Numbers on a computer

Chris Paciorek

2025-08-12

Table of contents

Overview

References:

- Gentle, Computational Statistics, Chapter 2.
- http://www.lahey.com/float.htm
- And for more gory detail, see Monahan, Chapter 2.

A quick note that, as we've already seen, Python's version of scientific notation is `XeY`, which means $X \cdot 10^Y$.

A second note is that the concepts developed here apply outside of Python, but we'll illustrate the principles of computer numbers using Python. Python usually makes use of the *double* type (8 bytes)

in C for the underlying representation of real-valued numbers in C variables, so what we'll really be seeing is how such types behave in C on most modern machines. It's actually a bit more complicated in that one can use real-valued numbers that use something other than 8 bytes in numpy by specifying a `dtype`.

The handling of integers is even more complicated. In numpy, the default is 8 byte integers, but other integer dtypes are available. And in Python itself, integers can be arbitrarily large.

1. Basic representations

Everything in computer memory or on disk is stored in terms of bits. A *bit* is essentially a switch than can be either on or off. Thus everything is encoded as numbers in base 2, i.e., 0s and 1s. 8 bits make up a *byte*. As discussed in Unit 2, for information stored as plain text (ASCII), each byte is used to encode a single character (as previously discussed, actually only 7 of the 8 bits are actually used, hence there are $2^7 = 128$ ASCII characters). One way to represent a byte is to write it in hexadecimal, rather than as 8 0/1 bits. Since there are $2^8 = 256$ possible values in a byte, we can represent it more compactly as 2 base-16 numbers, such as "3e" or "a0" or "ba". A file format is nothing more than a way of interpreting the bytes in a file.

We'll create some helper functions to all us to look at the underlying binary representation.

```
from bitstring import Bits

def bits(x, type='float', len=64):
    if type == 'float':
        obj = Bits(float = x, length = len)
    elif type == 'int':
        obj = Bits(int = x, length = len)
    else:
        return None
    return(obj.bin)

def dg(x, form = '.20f'):
    print(format(x, form))
```

Note that 'b' is encoded as one more than 'a', and similarly for '0', '1', and '2'. We could check these against, say, the Wikipedia table that shows the ASCII encoding.

```
Bits(bytes=b'a').bin
```

```
'01100001'
```

```
Bits(bytes=b'b').bin
```

```
'01100010'
```

```
Bits(bytes=b'0').bin
```

```
'00110000'
```

```
Bits(bytes=b'1').bin
```

```
'00110001'
```

```
Bits(bytes=b'2').bin
```

```
'00110010'
```

```
Bits(bytes=b'@').bin
```

```
'01000000'
```

We can think about how we'd store an integer in terms of bytes. With two bytes (16 bits), we could encode any value from $0, \ldots, 2^{16} - 1 = 65535$. This is an *unsigned* integer representation. To store negative numbers as well, we can use one bit for the sign, giving us the ability to encode -32767 - 32767 ($\pm 2^{15} - 1$).

Note that in general, rather than be stored simply as the sign and then a number in base 2, integers (at least the negative ones) are actually stored in different binary encoding to facilitate arithmetic.

Here's what a 64-bit integer representation the actual bits.

```
np.binary_repr(0, width=64)
```

```
'0000000000000000000000000000000000000000000000000000000000000000'
```

```
np.binary_repr(1, width=64)
```

```
'0000000000000000000000000000000000000000000000000000000000000001'
```

```
np.binary_repr(2, width=64)
```

```
'0000000000000000000000000000000000000000000000000000000000000010'
```

```
np.binary_repr(-1, width=64)
```

```
'1111111111111111111111111111111111111111111111111111111111111111'
```

What do I mean about facilitating arithmetic? As an example, consider adding the binary representations of -1 and 1. Nice, right?

Finally note that the set of computer integers is not closed under arithmetic. We get an overflow (i.e., a result that is too large to be stored as an integer of the particular length):

```
a = np.int32(3423333)
a * a       # overflows
```

```
<string>:1: RuntimeWarning: overflow encountered in scalar multiply
np.int32(-1756921895)
```

```
a = np.int64(3423333)
a * a       # doesn't overflow if we use 64 bit int
```

```
np.int64(11719208828889)
```

This is disconcerting behavior with numpy…:

```
a = np.int64(34233332342343)
a * a
```

```
np.int64(1001093889201452977)
```

```
a=np.int64(10000000000)
a *a
```

```
np.int64(7766279631452241920)
```

```
a = 34233332342343
a * a
```

```
1171921043261307270950729649
```

That said, if we use Python's `int` rather than numpy's integers, we don't get overflow. But we do use more than 8 bytes that would be used by numpy. And if we use Python `int` or lists of such values, we're not set up for efficient array-based computation.

```
a = 34233332342343
a * a
```

```
1171921043261307270950729649
```

```
sys.getsizeof(a)
```

```
32
```

```
sys.getsizeof(a*a)
```

```
36
```

In C, one generally works with 8 byte real-valued numbers (aka *floating point* numbers or *floats*). However, many years ago, an initial standard representation used 4 bytes. Then people started using 8 bytes, which became known as *double precision floating points* or *doubles*, whereas the 4-byte version became known as *single precision*. Now with GPUs, single precision is often used for speed and reduced memory use.

