish the logical right shift operator with an additional greater-than sign (>>>). Since the left shift operator behaves consistently across both kinds of shifts, these languages don't define a logical left shift counterpart.

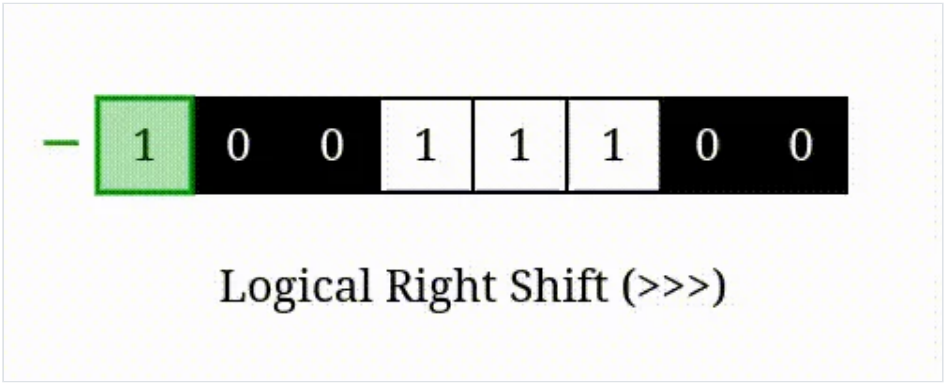
Conventionally, a turned-on sign bit indicates negative numbers, which helps keep the arithmetic properties of a binary sequence:

Decimal Value	Signed Binary Value	Sign Bit	Sign	Meaning
-100_{10}	10011100_2	1	-	Negative number
28_{10}	00011100_2	0	+	Positive number or zero

Looking from the left at these two binary sequences, you can see that their first bit carries the sign information, while the remaining part consists of the **magnitude** bits, which are the same for both numbers.

> **Note:** The specific decimal values will depend on how you decide to express signed numbers in binary. It varies between languages and gets even more complicated in Python, so you can ignore it for the moment. You'll have a better picture once you get to the binary number representations section below.

A logical right shift, also known as an **unsigned right shift** or a **zero-fill right shift**, moves the entire binary sequence, including the sign bit, and fills the resulting gap on the left with zeros:



Notice how the information about the sign of the number is lost. Regardless of the original sign, it'll always produce a nonnegative integer because the sign bit gets replaced by zero. As long as you aren't interested in the numeric values, a logical right shift can be useful in processing low-level binary data.

However, because signed binary numbers are typically stored on a fixed-length bit sequence in most languages, it can make the result wrap around the extreme values. You can see this in an interactive Java Shell tool:

Java

```
jshell> -100 >>> 1
$1 ==> 2147483598
```

The resulting number changes its sign from negative to positive, but it also overflows, ending up very close to Java's maximum integer:

Java

```
jshell> Integer.MAX_VALUE
$2 ==> 2147483647
```

This number may seem arbitrary at first glance, but it's directly related to the number of bits that Java allocates for the `Integer` data type:

Java

```
jshell> Integer.toBinaryString(-100)
$3 ==> "11111111111111111111111110011100"
```

It uses 32 bits to store signed integers in two's complement representation. When you take the sign bit out, you're left with 31 bits, whose maximum decimal value is equal to 2^{31} - 1, or 2147483647_{10}.

Python, on the other hand, stores integers as if there were an infinite number of bits at your disposal. Consequently, a logical right shift operator wouldn't be well defined in pure Python, so it's missing from the language. You can still simulate it, though.

One way of doing so is to take advantage of the unsigned data types available in C that are exposed through the built-in `ctypes` module:

Python

```
>>> from ctypes import c_uint32 as unsigned_int32
>>> unsigned_int32(-100).value >> 1
2147483598
```

They let you pass in a negative number but don't attach any special meaning to the sign bit. It's treated like the rest of the magnitude bits.

While there are only a few predefined unsigned integer types in C, which differ in bit-length, you can create a custom function in Python to handle arbitrary bit-lengths:

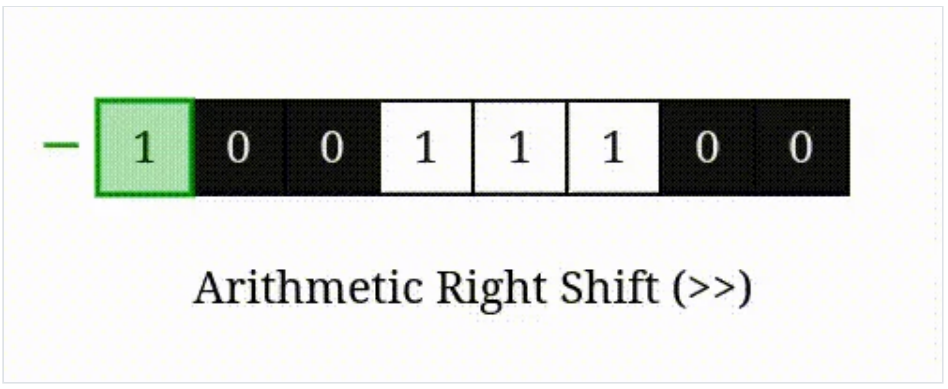
Python

```
>>> def logical_rshift(signed_integer, places, num_bits=32):
...     unsigned_integer = signed_integer % (1 << num_bits)
...     return unsigned_integer >> places
...
>>> logical_rshift(-100, 1)
2147483598
```

This converts a signed bit sequence to an unsigned one and then performs the regular arithmetic right shift.

However, since bit sequences in Python aren't fixed in length, they don't really have a sign bit. Moreover, they don't use the traditional two's complement representation like in C or Java. To mitigate that, you can take advantage of the modulo operation, which will keep the original bit patterns for positive integers while appropriately wrapping around the negative ones.

An arithmetic right shift (>>), sometimes called the **signed right shift** operator, maintains the sign of a number by replicating its sign bit before moving bits to the right:



In other words, it fills the gap on the left with whatever the sign bit was. Combined with the two's complement representation of signed binary, this results in an arithmetically correct value. Regardless of whether the number is positive or negative, an arithmetic right shift is equivalent to floor division.

As you're about to find out, Python doesn't always store integers in plain two's complement binary. Instead, it follows a custom adaptive strategy that works like sign-magnitude with an unlimited number of bits. It converts numbers back and forth between their internal representation and two's complement to mimic the standard behavior of the arithmetic shift.

Binary Number Representations

You've experienced firsthand the lack of unsigned data types in Python when using the bitwise negation (~) and the right shift operator (>>). You've seen hints about the unusual approach to storing integers in Python, which makes handling negative numbers tricky. To use bitwise operators effectively, you need to know about the various representations of numbers in binary.

Unsigned Integers

In programming languages like C, you choose whether to use the signed or unsigned flavor of a given numeric type. Unsigned data types are more suitable when you know for sure that you'll never need to deal with negative numbers. By allocating that one extra bit, which would otherwise serve as a sign bit, you practically double the range of available values.

It also makes things a little safer by increasing the maximum limit before an overflow happens. However, overflows happen only with fixed bit-lengths, so they're irrelevant to Python, which doesn't have such constraints.

