

3.2: Integers and Floating-Point Numbers

Digital computers store all data in the form of bits (ones and zeros), and a number is typically stored as a sequence of bits of *fixed length*. For example, a number labeled x might refer to a sequence of eight bits starting from some specific address, as shown in the following figure:

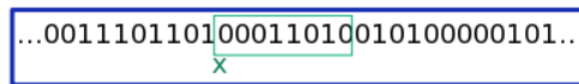


Figure 3.2.1: An eight-bit number stored in a variable x .

The green box indicates the set of memory addresses, eight bits long, where the number is stored. (The other bits, to the left and right of this box, might be used by the computer for other purposes, or simply unused.) What actual number does this sequence of eight bits represent? It depends: there are two types of formats for storing numbers, called **integers** and **floating-point numbers**.

3.2.1 Integers

In integer format, one of the bits is used to denote the sign of the number (positive or negative), and the remaining bits specify an integer in a binary representation. Because only a fixed number of bits is available, only a finite range of integers can be represented. On modern 64-bit computers, integers are typically stored using 64 bits, so only 2^{64} distinct integers can be represented. The minimum and maximum integers are

$$n_{\min} = -2^{63} = -9223372036854775808 \quad (3.2.1)$$

$$n_{\max} = 2^{63} - 1 = +9223372036854775807 \quad (3.2.2)$$

Python is somewhat able to conceal this limitation through a "variable-width integer" feature. If the numbers you're dealing with exceed the above limits, it can convert the numbers to a different format, in which the number of bits can be arbitrarily larger than 64 bits. However, these variable-width integers have various limitations. For one thing, they cannot be stored in a Scipy array meant for storing standard fixed-width integers:

```
>>> n = array([-2**63])           # Create an integer array.
>>> n                             # Check the array's contents.
array([-9223372036854775808])
>>> n.dtype                       # Check the array's data type: it should store
dtype('int64')
>>> m = -2**63 - 1                # If we specify an integer that's too negative
>>> m
-9223372036854775809
>>> n[0] = m                      # What if we try to store it in a 64-bit integer?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: Python int too large to convert to C long
```

Moreover, performing arithmetic operations on variable-width integers is much slower than standard fixed-width arithmetic. Therefore, it's generally best to avoid dealing with integers that are too large or too small.

3.2.2 Floating-Point Numbers

Floating-point numbers (sometimes called **floats**) are a format for approximately representing "real numbers"—numbers that are not necessarily integers. Each number is broken up into three components: the "sign", "fraction", and "exponent". For example, the charge of an electron can be represented as

$$q \approx -1.60217657 \times 10^{-19} = (-)(160217657) \times 10^{-27} \quad (3.2.3)$$

This number can thus be stored as one bit (the sign $-$), and two integers (the fraction 160217646 and the exponent -27). The two integers are stored in fixed-width format, so the overall floating-point number also has fixed width (64 bits in modern computers). The actual implementation details of floating-point numbers differs from the above example in a few ways—notably, the numbers are represented using powers of 2 rather than powers of 10—but this is the basic idea.

In Python code, floating-point numbers are specified in decimal notation (e.g. `0.000001`) or exponential notation (e.g. `1e-6`). If you want to represent an integer in floating-point format, give it a trailing decimal point, like this: `3.` (or, equivalently, `3.0`). If you perform arithmetic between an integer and a floating-point number, like `2 + 3.0`, the result is a floating-point number. And if you divide two integers, the result is a floating-point number:

```
>>> a, b = 6, 3    # a and b are both integers
>>> a/b            # a/b is in floating-point format
2.0
```

Note

In Python 2, dividing two integers yielded a rounded integer. This is a frequent source of bugs, so be aware of this behavior if you ever use Python 2.

Numerical Imprecision

Because floating-point numbers use a finite number of bits, the vast majority of real numbers cannot be *exactly* represented in floating-point format. They can only be approximated. This gives rise to interesting quirks, like this:

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

The reason is that the decimal number 0.1 does not correspond exactly to a floating-point representation, so when you tell the computer to create a number "0.1", it instead approximates that number using a floating-point number whose decimal value is 0.1000000000000000055511512...

Because of this lack of precision, you should not compare floating-point numbers using Python's equality operator `==` :

```
>>> x = 0.1 + 0.1 + 0.1
>>> y = 0.3
>>> x == y
False
```

Instead of using `==`, you can compare `x` and `y` by checking if they're closer than a certain amount:

```
>>> epsilon = 1e-6
>>> abs(x-y) < epsilon
True
```

The "density" of real numbers represented exactly floating point numbers decreases exponentially with the magnitude of the number. Hence, large floating-point numbers are less precise than small ones. For this reason, numerical algorithms (such as Gaussian elimination) often try to avoid multiplying by very large numbers, or dividing by very small numbers.

Special Values

Like integers, floating-point numbers have a maximum and minimum number that can be represented (this is unavoidable, since they have only a finite number of bits). Any number above the maximum is assigned a special value, `inf` (infinity):

```
>>> 1e308
1e+308
>>> 1e309
inf
```

Similarly, any number below the floating-point minimum is represented by `-inf`. There is another special value called `nan` (not-a-number), which represents the results of calculations which don't make sense, like "infinity minus infinity":

```
>>> x = 1e310
>>> x
inf
>>> y = x - x
>>> y
nan
```

If you ever need to check if a number is `inf`, you can use Scipy's `isinf` function. You can check for `nan` using Scipy's `isnan` function:

```
>>> isinf(x)
True
>>> isnan(y)
True
>>> isnan(x)
False
```

3.2.3 Complex Numbers

Complex numbers are not a fundamental data type. Python implements each complex number as a composite of two floating-point numbers (the real and imaginary parts). You can specify a complex number using the notation `X+Yj`:

```
>>> z = 2+1j
>>> z
(2+1j)
```

Python's arithmetic operations can handle arithmetic operations on complex numbers:

```
>>> u = 3.4-1.2j
>>> z * u
(8+1j)
>>> z/u
(0.4307692307692308+0.44615384615384623j)
```

You can retrieve the real and imaginary parts of a complex number using the `.real` and `.imag` slots:

```
>>> z.real
2.0
>>> z.imag
1.0
```

Alternatively, you can use Scipy's `real` and `imag` functions (which also work on arrays). Similarly, the `absolute` (or `abs`) and `angle` functions return the magnitude and argument of a complex number:

```
>>> z = 2+1j
>>> real(z)
array(2.0)
>>> imag(z)
array(1.0)
>>> absolute(z)
2.2360679774997898
>>> angle(z)
0.46364760900080609
```

This page titled [3.2: Integers and Floating-Point Numbers](#) is shared under a [CC BY-SA 4.0](#) license and was authored, remixed, and/or curated by [Y. D. Chong](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.