Let's see how this plays out in terms of memory use in Python.

```
x = np.random.normal(size = 100000)
sys.getsizeof(x)
```

```
800112
```

```
x = np.array(np.random.normal(size = 100000), dtype = "float32")
sys.getsizeof(x)
```

```
400112
```

```
x = np.array(np.random.normal(size = 100000), dtype = "float16")
sys.getsizeof(x)
```

200112

We can easily calculate the number of megabytes (MB) a vector of floating points (in double precision) will use as the number of elements times 8 (bytes/double) divided by 10^6 to convert from bytes to megabytes. (In some cases when considering computer memory, people use mebibyte (MiB), which is $1,048,576 = 2^{20} = 1024^2$ bytes (so slightly different than 10^6), and call that a megabyte – see here for more details).

Finally, `numpy` has some helper functions that can tell us about the characteristics of computer numbers on the machine that Python is running.

```
np.iinfo(np.int32)
```

```
iinfo(min=-2147483648, max=2147483647, dtype=int32)
```

```
np.iinfo(np.int64)
```

```
iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

```
np.binary_repr(2147483647, width=32)
```

```
'01111111111111111111111111111111'
```

```
np.binary_repr(-2147483648, width=32)
```

```
'10000000000000000000000000000000'
```

```
np.binary_repr(2147483648, width=32)  # strange
```

```
'10000000000000000000000000000000'
```

```
np.int32(2147483648)
```

```
OverflowError: Python integer 2147483648 out of bounds for int32
```

```
np.binary_repr(1, width=32)
```

```
'00000000000000000000000000000001'
```

```
np.binary_repr(-1, width=32)
```

```
'11111111111111111111111111111111'
```

So the max for a 32-bit (4-byte) integer is $2147483647 = 2^{31} - 1$, which is consistent with 4 bytes. Since we have both negative and positive numbers, we have $2 \cdot 2^{31} = 2^{32} = (2^8)^4$, i.e., 4 bytes, with each byte having 8 bits.

2. Floating point basics

Representing real numbers

Initial exploration

Reals (also called floating points) are stored on the computer as an approximation, albeit a very precise approximation. As an example, if we represent the distance from the earth to the sun using a double, the error is around a millimeter. However, we need to be very careful if we're trying to do a calculation that produces a very small (or very large number) and particularly when we want to see if numbers are equal to each other.

If you run the code here, the results may surprise you.

```
0.3 - 0.2 == 0.1
0.3
0.2
0.1 # Hmmm...

np.float64(0.3) - np.float64(0.2) == np.float64(0.1)

0.75 - 0.5 == 0.25
0.6 - 0.4 == 0.2
## any ideas what is different about those two comparisons?
```

Next, let's consider the number of digits of accuracy we have for a variety of numbers. We'll use `format` within a handy wrapper function, `dg`, defined earlier, to view as many digits as we want:

```
a = 0.3
b = 0.2
dg(a)
```

```
0.29999999999999998890
```

```
dg(b)
```

```
0.20000000000000001110
```

```
dg(a-b)
```

```
0.09999999999999997780
```

```
dg(0.1)
```

```
0.10000000000000000555
```

```
dg(1/3)
```

```
0.33333333333333331483
```

So empirically, it looks like we're accurate up to the 16th decimal place

But actually, the key is the number of digits, not decimal places.

```
dg(1234.1234)
```

```
1234.12339999999994688551
```

```
dg(1234.123412341234)
```

```
1234.123412341233915867 63
```

Notice that we can represent the result accurately only up to 16 significant digits. This suggests no need to show more than 16 significant digits and no need to print out any more when writing to a file (except that if the number is bigger than 10^{16} then we need extra digits to correctly show the magnitude of the number if not using scientific notation). And of course, often we don't need anywhere near that many.

Let's return to our comparison, `0.75-0.5 == 0.25`.