The quickest way to get a taste of the unsigned numeric types in Python is to use the previously mentioned `ctypes` module:

Python

```
>>> from ctypes import c_uint8 as unsigned_byte
>>> unsigned_byte(-42).value
214
```

Since there's no sign bit in such integers, all their bits represent the magnitude of a number. Passing a negative number forces Python to reinterpret the bit pattern as if it had only the magnitude bits.

Signed Integers

The sign of a number has only two states. If you ignore zero for a moment, then it can be either positive or negative, which translates nicely to the binary system. Yet there are a few alternative ways to represent signed integers in binary, each with its own pros and cons.

Probably the most straightforward one is the **sign-magnitude**, which builds naturally on top of unsigned integers. When a binary sequence is interpreted as sign-magnitude, the most significant bit plays the role of a sign bit, while the rest of the bits work the same as usual:

Binary Sequence	Sign-Magnitude Value	Unsigned Value
00101010_2	42_{10}	42_{10}
10101010_2	-42_{10}	170_{10}

A zero on the leftmost bit indicates a positive (+) number, and a one indicates a negative (-) number. Notice that a sign bit doesn't contribute to the number's absolute value in sign-magnitude representation. It's there only to let you flip the sign of the remaining bits.

Why the leftmost bit?

It keeps **bit indexing** intact, which, in turn, helps maintain backward compatibility of the bit weights used to calculate the decimal value of a binary sequence. However, not everything about sign-magnitude is so great.

> **Note:** Binary representations of signed integers only make sense on fixed-length bit sequences. Otherwise, you couldn't tell where the sign bit was. In Python, however, you can represent integers with as many bits as you like:
>
> Python
>
> ```
> >>> f"{-5 & 0b1111:04b}"
> '1011'
> >>> f"{-5 & 0b11111111:08b}"
> '11111011'
> ```
>
> Whether it's four bits or eight, the sign bit will always be found in the leftmost position.

The range of values that you can store in a sign-magnitude bit pattern is **symmetrical**. But it also means that you end up with two ways to convey zero:

Binary Sequence	Sign-Magnitude Value	Unsigned Value
00000000_2	$+0_{10}$	0_{10}
10000000_2	-0_{10}	128_{10}

Zero doesn't technically have a sign, but there's no way not to include one in sign-magnitude. While having an **ambiguous zero** isn't ideal in most cases, it's not the worst part of the story. The biggest downside of this method is cumbersome binary arithmetic.

When you apply standard **binary arithmetic** to numbers stored in sign-magnitude, it may not give you the expected results. For example, adding two numbers with the same magnitude but opposite signs won't make them cancel out:

Expression	Binary Sequence	Sign-Magnitude Value
`a`	00101010_2	42_{10}
`b`	10101010_2	-42_{10}
`a + b`	11010100_2	-84_{10}

The sum of 42 and -42 doesn't produce zero. Also, the carryover bit can sometimes propagate from magnitude to the sign bit, inverting the sign and yielding an unexpected result.

To address these problems, some of the early computers employed **one's complement** representation. The idea was to change how decimal numbers are mapped to particular binary sequences so that they can be added up correctly. For a deeper dive into one's complement, you can expand the section below.

> One's Complement Show/Hide

Nevertheless, modern computers don't use one's complement to represent integers because there's an even better way called **two's complement**. By applying a small modification, you can eliminate double zero and simplify the binary arithmetic in one go. To explore two's complement in more detail, you can expand the section below.

> Two's Complement Show/Hide

There are a few other variants of signed number representations, but they're not as popular.

Floating-Point Numbers

The IEEE 754 standard defines a binary representation for real numbers consisting of the **sign, exponent, and mantissa** bits. Without getting into too many technical details, you can think of it as the scientific notation for binary numbers. The decimal point "floats" around to accommodate a varying number of significant figures, except it's a binary point.

Two data types conforming to that standard are widely supported:

1. **Single precision:** 1 sign bit, 8 exponent bits, 23 mantissa bits
2. **Double precision:** 1 sign bit, 11 exponent bits, 52 mantissa bits

Python's `float` data type is equivalent to the double-precision type. Note that some applications require more or fewer bits. For example, the OpenEXR image format takes advantage of half precision to represent pixels with a high dynamic range of colors at a reasonable file size.

The number Pi (π) has the following binary representation in single precision when rounded to five decimal places:

Sign	Exponent	Mantissa
0_2	10000000_2	$.10010010000111111010000_2$

The sign bit works just like with integers, so zero denotes a positive number. For the exponent and mantissa, however, different rules can apply depending on a few edge cases.

First, you need to convert them from binary to the decimal form:

- **Exponent:** 128_{10}
- **Mantissa:** $2^{-1} + 2^{-4} + \ldots + 2^{-19} = 299261_{10}/524288_{10} \approx 0.570795_{10}$

The exponent is stored as an unsigned integer, but to account for negative values, it usually has a bias equal to 127_{10} in single precision. You need to subtract it to recover the actual exponent.

Mantissa bits represent a fraction, so they correspond to negative powers of two. Additionally, you need to add one to the mantissa because it assumes an implicit leading bit before the radix point in this particular case.

Putting it all together, you arrive at the following formula to convert a floating-point binary number into a decimal one:

$$(-1)^{sign} \times 2^{(exponent-bias)} \times (mantissa + 1)$$

When you substitute the variables for the actual values in the example above, you'll be able to decipher the bit pattern of a floating-point number stored in single precision:

$$\begin{aligned} \pi &= (-1)^0 \times 2^{(128-127)} \times (0.570795 + 1) \\ &= 1 \times 2 \times 1.570795 \\ &= 3.14159 \end{aligned}$$

There it is, granted that Pi has been rounded to five decimal places. You'll learn how to display such numbers in binary later on.

Fixed-Point Numbers

While floating-point numbers are a good fit for engineering purposes, they fail in **monetary calculations** due to their limited precision. For example, some numbers with a finite representation in decimal notation have only an infinite representation in binary. That often results in a rounding error, which can accumulate over time:

Python

```
>>> 0.1 + 0.2
0.30000000000000004
```

In such cases, you're better off using Python's `decimal` module, which implements **fixed-point** arithmetic and lets you specify where to put the decimal point on a given bit-length. For example, you can tell it how many digits you want to preserve:

Python

```
>>> from decimal import Decimal, localcontext
>>> with localcontext() as context:
...     context.prec = 5  # Number of digits
...     print(Decimal("123.456") * 1)
...
123.46
```

However, it includes all digits, not just the fractional ones.

> **Note:** If you're working with rational numbers, then you might be interested in checking out the `fractions` module, which is part of Python's standard library.

If you can't or don't want to use a fixed-point data type, a straightforward way to reliably store currency values is to scale the amounts to the smallest unit, such as cents, and represent them with integers.

Integers in Python

In the old days of programming, computer memory was at a premium. Therefore, languages would give you pretty granular control over how many bytes to allocate for your data. Let's take a quick peek at a few integer types from C as an example:

Type	Size	Minimum Value	Maximum Value
`char`	1 byte	-128	127
`short`	2 bytes	-32,768	32,767
`int`	4 bytes	-2,147,483,648	2,147,483,647
`long`	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

These values might vary from platform to platform. However, such an abundance of numeric types allows you to arrange data in memory compactly. Remember that these don't even include unsigned types!