```
dg(0.75)
```

```
0.75000000000000000000
```

```
dg(0.50)
```

```
0.50000000000000000000
```

What's different about the numbers 0.75 and 0.5 compared to 0.3, 0.2, 0.1?

Machine epsilon

Machine epsilon is the term used for indicating the (relative) accuracy of real numbers and it is defined as the smallest float, x, such that $1 + x \neq 1$:

```
1e-16 + 1.0
```

```
1.0
```

```
np.array(1e-16) + np.array(1.0)
```

```
np.float64(1.0)
```

```
1e-15 + 1.0
```

```
1.000000000000001
```

```
np.array(1e-15) + np.array(1.0)
```

```
np.float64(1.000000000000001)
```

```
2e-16 + 1.0
```

```
1.0000000000000002
```

```
np.finfo(np.float64).eps
```

```
np.float64(2.220446049250313e-16)
```

```
dg(2e-16 + 1.0)
```

```
1.0000000000000022204
```

What about in single precision, e.g. on a GPU?

```
np.finfo(np.float32).eps
```

```
np.float32(1.1920929e-07)
```

Floating point representation

Floating point refers to the decimal point (or *radix* point since we'll be working with base 2 and *decimal* relates to 10).

To proceed further we need to consider scientific notation, such as in writing Avogadro's number as $+6.023 \times 10^{23}$. As a baseline for what is about to follow note that we can express a decimal number in the following expansion

$$6.037 = 6 \times 10^0 + 0 \times 10^{-1} + 3 \times 10^{-2} + 7 \times 10^{-3}$$

A real number on a computer is stored in what is basically scientific notation:

$$\pm d_0.d_1 d_2 \ldots d_p \times b^e$$

where b is the base, e is an integer and $d_i \in \{0, \ldots, b-1\}$. e is called the *exponent* and $d = d_1 d_2 \ldots d_p$ is called the *mantissa*.

The great thing about floating points is that we can represent numbers that range from incredibly small to very large while maintaining good precision. The floating point *floats* to adjust to the size of the number. Suppose we had only three digits to use and were in base 10. In floating point notation we can express $0.12 \times 0.12 = 0.0144$ as $(1.20 \times 10^{-1}) \times (1.20 \times 10^{-1}) = 1.44 \times 10^{-2}$, but if we had fixed the decimal point, we'd have $0.120 \times 0.120 = 0.014$ and we'd have lost a digit of accuracy. (Furthermore, we wouldn't be able to represent numbers bigger than 0.99.)

Let's consider the choices that the computer pioneers needed to make in using this system to represent numbers on a computer using base 2 ($b = 2$). First, we need to choose the number of bits to represent e so that we can represent sufficiently large and small numbers. Second we need to choose the number of bits, p, to allocate to $d = d_1 d_2 \ldots d_p$, which determines the accuracy of any computer representation of a real.

More specifically, the actual storage of a number on a computer these days is generally as a double in the form:

$$(-1)^S \times 1.d \times 2^{e-1023} = (-1)^S \times 1.d_1 d_2 \ldots d_{52} \times 2^{e-1023}$$

where the computer uses base 2, $b = 2$, (so $d_i \in \{0, 1\}$) because base-2 arithmetic is faster than base-10 arithmetic. The leading 1 normalizes the number; i.e., ensures there is a unique representation for a given computer number. This avoids representing any number in multiple ways, e.g., either $1 = 1.0 \times 2^0 = 0.1 \times 2^1 = 0.01 \times 2^2$. For a double, we have 8 bytes=64 bits. Consider our representation as (S, d, e) where S is the sign. The leading 1 is the *hidden bit* and doesn't need to be stored because it is always present. In general e is represented using 11 bits ($2^{11} = 2048$), and the subtraction takes

the place of having a sign bit for the exponent. (Note that in our discussion we'll just think of e in terms of its base 10 representation, although it is of course represented in base 2.) This leaves $p = 52 = 64 - 1 - 11$ bits for d.

In this code I force storage as a double by tacking on a decimal place, `.0`.

```
bits(2.0**(-1)) # 1/2
```

```
'0011111111100000000000000000000000000000000000000000000000000000'
```

```
bits(2.0**0)  # 1
```

```
'0011111111110000000000000000000000000000000000000000000000000000'
```

```
bits(2.0**1)  # 2
```

```
'0100000000000000000000000000000000000000000000000000000000000000'
```

```
bits(2.0**1 + 2.0**0)  # 3
```

```
'0100000000001000000000000000000000000000000000000000000000000000'
```

```
bits(2.0**2)  # 4
```

```
'0100000000010000000000000000000000000000000000000000000000000000'
```

```
bits(-2)
```

```
'1100000000000000000000000000000000000000000000000000000000000000'
```

Let's see that we can manually work out the bit-wise representation of 5.25 and it matches what we came up with before in class:

```
bits(5.25)
```

```
'0100000000010101000000000000000000000000000000000000000000000000'
```

So that is $1.0101 \times 2^{1025-1023} = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$, where the 2nd through 12th bits are 10000000001, which codes for $1 \times 2^{10} + 2^0 = 1025$.

> **Question**
>
> Given a fixed number of bits for a number, what is the tradeoff between using bits for the d part vs. bits for the e part?

Let's consider what can be represented exactly:

```
dg(.1)
```

```
0.1000000000000000555
```

```
dg(.5)
```

```
0.5000000000000000000
```

```
dg(.25)
```

```
0.2500000000000000000
```

```
dg(.26)
```

```
0.2600000000000000888
```

```
dg(1/32)
```

```
0.0312500000000000000
```

```
dg(1/33)
```

```
0.0303030303030303087
```

So why is 0.5 stored exactly and 0.1 not stored exactly? By analogy, consider the difficulty with representing 1/3 in base 10.

Overflow and underflow

The magnitudes of the largest and smallest numbers we can represent are $2^{e_{\max}}$ and $2^{e_{\min}}$ where $e_{\max}$ and $e_{\min}$ are the smallest and largest possible values of the exponent. Let's consider the exponent and what we can infer about the range of possible numbers. With 11 bits for e, we can represent $2^{11} = 2048$ different exponent values, $e \in \{0, 1, 2, \ldots, 2047\}$. So the largest number we could represent should have magnitude 2^{1024}. What is this in base 10?

```
x = np.float64(10)
x**308
```

```
np.float64(1e+308)
```

```
x**309
```

```
<string>:1: RuntimeWarning: overflow encountered in scalar power
np.float64(inf)
```

```
np.log10(2.0**1024)  # Just barely overflows.
```

```
OverflowError: (34, 'Numerical result out of range')
```

```
np.log10(2.0**1023)
```

```
np.float64(307.95368556425274)
```

```
np.finfo(np.float64)
```

```
finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308, dtype=float64)
```

We could have been smarter about the calculation of 2^{1024} in base 10: $\log_{10} 2^{1024} = \log_2 2^{1024} / \log_2 10 = 1024/3.32 \approx 308$.

(Note that the reason that 2^{1024} overflows is that we need a way to represent infinity.)

The result is analogous for the smallest number, so we have that floating points can range in magnitude between about 1×10^{-308} and 1×10^{308}. Producing something larger or smaller in magnitude than these values is called overflow and underflow respectively.

Let's see what happens when we underflow in numpy. Note that there is no warning.

```
x**(-308)
```

```
np.float64(1e-308)
```

```
x**(-330)
```

```
np.float64(0.0)
```

Something subtle happens for numbers like 10^{-309} through 10^{-323}. They can actually be represented despite the fact that it doesn't seem like we should be able to represent numbers smaller than $2^{-1023} \approx 10^{-308}$. Investigating that may be an extra credit problem on a problem set.

Integers or floats?

Values stored as integers should overflow if they exceed the maximum integer.

Should 2^{65} overflow?

```
np.log2(np.iinfo(np.int64).max)
```

```
np.float64(63.0)
```

```
x = np.int64(2)
# Yikes!
x**64
```

```
np.int64(0)
```

Python's `int` type doesn't overflow.

```
# Interesting:
print(2**64)
```

```
18446744073709551616
```

```
print(2**100)
```

```
1267650600228229401496703205376
```

Of course, doubles won't overflow until much larger values than 4- or 8-byte integers because we know they can be as big as 10^{308}.

```
x = np.float64(2)
dg(x**64, '.2f')
```

```
18446744073709551616.00
```

```
dg(x**100, '.2f')
```

```
126765060022822940149670320537600
```

However we need to think about what integer-valued numbers can and can't be stored exactly in our base 2 representation of floating point numbers. It turns out that integer-valued numbers can be stored exactly as doubles when their absolute value is less than 2^{53}.

> Challenge
>
> Why 2^{53}? Write out what integers can be stored exactly in our base 2 representation of floating point numbers.

You can force storage as integers or doubles in a few ways.