On the other end of the spectrum are languages such as JavaScript, which have just one numeric type to rule them all. While this is less confusing for beginning programmers, it comes at the price of increased memory consumption, reduced processing efficiency, and decreased precision.

When talking about bitwise operators, it's essential to understand how Python handles integer numbers. After all, you'll use these operators mainly to work with integers. There are a couple of wildly different representations of integers in Python that depend on their values.

Interned Integers

In CPython, very small integers between -5_{10} and 256_{10} are interned in a global cache to gain some performance because numbers in that range are commonly used. In practice, whenever you refer to one of those values, which are singletons created at the interpreter startup, Python will always provide the same instance:

Python

```
>>> a = 256
>>> b = 256
>>> a is b
True
>>> print(id(a), id(b), sep="\n")
94050914330336
94050914330336
```

Both variables have the same identity because they refer to the exact same object in memory. That's typical of reference types but not immutable values such as integers. However, when you go beyond that range of cached values, Python will start creating distinct copies during variable assignment:

Python

```
>>> a = 257
>>> b = 257
>>> a is b
False
>>> print(id(a), id(b), sep="\n")
140282370939376
140282370939120
```

Despite having equal values, these variables point to separate objects now. But don't let that fool you. Python will occasionally jump in and optimize your code behind the scenes. For example, it'll cache a number that occurs on the same line multiple times regardless of its value:

Python

```
>>> a = 257
>>> b = 257
>>> print(id(a), id(b), sep="\n")
140258768039856
140258768039728
>>> print(id(257), id(257), sep="\n")
140258768039760
140258768039760
```

Variables `a` and `b` are independent objects because they reside at different memory locations, while the numbers used literally in `print()` are, in fact, the same object.

> **Note:** Interning is an implementation detail of the CPython interpreter, which might change in future versions, so don't rely on it in your programs.

Interestingly, there's a similar **string interning** mechanism in Python, which kicks in for short texts comprised of ASCII letters only. It helps speed up dictionary lookups by allowing their keys to be compared by memory addresses, or C pointers, instead of by the individual string characters.

Fixed-Precision Integers

Integers that you're most likely to find in Python will leverage the C `signed long` data type. They use the classic two's complement binary representation on a fixed number of bits. The exact bit-length will depend on your hardware platform, operating system, and Python interpreter version.

Modern computers typically use 64-bit architecture, so this would translate to decimal numbers between -2^{63} and $2^{63} - 1$. You can check the maximum value of a fixed-precision integer in Python in the following way:

Python

```
>>> import sys
>>> sys.maxsize
9223372036854775807
```

It's huge! Roughly 9 million times the number of stars in our galaxy, so it should suffice for everyday use. While the maximum value that you could squeeze out of the `unsigned long` type in C is even bigger, on the order of 10^{19}, integers in Python have no theoretical limit. To allow this, numbers that don't fit on a fixed-length bit sequence are stored differently in memory.

Arbitrary-Precision Integers

Do you remember that popular K-pop song "Gangnam Style" that became a worldwide hit in 2012? The YouTube video was the first to break a billion views. Soon after that, so many people had watched the video that it made the view counter overflow. YouTube had no choice but to upgrade their counter from 32-bit signed integers to 64-bit ones.

That might give plenty of headroom for a view counter, but there are even bigger numbers that aren't uncommon in real life, notably in the scientific world. Nonetheless, Python can deal with them effortlessly:

Python

```
>>> from math import factorial
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
```

This number has fifty-two decimal digits. It would take at least 170 bits to represent it in binary with the traditional approach:

Python

```
>>> factorial(42).bit_length()
170
```

Since they're well over the limits that any of the C types have to offer, such astronomical numbers are converted into a sign-magnitude positional system, whose base is 2^{30}. Yes, you read that correctly. Whereas you have ten fingers, Python has over a billion!

Again, this may vary depending on the platform you're currently using. When in doubt, you can double-check:

Python

```
>>> import sys
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

This will tell you how many bits are used per digit and what the size in bytes is of the underlying C structure. To get the same namedtuple in Python 2, you'd refer to the `sys.long_info` attribute instead.

While this conversion between fixed- and arbitrary-precision integers is done seamlessly under the hood in Python 3, there was a time when things were more explicit. For more information, you can expand the box below.

`int` and `long` in Python 2 Show/Hide

Such a representation eliminates integer overflow errors and gives the illusion of infinite bit-length, but it requires significantly more memory. Additionally, performing bignum arithmetic is slower than with fixed precision because it can't run directly in hardware without an intermediate layer of emulation.

Another challenge is keeping a consistent behavior of the bitwise operators across alternative integer types, which is crucial in handling the sign bit. Recall that fixed-precision integers in Python use the standard two's complement representation from C, while large integers use sign-magnitude.

To mitigate that difference, Python will do the necessary binary conversion for you. It might change how a number is represented before and after applying a bitwise operator. Here's a relevant comment from the CPython source code, which explains this in more detail:

> Bitwise operations for negative numbers operate as though on a two's complement representation. So convert arguments from sign-magnitude to two's complement, and convert the result back to sign-magnitude at the end. (Source)

In other words, **negative numbers** are treated as two's complement bit sequences when you apply the bitwise operators on them, even though the result will be presented to you in sign-magnitude form. There are ways to emulate the sign bit and some of the unsigned types in Python, though.

Bit Strings in Python

You're welcome to use pen and paper throughout the rest of this article. It may even serve as a great exercise! However, at some point, you'll want to verify whether your binary sequences or **bit strings** correspond to the expected numbers in Python. Here's how.

Converting `int` to Binary

To reveal the bits making up an integer number in Python, you can print a formatted string literal, which optionally lets you specify the number of leading zeros to display:

Python

```
>>> print(f"{42:b}")  # Print 42 in binary
101010
>>> print(f"{42:032b}")  # Print 42 in binary on 32 zero-padded digits
00000000000000000000000000101010
```

Alternatively, you can call `bin()` with the number as an argument:

Python

```
>>> bin(42)
'0b101010'
```

This global built-in function returns a string consisting of a **binary literal**, which starts with the prefix `0b` and is followed by ones and zeros. It always shows the minimum number of digits without the leading zeros.

You can use such literals verbatim in your code, too:

Python

```
>>> age = 0b101010
>>> print(age)
42
```

Other integer literals available in Python are the **hexadecimal** and **octal** ones, which you can obtain with the `hex()` and `oct()` functions, respectively:

Python

```
>>> hex(42)
'0x2a'
>>> oct(42)
'0o52'
```

Notice how the hexadecimal system, which is base sixteen, takes advantage of letters `A` through `F` to augment the set of available digits. The octal literals in other programming languages are usually prefixed with plain zero, which might be confusing. Python explicitly forbids such literals to avoid making a mistake:

Python

```
>>> 052
  File "<stdin>", line 1
SyntaxError: leading zeros in decimal integer literals are not permitted;
use an 0o prefix for octal integers
```

You can express the same value in different ways using any of the mentioned integer literals:

Python