```
x = 3; type(x)
```

```
<class 'int'>
```

```
x = np.float64(x); type(x)
```

```
<class 'numpy.float64'>
```

```
x = 3.0; type(x)
```

```
<class 'float'>
```

```
x = np.float64(3); type(x)
```

```
<class 'numpy.float64'>
```

Precision

Consider our representation as (S, d, e) where we have $p = 52$ bits for d. Since we have $2^{52} \approx 0.5 \times 10^{16}$, we can represent about that many discrete values, which means we can accurately represent about 16 digits (in base 10). The result is that floats on a computer are actually discrete (we have a finite number of bits), and if we get a number that is in one of the gaps (there are uncountably many reals), it's approximated by the nearest discrete value. The accuracy of our representation is to within 1/2 of the gap between the two discrete values bracketing the true number. Let's consider the implications for accuracy in working with large and small numbers. By changing e we can change the magnitude of a number. So regardless of whether we have a very large or small number, we have about 16 digits of accuracy, since the absolute spacing depends on what value is represented by the least significant digit (the *ulp*, or *unit in the last place*) in d, i.e., the $p = 52$nd one, or in terms of base 10, the 16th digit. Let's explore this:

```
# large vs. small numbers
dg(.1234123412341234)
```

```
0.12341234123412339607
```

```
dg(1234.1234123412341234) # not accurate to 16 decimal places
```

```
1234.1234123412341432413
```

```
dg(123412341234.123412341234) # only accurate to 4 places
```

```
123412341234.1234130859375000000
```

```
dg(1234123412341234.123412341234) # no places!
```

```
1234123412341234.0000000000000000000
```

```
dg(12341234123412341234) # fewer than no places!
```

```
12341234123412340736.0000000000000000000
```

We can see the implications of this in the context of calculations:

```
dg(1234567812345678.0 - 1234567812345677.0)
```

```
1.0000000000000000000
```

```
dg(12345678123456788888.0 - 12345678123456788887.0)
```

```
0.0000000000000000000
```

```
dg(12345678123456780000.0 - 12345678123456770000.0)
```

```
10240.0000000000000000000
```

The spacing of possible computer numbers that have a magnitude of about 1 leads us to another definition of *machine epsilon* (an alternative, but essentially equivalent definition to that given previously. Machine epsilon tells us also about the relative spacing of numbers.

First let's consider numbers of magnitude one. The next biggest number we can represent after $1 = 1.00...00 \times 2^0$ is $1.000...01 \times 2^0$. The difference between those two numbers (i.e., the spacing) is

$$\begin{aligned} \epsilon = & \quad 0.00...01 \times 2^0 \\ = & \quad 0 \times 2^0 + 0 \times 2^{-1} + \cdots + 0 \times 2^{-51} + 1 \times 2^{-52} \\ = & \quad 1 \times 2^{-52} \\ \approx & \quad 2.2 \times 10^{-16}. \end{aligned}$$

Machine epsilon gives the *absolute spacing* for numbers near 1 and the *relative spacing* for numbers with a different order of magnitude and therefore a different absolute magnitude of the error in representing a real. The relative spacing at x is

$$\frac{(1+\epsilon)x - x}{x} = \epsilon$$

since the next largest number from x is given by $(1+\epsilon)x$.

Suppose $x = 1 \times 10^6$. Then the absolute error in representing a number of this magnitude is $x\epsilon \approx 2 \times 10^{-10}$. (Actually the error would be one-half of the spacing, but that's a minor distinction.) We can see by looking at the numbers in decimal form, where we are accurate to the order 10^{-10} but not 10^{-11}. This is equivalent to our discussion that we have only 16 digits of accuracy.

```
dg(1000000.1)
```

```
1000000.0999999999671693563
```

Let's see what arithmetic we can do exactly with integer-valued numbers stored as doubles and how that relates to the absolute spacing of numbers we've just seen:

```
2.0**52
```

```
4503599627370496.0
```

```
2.0**52+1
```

```
4503599627370497.0
```

```
2.0**53
```

```
9007199254740992.0
```

```
2.0**53+1
```

```
9007199254740992.0
```

```
2.0**53+2
```

```
9007199254740994.0
```

```
dg(2.0**54)
```

```
18014398509481984.0000000000000000000
```

```
dg(2.0**54+2)
```

```
18014398509481984.0000000000000000000
```

```
dg(2.0**54+4)
```

```
18014398509481988.0000000000000000000
```

```
bits(2**53)
```

```
'0100001101000000000000000000000000000000000000000000000000000000'
```

```
bits(2**53+1)
```

```
'0100001101000000000000000000000000000000000000000000000000000000'
```

```
bits(2**53+2)
```

```
'0100001101000000000000000000000000000000000000000000000000000001'
```

```
bits(2**54)
```

```
'0100001101010000000000000000000000000000000000000000000000000000'
```

```
bits(2**54+2)
```

```
'0100001101010000000000000000000000000000000000000000000000000000'
```

```
bits(2**54+4)
```

```
'0100001101010000000000000000000000000000000000000000000000000001'
```

The absolute spacing is $x\epsilon$, so we have spacings of $2^{52} \times 2^{-52} = 1$, $2^{53} \times 2^{-52} = 2$, $2^{54} \times 2^{-52} = 4$ for numbers of magnitude 2^{52}, 2^{53}, and 2^{54}, respectively.

With a bit more work (e.g., using Mathematica), one can demonstrate that doubles in Python in general are represented as the nearest number that can stored with the 64-bit structure we have discussed and that the spacing is as we have discussed. The results below show the spacing that results, in base 10, for numbers around 0.1. The numbers Python reports are spaced in increments of individual bits in the base 2 representation.