```
>>> 42 == 0b101010 == 0x2a == 0o52
True
```

Choose the one that makes the most sense in context. For example, it's customary to express bitmasks with hexadecimal notation. On the other hand, the octal literal is rarely seen these days.

All numeric literals in Python are case insensitive, so you can prefix them with either lowercase or uppercase letters:

Python

```
>>> 0b101 == 0B101
True
```

This also applies to floating-point number literals that use scientific notation as well as complex number literals.

Converting Binary to `int`

Once you have your bit string ready, you can get its decimal representation by taking advantage of a binary literal:

Python

```
>>> 0b101010
42
```

This is a quick way to do the conversion while working inside the interactive Python interpreter. Unfortunately, it won't let you convert bit sequences synthesized at runtime because all literals need to be hard-coded in the source code.

> **Note:** You might be tempted to evaluate Python code with `eval("0b101010")`, but that's an easy way to compromise the security of your program, so don't do it!

Calling `int()` with two arguments will work better in the case of dynamically generated bit strings:

Python

```
>>> int("101010", 2)
42
>>> int("cafe", 16)
51966
```

The first argument is a string of digits, while the second one determines the base of the numeral system. Unlike a binary literal, a string can come from anywhere, even a user typing on the keyboard. For a deeper look at `int()`, you can expand the box below.

Other Uses of `int()` Show/Hide

So far, so good. But what about negative numbers?

Emulating the Sign Bit

When you call `bin()` on a negative integer, it merely prepends the minus sign to the bit string obtained from the corresponding positive value:

Python

```
>>> print(bin(-42), bin(42), sep="\n ")
-0b101010
 0b101010
```

Changing the sign of a number doesn't affect the underlying bit string in Python. Conversely, you're allowed to prefix a bit string with the minus sign when transforming it to decimal form:

Python

```
>>> int("-101010", 2)
-42
```

That makes sense in Python because, internally, it doesn't use the sign bit. You can think of the sign of an integer number in Python as a piece of information stored separately from the modulus.

However, there are a few workarounds that let you emulate fixed-length bit sequences containing the sign bit:

- Bitmask
- Modulo operation (`%`)
- `ctypes` module
- `array` module
- `struct` module

You know from earlier sections that to ensure a certain bit-length of a number, you can use a nifty bitmask. For example, to keep one byte, you can use a mask composed of exactly eight turned-on bits:

Python

```
>>> mask = 0b11111111  # Same as 0xff or 255
>>> bin(-42 & mask)
'0b11010110'
```

Masking forces Python to temporarily change the number's representation from sign-magnitude to two's complement and then back again. If you forget about the decimal value of the resulting binary literal, which is equal to 214_{10}, then it'll represent -42_{10} in two's complement. The leftmost bit will be the sign bit.

Alternatively, you can take advantage of the modulo operation that you used previously to simulate the logical right shift in Python:

Python

```
>>> bin(-42 % (1 << 8))  # Give me eight bits
'0b11010110'
```

If that looks too convoluted for your taste, then you can use one of the modules from the standard library that express the same intent more clearly. For example, using `ctypes` will have an identical effect:

Python

```
>>> from ctypes import c_uint8 as unsigned_byte
>>> bin(unsigned_byte(-42).value)
'0b11010110'
```

You've seen it before, but just as a reminder, it'll piggyback off the unsigned integer types from C.

Another standard module that you can use for this kind of conversion in Python is the `array` module. It defines a data structure that's similar to a `list` but is only allowed to hold elements of the same numeric type. When declaring an array, you need to indicate its type up front with a corresponding letter:

Python

```
>>> from array import array
>>> signed = array("b", [-42, 42])
>>> unsigned = array("B")
>>> unsigned.frombytes(signed.tobytes())
>>> unsigned
array('B', [214, 42])
>>> bin(unsigned[0])
'0b11010110'
>>> bin(unsigned[1])
'0b101010'
```

For example, `"b"` stands for an 8-bit signed byte, while `"B"` stands for its unsigned equivalent. There are a few other predefined types, such as a signed 16-bit integer or a 32-bit floating-point number.

Copying raw bytes between these two arrays changes how bits are interpreted. However, it takes twice the amount of memory, which is quite wasteful. To perform such a bit rewriting in place, you can rely on the `struct` module, which uses a similar set of format characters for type declarations:

Python

```
>>> from struct import pack, unpack
>>> unpack("BB", pack("bb", -42, 42))
(214, 42)
>>> bin(214)
'0b11010110'
```

Packing lets you lay objects in memory according to the given C data type specifiers. It returns a read-only `bytes()` object, which contains raw bytes of the resulting block of memory. Later, you can read back those bytes using a different set of type codes to change how they're translated into Python objects.

Up to this point, you've used different techniques to obtain fixed-length bit strings of integers expressed in two's complement representation. If you want to convert these types of bit sequences back to Python integers instead, then you can try this function:

Python

```
def from_twos_complement(bit_string, num_bits=32):
    unsigned = int(bit_string, 2)
    sign_mask = 1 << (num_bits - 1)  # For example 0b100000000
    bits_mask = sign_mask - 1        # For example 0b011111111
    return (unsigned & bits_mask) - (unsigned & sign_mask)
```

The function accepts a string composed of binary digits. First, it converts the digits to a plain unsigned integer, disregarding the sign bit. Next, it uses two bitmasks to extract the sign and magnitude bits, whose locations depend on the specified bit-length. Finally, it combines them using regular arithmetic, knowing that the value associated with the sign bit is negative.

You can try it out against the trusty old bit string from earlier examples:

Python

```
>>> int("11010110", 2)
214
>>> from_twos_complement("11010110")
214
>>> from_twos_complement("11010110", num_bits=8)
-42
```

Python's `int()` treats all the bits as the magnitude, so there are no surprises there. However, this new function assumes a 32-bit long string by default, which means the sign bit is implicitly equal to zero for shorter strings. When you request a bit-length that matches your bit string, then you'll get the expected result.

While integer is the most appropriate data type for working with bitwise operators in most cases, you'll sometimes need to extract and manipulate fragments of structured binary data, such as image pixels. The `array` and `struct` modules briefly touch upon this topic, so you'll explore it in more detail next.

Seeing Data in Binary

You know how to read and interpret individual bytes. However, real-world data often consists of more than one byte to convey information. Take the `float` data type as an example. A single floating-point number in Python occupies as many as eight bytes in memory.

How do you see those bytes?

You can't simply use bitwise operators because they don't work with floating-point numbers:

Python

```
>>> 3.14 & 0xff
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for &: 'float' and 'int'
```

You have to forget about the particular data type you're dealing with and think of it in terms of a generic stream of bytes. That way, it won't matter what the bytes represent outside the context of them being processed by the bitwise operators.

To get the `bytes()` of a floating-point number in Python, you can pack it using the familiar `struct` module:

Python

```
>>> from struct import pack
>>> pack(">d", 3.14159)
b'@\t!\xf9\xf0\x1b\x86n'
```

Ignore the format characters passed through the first argument. They won't make sense until you get to the byte order section below. Behind this rather obscure textual representation hides a list of eight integers:

Python