```
dg(0.1234567812345678)
```

```
0.12345678123456779729
```

```
dg(0.12345678123456781)
```

```
0.12345678123456781117
```

```
dg(0.12345678123456782)
```

```
0.12345678123456782505
```

```
dg(0.12345678123456783)
```

```
0.12345678123456782505
```

```
dg(0.12345678123456784)
```

```
0.12345678123456783892
```

```
bits(0.1234567812345678)
```

```
'0011111110111111100110101101110100010101110111111001101001000011O'
```

```
bits(0.12345678123456781)
```

```
'0011111110111111100110101101110100010101110111111001101001000011l'
```

```
bits(0.12345678123456782)
```

```
'0011111110111111100110101101110100010101110111111001101001000100O'
```

```
bits(0.12345678123456783)
```

```
'0011111110111111100110101101110100010101110111111001101001000100O'
```

```
bits(0.12345678123456784)
```

```
'0011111110111111100110101101110100010101110111111001101001000100l'
```

Working with higher precision numbers

As we've seen, Python will automatically work with integers in arbitrary precision. (Note that R does not do this – R uses 4-byte integers, and for many calculations it's best to use R's `numeric` type because integers that aren't really large can be expressed exactly.)

For higher precision floating point numbers you can make use of the `gmpy2` package.

```
import gmpy2
gmpy2.get_context().precision=200
gmpy2.const_pi()

## not sure why this shows ...00004
gmpy2.mpfr(".1234567812345678")
```

3. Implications for calculations and comparisons

Computer arithmetic is not mathematical arithmetic!

As mentioned for integers, computer number arithmetic is not closed, unlike real arithmetic. For example, if we multiply two computer floating points, we can overflow and not get back another computer floating point.

Another mathematical concept we should consider here is that computer arithmetic does not obey the associative and distributive laws, i.e., $(a+b)+c$ may not equal $a+(b+c)$ on a computer and $a(b+c)$ may not be the same as $ab+ac$. Here's an example with multiplication:

```
val1 = 1/10; val2 = 0.31; val3 = 0.57
res1 = val1*val2*val3
res2 = val3*val2*val1
res1 == res2
```

```
False
```

```
dg(res1)
```

```
0.01766999999999999821
```

```
dg(res2)
```

```
0.01767000000000000168
```

Calculating with integers vs. floating points

It's important to note that operations with integers are fast and exact (but can easily overflow – albeit not with Python's base `int`) while operations with floating points are slower and approximate. Because of this slowness, floating point operations (*flops*) dominate calculation intensity and are used as the metric for the amount of work being done - a multiplication (or division) combined with an addition (or subtraction) is one flop. We'll talk a lot about flops in the unit on linear algebra.

Comparisons

As we saw, we should never test `x == y` unless:

1. `x` and `y` are represented as integers,
2. they are integer-valued but stored as doubles that are small enough that they can be stored exactly), or
3. they are decimal numbers that have been created in the same way (e.g., `0.4-0.3 == 0.4-0.3` returns `TRUE` but `0.1 == 0.4-0.3` does not).

Similarly we should be careful about testing `x == 0`. And be careful of greater than/less than comparisons. For example, be careful of `x[ x < 0 ] = np.nan` if what you are looking for is values that might be *mathematically* less than zero, rather than whatever is *numerically* less than zero.

```
4 - 3 == 1
```

```
True
```

```
4.0 - 3.0 == 1.0
```

```
True
```

```
4.1 - 3.1 == 1.0
```

```
False
```

```
0.4-0.3 == 0.1
```

```
False
```

```
0.4-0.3 == 0.4-0.3
```

```
True
```

One nice approach to checking for approximate equality is to make use of *machine epsilon*. If the relative spacing of two numbers is less than *machine epsilon*, then for our computer approximation, we say they are the same. Here's an implementation that relies on the absolute spacing being $x\epsilon$ (see above).

```
x = 12345678123456781000
y = 12345678123456782000

def approx_equal(a,b):
  if abs(a - b) < np.finfo(np.float64).eps * abs(a + b):
    print("approximately equal")
  else:
    print ("not equal")


approx_equal(a,b)
```

```
not equal
```

```
x = 1234567812345678
y = 1234567812345677

approx_equal(a,b)
```

```
not equal
```

Actually, we probably want to use a number slightly larger than machine epsilon to be safe.

Finally, sometimes we encounter the use of an unusual integer as a symbol for missing values. E.g., a datafile might store missing values as -9999. Testing for this using `==` with floats should generally be ok:`x [ x == -9999 ] = np.nan`, because integers of this magnitude are stored exactly as floating point values. But to be really careful, you can read in as an integer or character type and do the assessment before converting to a float.

Calculations

Given the limited *precision* of computer numbers, we need to be careful when in the following two situations.

1. Subtracting large numbers that are nearly equal (or adding negative and positive numbers of the same magnitude). You won't have the precision in the answer that you would like. How many decimal places of accuracy do we have here?

   ```
   # catastrophic cancellation w/ large numbers
   dg(123456781234.56 - 123456781234.00)
   ```

   ```
   0.55999755859375000000
   ```

   The absolute error in the original numbers here is of the order $\epsilon x = 2.2 \times 10^{-16} \cdot 1 \times 10^{11} \approx 1 \times 10^{-5} = .00001$. While we might think that the result is close to the value 1 and should have error of about machine epsilon, the relevant absolute error is in the original numbers, so we actually only have about five significant digits in our result because we cancel out the other digits.

   This is called *catastrophic cancellation*, because most of the digits that are left represent rounding error – many of the significant digits have cancelled with each other.
   Here's catastrophic cancellation with small numbers. The right answer here is exactly 0.00000000000000000001234.

   ```
   # catastrophic cancellation w/ small numbers
   x = .00000000000123412341234
   y = .00000000000123412340000

   # So we know the right answer is .00000000000000000001234 exactly.

   dg(x-y, '.35f')
   ```

   ```
   0.00000000000000000000123399999315140
   ```

```
## [1] "0.00000000000000000000123399999315140"
```

But the result is accurate only to 8 places + 20 = 28 decimal places, as expected from a machine precision-based calculation, since the "1" is in the 13th position, after 12 zeroes (12+16=28). Ideally, we would have accuracy to 36 places (16 digits + the 20 zeroes), but we've lost 8 digits to catastrophic cancellation.

It's best to do any subtraction on numbers that are not too large. For example, if we compute the sum of squares in a naive way, we can lose all of the information in the calculation because the information is in digits that are not computed or stored accurately:

$$s^2 = \sum x_i^2 - n\bar{x}^2$$

```
## No problem here:
x = np.array([-1.0, 0.0, 1.0])
n = len(x)
np.sum(x**2)-n*np.mean(x)**2
```

```
np.float64(2.0)
```

```
np.sum((x - np.mean(x))**2)
```

```
np.float64(2.0)
```

```
## Adding/subtracting a constant shouldn't change the result:
x = x + 1e8
np.sum(x**2)-n*np.mean(x)**2  ## YIKES!
```

```
np.float64(0.0)
```

```
np.sum((x - np.mean(x))**2)
```

```
np.float64(2.0)
```

A good principle to take away is to subtract off a number similar in magnitude to the values (in this case $\bar{x}$ is obviously ideal) and adjust your calculation accordingly. In general, you can sometimes rearrange your calculation to avoid catastrophic cancellation. Another example involves the quadratic formula for finding a root (p. 101 of Gentle).

2. Adding or subtracting numbers that are very different in magnitude. The precision will be that of the large magnitude number, since we can only represent that number to a certain absolute accuracy, which is much less than the absolute accuracy of the smaller number:

```
dg(123456781234.2)
```

```
123456781234.19999694824218750000
```

```
dg(123456781234.2 - 0.1)        # truth: 123456781234.1
```

```
123456781234.09999084472656250000
```

```
dg(123456781234.2 - 0.01)       # truth: 123456781234.19
```

```
123456781234.19000244140625000000
```

```
dg(123456781234.2 - 0.001)      # truth: 123456781234.199
```

```
123456781234.19989986816406250000
```

```
dg(123456781234.2 - 0.0001)     # truth: 123456781234.1999
```

```
123456781234.19989013671875000000
```

```
dg(123456781234.2 - 0.00001)    # truth: 123456781234.19999
```

```
123456781234.19998168945312500000
```

```
dg(123456781234.2 - 0.000001)   # truth: 123456781234.199999
```

```
123456781234.19999694824218750000
```

```
123456781234.2 - 0.000001 == 123456781234.2
```

```
True
```

The larger number in the calculations above is of magnitude 10^{11}, so the absolute error in representing the larger number is around 1×10^{-5}. Thus in the calculations above we can only expect the answers to be accurate to about 1×10^{-5}. In the last calculation above, the smaller number is smaller than 1×10^{-5} and so doing the subtraction has had no effect. This is analogous to trying to do $1 + 1 \times 10^{-16}$ and seeing that the result is still 1.

A work-around when we are adding numbers of very different magnitudes is to add a set of numbers in increasing order. However, if the numbers are all of similar magnitude, then by the time you add ones later in the summation, the partial sum will be much larger than the new term. A (second) work-around to that problem is to add the numbers in a tree-like fashion, so that each addition involves a summation of numbers of similar size.

Given the limited *range* of computer numbers, be careful when you are:

- Multiplying or dividing many numbers, particularly large or small ones. **Never take the product of many large or small numbers** as this can cause over- or under-flow. Rather compute on the log scale and only at the end of your computations should you exponentiate. E.g.,

$$\prod_i x_i / \prod_j y_j = \exp(\sum_i \log x_i - \sum_j \log y_j)$$

In many cases we keep our final calculation on the log scale and never need to exponentiate (e.g., maximizing a log-likelihood).

Let's consider some challenges that illustrate that last concern.

- Challenge: consider multiclass logistic regression, where you have quantities like this:

$$p_j = \text{Prob}(y = j) = \frac{\exp(x\beta_j)}{\sum_{k=1}^K \exp(x\beta_k)} = \frac{\exp(z_j)}{\sum_{k=1}^K \exp(z_k)}$$

for $z_k = x\beta_k$. What will happen if the z values are very large in magnitude (either positive or negative)? How can we reexpress the equation so as to be able to do the calculation? Hint: think about multiplying by $\frac{c}{c}$ for a carefully chosen c.

- Second challenge: The same issue arises in the following calculation. Suppose I want to calculate a predictive density (e.g., in a model comparison in a Bayesian context):

$$\begin{aligned} f(y^*|y,x) = & \int f(y^*|y,x,\theta)\pi(\theta|y,x)d\theta \\ \approx & \frac{1}{m}\sum_{j=1}^{m}\prod_{i=1}^{n} f(y_i^*|x,\theta_j) \\ = & \frac{1}{m}\sum_{j=1}^{m}\exp\sum_{i=1}^{n}\log f(y_i^*|x,\theta_j) \\ \equiv & \frac{1}{m}\sum_{j=1}^{m}\exp(v_j) \end{aligned}$$

  First, why do I use the log conditional predictive density? Second, let's work with an estimate of the unconditional predictive density on the log scale, $\log f(y^*|y,x) \approx \log \frac{1}{m}\sum_{j=1}^{m}\exp(v_j)$. Now note that e^{v_j} may be quite small as v_j is the sum of log likelihoods. So what happens if we have terms something like e^{-1000}? So we can't exponentiate each individual v_j. This is what is known as the "log sum of exponentials" problem (and the solution as the "log-sum-exp trick"). Thoughts?

Numerical issues come up frequently in linear algebra. For example, they come up in working with positive definite and semi-positive-definite matrices, such as covariance matrices. You can easily get negative numerical eigenvalues even if all the eigenvalues are positive or non-negative. Here's an example where we use an squared exponential correlation as a function of time (or distance in 1-d), which is *mathematically* positive definite (i.e., all the eigenvalues are positive) but not numerically positive definite:

```
xs = np.arange(100)
dists = np.abs(xs[:, np.newaxis] - xs)
corr_matrix = np.exp(-(dists/10)**2)     # This is a p.d. matrix (mathematically).
scipy.linalg.eigvals(corr_matrix)[80:99]  # But not numerically!
```

```
array([-2.10940172e-16+9.49535005e-17j, -2.10940172e-16-9.49535005e-17j,
       -1.77588414e-16+1.30159787e-16j, -1.77588414e-16-1.30159787e-16j,
       -2.09305160e-16+0.00000000e+00j,  2.23868953e-16+3.21636714e-17j,
        2.23868953e-16-3.21636714e-17j,  1.98271728e-16+9.08169093e-17j,
        1.98271728e-16-9.08169093e-17j, -1.23772161e-16+6.06460242e-17j,
       -1.23772161e-16-6.06460242e-17j, -1.49116603e-16+0.00000000e+00j,
       -2.48119681e-18+1.51188614e-16j, -2.48119681e-18-1.51188614e-16j,
       -4.08134828e-17+6.79676244e-17j, -4.08134828e-17-6.79676244e-17j,
        1.27902399e-16+2.34692846e-17j,  1.27902399e-16-2.34692846e-17j,
        5.23485748e-17+4.08641515e-17j])
```

Final note

How the computer actually does arithmetic with the floating point representation in base 2 gets pretty complicated, and we won't go into the details. These rules of thumb should be enough for our practical purposes. Monahan and the URL reference have many of the gory details.