```
>>> list(b"@\t!\xf9\xf0\x1b\x86n")
[64, 9, 33, 249, 240, 27, 134, 110]
```

Their values correspond to the subsequent bytes used to represent a floating-point number in binary. You can combine them to produce a very long bit string:

Python

```
>>> from struct import pack
>>> "".join([f"{b:08b}" for b in pack(">d", 3.14159)])
'0100000000001001001000011111100111111000000011011100001100110111 0'
```

These 64 bits are the sign, exponent, and mantissa in double precision that you read about earlier. To synthesize a `float` from a similar bit string, you can reverse the process:

Python

```
>>> from struct import unpack
>>> bits = "0100000000001001001000011111100111111000000011011100001100110111O"
>>> unpack(
...   ">d",
...   bytes(int(bits[i:i+8], 2) for i in range(0, len(bits), 8))
... )
(3.14159,)
```

`unpack()` returns a tuple because it allows you to read more than one value at a time. For example, you could read the same bit string as four 16-bit signed integers:

Python

```
>>> unpack(
...   ">hhhh",
...   bytes(int(bits[i:i+8], 2) for i in range(0, len(bits), 8))
... )
(16393, 8697, -4069, -31122)
```

As you can see, the way a bit string should be interpreted must be known up front to avoid ending up with garbled data. One important question you need to ask yourself is which end of the byte stream you should start reading from—left or right. Read on to find out.

Byte Order

There's no dispute about the order of bits in a single byte. You'll always find the least-significant bit at index zero and the most-significant bit at index seven, regardless of how they're physically laid out in memory. The bitwise shift operators rely on this consistency.

However, there's no consensus for the **byte order** in multibyte chunks of data. A piece of information comprising more than one byte can be read from left to right like an English text or from right to left like an Arabic one, for example. Computers see bytes in a binary stream like humans see words in a sentence.

It doesn't matter which direction computers choose to read the bytes from as long as they apply the same rules everywhere. Unfortunately, different computer architectures use different approaches, which makes transferring data between them challenging.

Big-Endian vs Little-Endian

Let's take a 32-bit unsigned integer corresponding to the number 1969_{10}, which was the year when Monty Python first appeared on TV. With all the leading zeros, it has the following binary representation $00000000000000000000011110110001_2$.

How would you store such a value in computer memory?

If you imagine memory as a one-dimensional tape consisting of bytes, then you'd need to break that data down into individual bytes and arrange them in a contiguous block. Some find it natural to start from the left end because that's how they read, while others prefer starting at the right end:

Byte Order	Address N	Address N+1	Address N+2	Address N+3
Big-Endian	00000000_2	00000000_2	00000111_2	10110001_2
Little-Endian	10110001_2	00000111_2	00000000_2	00000000_2

When bytes are placed from left to right, the most-significant byte is assigned to the lowest memory address. This is known as the **big-endian** order. Conversely, when bytes are stored from right to left, the least-significant byte comes first. That's called **little-endian** order.

> **Note:** These humorous names draw inspiration from the eighteenth-century novel *Gulliver's Travels* by Jonathan Swift. The author describes a conflict between the Little-Endians and the Big-Endians over the correct way to break the shell of a boiled egg. While Little-Endians prefer to start with the little pointy end, Big-Endians like the big end more.

Which way is better?

From a practical standpoint, there's no real advantage of using one over the other. There might be some marginal gains in performance at the hardware level, but you won't notice them. Major network protocols use the big-endian order, which allows them to filter data packets more quickly given the hierarchical design of IP addressing. Other than that, some people may find it more convenient to work with a particular byte order when debugging.

Either way, if you don't get it right and mix up the two standards, then bad things start to happen:

Python

```
>>> raw_bytes = (1969).to_bytes(length=4, byteorder="big")
>>> int.from_bytes(raw_bytes, byteorder="little")
2970025984
>>> int.from_bytes(raw_bytes, byteorder="big")
1969
```

When you serialize some value to a stream of bytes using one convention and try reading it back with another, you'll get a completely useless result. This scenario is most likely when data is sent over a network, but you can also experience it when reading a local file in a specific format. For example, the header of a Windows bitmap always uses little-endian, while JPEG can use both byte orders.

Native Endianness

To find out your platform's endianness, you can use the `sys` module:

Python

```
>>> import sys
>>> sys.byteorder
'little'
```

You can't change endianness, though, because it's an intrinsic feature of your **CPU architecture**. It's impossible to mock it for testing purposes without hardware virtualization such as QEMU, so even the popular VirtualBox won't help.

Notably, the x86 family of processors from Intel and AMD, which power most modern laptops and desktops, are little-endian. Mobile devices are based on low-energy ARM architecture, which is bi-endian, while some older architectures such as the ancient Motorola 68000 were big-endian only.

For information on determining endianness in C, expand the box below.

Checking the Byte Order in C — Show/Hide

Once you know the native endianness of your machine, you'll want to convert between different byte orders when manipulating binary data. A universal way to do so, regardless of the data type at hand, is to reverse a generic `bytes()` object or a sequence of integers representing those bytes:

Python

```
>>> big_endian = b"\x00\x00\x07\xb1"
>>> bytes(reversed(big_endian))
b'\xb1\x07\x00\x00'
```

However, it's often more convenient to use the `struct` module, which lets you define standard C data types. In addition to this, it allows you to request a given byte order with an optional modifier:

Python

```
>>> from struct import pack, unpack
>>> pack(">I", 1969)  # Big-endian unsigned int
b'\x00\x00\x07\xb1'
>>> unpack("<I", b"\x00\x00\x07\xb1")  # Little-endian unsigned int
(2970025984,)
```

The greater-than sign (`>`) indicates that bytes are laid out in the big-endian order, while the less-than symbol (`<`) corresponds to little-endian. If you don't specify one, then native endianness is assumed. There are a few more modifiers, like the exclamation mark (`!`), which signifies the network byte order.

Network Byte Order

Computer networks are made of heterogeneous devices such as laptops, desktops, tablets, smartphones, and even light bulbs equipped with a Wi-Fi adapter. They all need agreed-upon protocols and standards, including the byte order for binary transmission, to communicate effectively.

At the dawn of the Internet, it was decided that the byte order for those network protocols would be **big-endian**.

Programs that want to communicate over a network can grab the classic C API, which abstracts away the nitty-gritty details with a socket layer. Python wraps that API through the built-in `socket` module. However, unless you're writing a custom binary protocol, you'll probably want to take advantage of an even higher-level abstraction, such as the HTTP protocol, which is text-based.

Where the `socket` module can be useful is in the byte order conversion. It exposes a few functions from the C API, with their distinctive, tongue-twisting names:

Python

```
>>> from socket import htons, htonl, ntohs, ntohl
>>> htons(1969)  # Host to network (short int)
45319
>>> htonl(1969)  # Host to network (long int)
2970025984
>>> ntohs(45319)  # Network to host (short int)
1969
>>> ntohl(2970025984)  # Network to host (long int)
1969
```

If your host already uses the big-endian byte order, then there's nothing to be done. The values will remain the same.

Bitmasks

A bitmask works like a graffiti stencil that blocks the paint from being sprayed on particular areas of a surface. It lets you isolate the bits to apply some function on them selectively. Bitmasking involves both the bitwise logical operators and the bitwise shift operators that you've read about.

You can find bitmasks in a lot of different contexts. For example, the subnet mask in IP addressing is actually a bitmask that helps you extract the network address. Pixel channels, which correspond to the red, green, and blue colors in the RGB model, can be accessed with a bitmask. You can also use a bitmask to define Boolean flags that you can then pack on a bit field.

There are a few common types of operations associated with bitmasks. You'll take a quick look at some of them below.

Getting a Bit

To read the value of a particular bit on a given position, you can use the bitwise AND against a bitmask composed of only one bit at the desired index:

Python

```
>>> def get_bit(value, bit_index):
...     return value & (1 << bit_index)
...
>>> get_bit(0b10000000, bit_index=5)
0
>>> get_bit(0b10100000, bit_index=5)
32
```

The mask will suppress all bits except for the one that you're interested in. It'll result in either zero or a power of two with an exponent equal to the bit index. If you'd like to get a simple yes-or-no answer instead, then you could shift to the right and check the least-significant bit:

Python

```
>>> def get_normalized_bit(value, bit_index):
...     return (value >> bit_index) & 1
...
>>> get_normalized_bit(0b10000000, bit_index=5)
0
>>> get_normalized_bit(0b10100000, bit_index=5)
1
```

This time, it will normalize the bit value so that it never exceeds one. You could then use that function to derive a Boolean `True` or `False` value rather than a numeric value.

Setting a Bit

Setting a bit is similar to getting one. You take advantage of the same bitmask as before, but instead of using bitwise AND, you use the bitwise OR operator:

Python

```
>>> def set_bit(value, bit_index):
...     return value | (1 << bit_index)
...
>>> set_bit(0b10000000, bit_index=5)
160
>>> bin(160)
'0b10100000'
```

The mask retains all the original bits while enforcing a binary one at the specified index. Had that bit already been set, its value wouldn't have changed.

Unsetting a Bit

To clear a bit, you want to copy all binary digits while enforcing zero at one specific index. You can achieve this effect by using the same bitmask once again, but in the inverted form:

Python

```
>>> def clear_bit(value, bit_index):
...     return value & ~(1 << bit_index)
...
>>> clear_bit(0b11111111, bit_index=5)
223
>>> bin(223)
'0b11011111'
```

Using the bitwise NOT on a positive number always produces a negative value in Python. While this is generally undesirable, it doesn't matter here because you immediately apply the bitwise AND operator. This, in turn, triggers the mask's conversion to two's complement representation, which gets you the expected result.

Toggling a Bit

Sometimes it's useful to be able to toggle a bit on and off again periodically. That's a perfect opportunity for the bitwise XOR operator, which can flip your bit like that:

Python

```
>>> def toggle_bit(value, bit_index):
...     return value ^ (1 << bit_index)
...
>>> x = 0b10100000
>>> for _ in range(5):
...     x = toggle_bit(x, bit_index=7)
...     print(bin(x))
...
0b100000
0b10100000
0b100000
0b10100000
0b100000
```

Notice the same bitmask being used again. A binary one on the specified position will make the bit at that index invert its value. Having binary zeros on the remaining places will ensure that the rest of the bits will be copied.

Bitwise Operator Overloading

The primary domain of bitwise operators is integer numbers. That's where they make the most sense. However, you've also seen them used in a Boolean context, in which they replaced the logical operators. Python provides alternative implementations for some of its operators and lets you overload them for new data types.

Although the proposal to overload the logical operators in Python was rejected, you can give new meaning to any of the bitwise operators. Many popular libraries, and even the standard library, take advantage of it.

Built-In Data Types

Python bitwise operators are defined for the following built-in data types:

- `int`
- `bool`
- `set` and `frozenset`
- `dict` (since Python 3.9)

It's not a widely known fact, but bitwise operators can perform operations from **set algebra**, such as union, intersection, and symmetric difference, as well as merge and update **dictionaries**.

> **Note:** At the time of writing, Python 3.9 hadn't been released, but you could take a sneak peek at the upcoming language features using Docker or pyenv.

When `a` and `b` are Python sets, then bitwise operators correspond to the following methods:

Set Method	Bitwise Operator
`a.union(b)`	`a | b`
`a.update(b)`	`a |= b`
`a.intersection(b)`	`a & b`
`a.intersection_update(b)`	`a &= b`

Set Method	Bitwise Operator
`a.symmetric_difference(b)`	`a ^ b`
`a.symmetric_difference_update(vegies)`	`a ^= b`

They do virtually the same thing, so it's up to you which syntax to use. Apart from that, there's also an overloaded minus operator (-), which implements a difference of two sets. To see them in action, assume you have the following two sets of fruits and vegetables:

Python

```
>>> fruits = {"apple", "banana", "tomato"}
>>> veggies = {"eggplant", "tomato"}
>>> fruits | veggies
{'tomato', 'apple', 'eggplant', 'banana'}
>>> fruits & veggies
{'tomato'}
>>> fruits ^ veggies
{'apple', 'eggplant', 'banana'}
>>> fruits - veggies  # Not a bitwise operator!
{'apple', 'banana'}
```

They share one common member, which is hard to classify, but the rest of their elements are disjoint.

One thing to watch out for is the immutable `frozenset()`, which is missing the methods for in-place updates. However, when you use their bitwise operator counterparts, the meaning is slightly different:

Python

```
>>> const_fruits = frozenset({"apple", "banana", "tomato"})
>>> const_veggies = frozenset({"eggplant", "tomato"})
>>> const_fruits.update(const_veggies)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    const_fruits.update(const_veggies)
AttributeError: 'frozenset' object has no attribute 'update'
>>> const_fruits |= const_veggies
>>> const_fruits
frozenset({'tomato', 'apple', 'eggplant', 'banana'})
```

It looks like `frozenset()` isn't so immutable after all when you use the bitwise operators, but the devil is in the details. Here's what actually happens:

Python

```
const_fruits = const_fruits | const_veggies
```

The reason it works the second time is that you don't change the original immutable object. Instead, you create a new one and assign it to the same variable again.

Python `dict` supports only bitwise OR, which works like a union operator. You can use it to update a dictionary in place or merge two dictionaries into a new one:

Python

```
>>> fruits = {"apples": 2, "bananas": 5, "tomatoes": 0}
>>> veggies = {"eggplants": 2, "tomatoes": 4}
>>> fruits | veggies  # Python 3.9+
{'apples': 2, 'bananas': 5, 'tomatoes': 4, 'eggplants': 2}
>>> fruits |= veggies  # Python 3.9+, same as fruits.update(veggies)
```

The augmented version of the bitwise operator is equivalent to `.update()`.

Third-Party Modules

Many popular libraries, including NumPy, pandas, and SQLAlchemy, overload the bitwise operators for their specific data types. This is the most likely place you'll find bitwise operators in Python because they aren't used very often in their original meaning anymore.

For example, NumPy applies them to vectorized data in a pointwise fashion:

Python

```
>>> import numpy as np
>>> np.array([1, 2, 3]) << 2
array([ 4,  8, 12])
```

This way, you don't need to manually apply the same bitwise operator to each element of the array. But you can't do the same thing with ordinary lists in Python.

pandas uses NumPy behind the scenes, and it also provides overloaded versions of the bitwise operators for its `DataFrame` and `Series` objects. However, they behave as you'd expect. The only difference is that they do their usual job on vectors and matrices of numbers instead of on individual scalars.

Things get more interesting with libraries that give the bitwise operators entirely new meanings. For example, SQLAlchemy provides a compact syntax for querying the database:

Python

```
session.query(User) \
       .filter((User.age > 40) & (User.name == "Doe")) \
       .all()
```

The bitwise AND operator (`&`) will eventually translate to a piece of SQL query. However, that's not very obvious, at least not to my IDE, which complains about the **unpythonic** use of bitwise operators when it sees them in this type of expression. It immediately suggests replacing every occurrence of `&` with a logical `and`, not knowing that doing so would make the code stop working!

This type of operator overloading is a controversial practice that relies on implicit magic you have to know up front. Some programming languages like Java prevent such abuse by disallowing operator overloading altogether. Python is more liberal in that regard and trusts that you know what you're doing.

Custom Data Types

To customize the behavior of Python's bitwise operators, you have to define a class and then implement the corresponding magic methods in it. At the same time, you can't redefine the behavior of the bitwise operators for the existing types. Operator overloading is possible only on new data types.

Here's a quick rundown of special methods that let you overload the bitwise operators:

Magic Method	Expression
`.__and__(self, value)`	`instance & value`
`.__rand__(self, value)`	`value & instance`
`.__iand__(self, value)`	`instance &= value`
`.__or__(self, value)`	`instance | value`
`.__ror__(self, value)`	`value | instance`
`.__ior__(self, value)`	`instance |= value`
`.__xor__(self, value)`	`instance ^ value`

Magic Method	Expression
`.__rxor__(self, value)`	`value ^ instance`
`.__ixor__(self, value)`	`instance ^= value`
`.__invert__(self)`	`~instance`
`.__lshift__(self, value)`	`instance << value`
`.__rlshift__(self, value)`	`value << instance`
`.__ilshift__(self, value)`	`instance <<= value`
`.__rshift__(self, value)`	`instance >> value`
`.__rrshift__(self, value)`	`value >> instance`
`.__irshift__(self, value)`	`instance >>= value`

You don't need to define all of them. For example, to have a slightly more convenient syntax for appending and prepending elements to a deque, it's sufficient to implement only `.__lshift__()` and `.__rrshift__()`:

Python

```
>>> from collections import deque
>>> class DoubleEndedQueue(deque):
...     def __lshift__(self, value):
...         self.append(value)
...     def __rrshift__(self, value):
...         self.appendleft(value)
...
>>> items = DoubleEndedQueue(["middle"])
>>> items << "last"
>>> "first" >> items
>>> items
DoubleEndedQueue(['first', 'middle', 'last'])
```

This user-defined class wraps a deque to reuse its implementation and augment it with two additional methods that allow for adding items to the left or right end of the collection.

Least-Significant Bit Steganography

Whew, that was a lot to process! If you're still scratching your head, wondering why you'd want to use bitwise operators, then don't worry. It's time to showcase what you can do with them in a fun way.

To follow along with the examples in this section, you can download the source code by clicking the link below:

> **Get the Source Code: Click here to get the source code you'll use** to learn about Python's bitwise operators in this tutorial.

You're going to learn about steganography and apply this concept to secretly embed arbitrary files in bitmap images.

Cryptography vs Steganography

Cryptography is about changing a message into one that is readable only to those with the right key. Everyone else can still see the encrypted message, but it won't make any sense to them. One of the first forms of cryptography was the substitution cipher, such as the Caesar cipher named after Julius Caesar.

Steganography is similar to cryptography because it also allows you to share secret messages with your desired audience. However, instead of using encryption, it cleverly hides information in a medium that doesn't attract attention. Examples include using invisible ink or writing an acrostic in which the first letter of every word or line forms a secret message.

Unless you knew that a secret message was concealed and the method to recover it, you'd probably ignore the carrier. You can combine both techniques to be even safer, hiding an encrypted message rather than the original one.

There are plenty of ways to smuggle secret data in the digital world. In particular, file formats carrying lots of data, such as audio files, videos, or images, are a great fit because they give you a lot of room to work with. Companies that release copyrighted material might use steganography to watermark individual copies and trace the source of a leak, for example.

Below, you'll inject secret data into a plain **bitmap**, which is straightforward to read and write in Python without the need for external dependencies.

Bitmap File Format

The word *bitmap* usually refers to the Windows bitmap (`.bmp`) file format, which supports a few alternative ways of representing pixels. To make life easier, you're going to assume that pixels are stored in 24-bit uncompressed RGB (red, green, and blue) format. A pixel will have three color channels that can each hold values from 0_{10} to 255_{10}.

Every bitmap begins with a **file header**, which contains metadata such as the image width and height. Here are a few interesting fields and their positions relative to the start of the header:

Field	Byte Offset	Bytes Length	Type	Sample Value
Signature	`0x00`	2	String	BM
File Size	`0x02`	4	Unsigned int	7,629,186
Reserved #1	`0x06`	2	Bytes	0
Reserved #2	`0x08`	2	Bytes	0
Pixels Offset	`0x0a`	4	Unsigned int	122
Pixels Size	`0x22`	4	Unsigned int	7,629,064
Image Width	`0x12`	4	Unsigned int	1,954
Image Height	`0x16`	4	Unsigned int	1,301
Bits Per Pixel	`0x1c`	2	Unsigned short	24
Compression	`0x1e`	4	Unsigned int	0
Colors Palette	`0x2e`	4	Unsigned int	0

You can infer from this header that the corresponding bitmap is 1,954 pixels wide and 1,301 pixels high. It doesn't use compression, nor does it have a color palette. Every pixel occupies 24 bits, or 3 bytes, and the raw pixel data starts at offset 122_{10}.

You can open the bitmap in **binary mode**, seek the desired offset, read the given number of bytes, and deserialize them using `struct` like before:

Python

```
from struct import unpack

with open("example.bmp", "rb") as file_object:
    file_object.seek(0x22)
    field: bytes = file_object.read(4)
    value: int = unpack("<I", field)[0]
```

Note that all integer fields in bitmaps are stored in the little-endian byte order.

You might have noticed a small discrepancy between the number of pixel bytes declared in the header and the one that would result from the image size. When you multiply 1,954 pixels × 1,301 pixels × 3 bytes, you get a value that is 2,602 bytes less than 7,629,064.

This is because pixel bytes are **padded** with zeros so that every row is a multiple of four bytes. If the width of the image times three bytes happens to be a multiple of four, then there's no need for padding. Otherwise, empty bytes are added at the end of every row.

> **Note:** To avoid raising suspicion, you'll need to take that padding into account by skipping the empty bytes. Otherwise, it would be a clear giveaway to someone who knows what to look for.

Bitmaps store pixel rows upside down, starting from the bottom rather than the top. Also, every pixel is serialized to a vector of color channels in a somewhat odd BGR order rather than RGB. However, this is irrelevant to the task of hiding secret data.

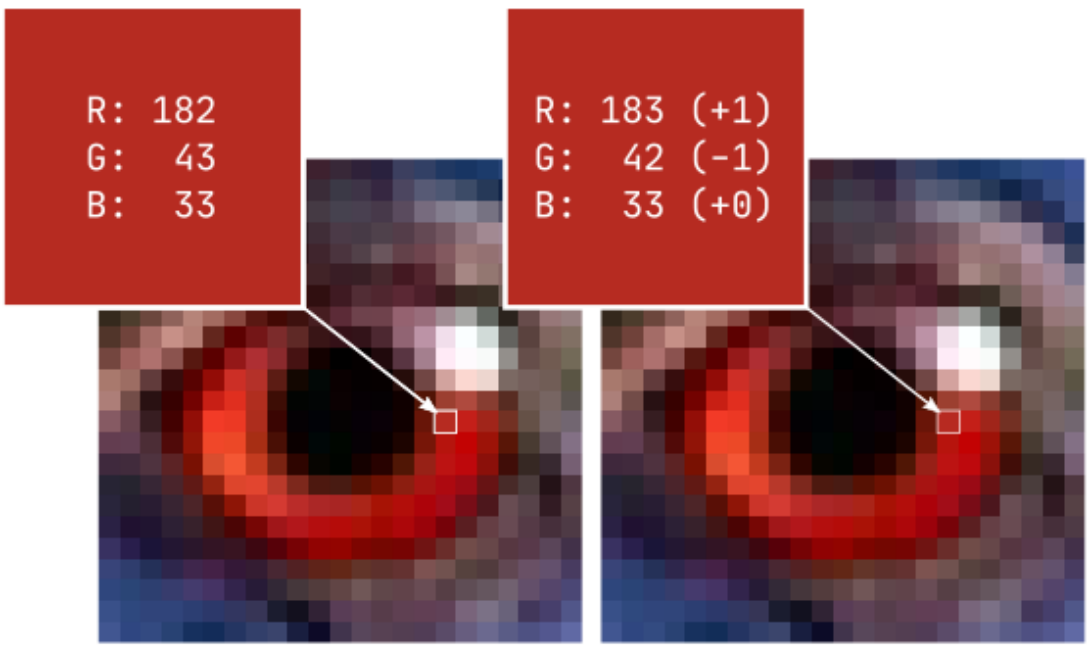
Bitwise Hide and Seek

You can use the bitwise operators to spread custom data over consecutive pixel bytes. The idea is to overwrite the least-significant bit in each of them with bits coming from the next secret byte. This will introduce the least amount of noise, but you can experiment with adding more bits to strike a balance between the size of injected data and pixel distortion.

> **Note:** Using the least-significant bit steganography doesn't affect the file size of the resulting bitmap. It'll remain the same as the original file.

In some cases, the corresponding bits will be the same, resulting in no change in pixel value whatsoever. However, even in the worst-case scenario, a pixel color will differ only by a fraction of a percent. Such a tiny anomaly will remain invisible to the human eye but can be detected with **steganalysis**, which uses statistics.

Take a look at these cropped images:



The one on the left comes from the original bitmap, while the image on the right depicts a processed bitmap with an embedded video stored on the least-significant bits. Can you spot the difference?

The following piece of code encodes the secret data onto the bitmap:

Python

```
for secret_byte, eight_bytes in zip(file.secret_bytes, bitmap.byte_slices):
    secret_bits = [(secret_byte >> i) & 1 for i in reversed(range(8))]
    bitmap[eight_bytes] = bytes(
        [
            byte | 1 if bit else byte & ~1
            for byte, bit in zip(bitmap[eight_bytes], secret_bits)
        ]
    )
```

For every byte of secret data and the corresponding eight bytes of pixel data, excluding the pad bytes, it prepares a list of bits to be spread over. Next, it overwrites the least-significant bit in each of the eight bytes using a relevant bitmask. The result is converted to a `bytes()` object and assigned back to the part of the bitmap that it originally came from.

To decode a file from the same bitmap, you need to know how many secret bytes were written to it. You could allocate a few bytes at the beginning of the data stream to store this number, or you could use the reserved fields from the bitmap header:

Python

```
@reserved_field.setter
def reserved_field(self, value: int) -> None:
    """Store a little-endian 32-bit unsigned integer."""
    self._file_bytes.seek(0x06)
    self._file_bytes.write(pack("<I", value))
```

This jumps to the right offset in the file, serializes the Python `int` to raw bytes, and writes them down.

You might also want to store the name of your secret file. Since it can have an arbitrary length, it makes sense to serialize it using a null-terminated string, which would precede the file contents. To create such a string, you need to encode a Python `str` object to bytes and manually append the null byte at the end:

Python

```
>>> from pathlib import Path
>>> path = Path("/home/jsmith/café.pdf")
>>> path.name.encode("utf-8") + b"\x00"
b'caf\xc3\xa9.pdf\x00'
```

Also, it doesn't hurt to drop the redundant parent directory from the path using `pathlib`.

The sample code supplementing this article will let you **encode**, **decode**, and **erase** a secret file from the given bitmap with the following commands:

Shell

```
$ python -m stegano example.bmp -d
Extracted a secret file: podcast.mp4
$ python -m stegano example.bmp -x
Erased a secret file from the bitmap
$ python -m stegano example.bmp -e pdcast.mp4
Secret file was embedded in the bitmap
```

This is a runnable module that can be executed by calling its encompassing directory. You could also make a portable ZIP-format archive out of its contents to take advantage of the Python ZIP application support.

This program relies on modules from the standard library mentioned in the article and a few others that you might not have heard about before. A critical module is `mmap`, which exposes a Python interface to **memory-mapped** files. They let you manipulate huge files using both the standard file API and the sequence API. It's as if the file were one big mutable list that you could slice.

Go ahead and play around with the bitmap attached to the supporting materials. It contains a little surprise for you!

Conclusion

Mastering Python bitwise operators gives you the ultimate freedom to manipulate **binary data** in your projects. You now know their **syntax** and different flavors as well as the data types that support them. You can also customize their behavior for your own needs.

In this tutorial, you learned how to:

- Use Python **bitwise operators** to manipulate individual bits
- Read and write binary data in a **platform-agnostic** way
- Use **bitmasks** to pack information on a single byte

- **Overload** Python bitwise operators in custom data types
- Hide **secret messages** in digital images

You also learned how computers use the **binary system** to represent different kinds of digital information. You saw several popular ways to interpret bits and how to mitigate the lack of **unsigned** data types in Python as well as Python's unique way of storing integer numbers in memory.

With this information, you're ready to make full use of binary data in your code. To download the source code used in the watermarking example and continue experimenting with bitwise operators, you can click the link below:

Get the Source Code: Click here to get the source code you'll use to learn about Python's bitwise operators in this tutorial.

About **Bartosz Zaczyński**

Bartosz is a bootcamp instructor, author, and polyglot programmer in love with Python. He helps his students get into software engineering by sharing over a decade of commercial experience in the IT industry.