Intro to Research Computing with Python: Numerics

Erik Spence

SciNet HPC Consortium

7 November 2013

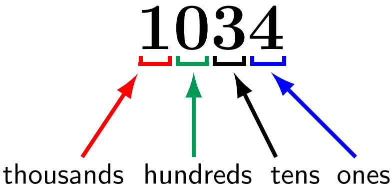
Today's class

Today we will discuss the following topics:

- Numbers. How are they represented and why.
- How computers store different types of numbers.
- The kinds of errors can creep into your calculations, if you're not careful.

How do we represent amounts?

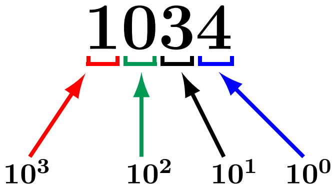
- We use numbers, of course.
- In grade school we are taught that numbers are organized in columns of digits. We learn the names of these columns.
- The numbers are understood as multiplying the digit in the column by the number that names the column.



$$\mathbf{1034 = (1 \times 1000) + (0 \times 100) + (3 \times 10) + (4 \times 1)}$$

Other ways to represent an amount

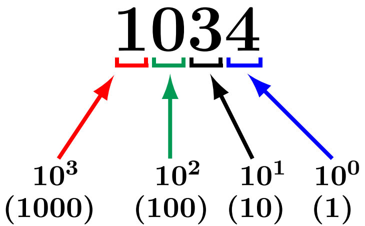
- Instead of using 'tens' and 'hundreds', let's represent the columns by powers of what we will call the 'base'.
- Our normal way of representing numbers is 'base 10', also called decimal.
- Each column represents a power of ten, and the coefficient can be one of 10 numerals (0-9).



$$1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

You can choose any base you want

How do we represent the quantity 1034 if we change bases? What about base 7? (septimal?)



$$1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

In base 7 the numerals have the range 0-6.

You can choose any base you want

How do we represent the quantity 1034 if we change bases? What about base 7? (septimal?)

$$\mathbf{1034} \qquad\qquad \mathbf{3005}$$

$$\begin{array}{cccc} 10^3 & 10^2 & 10^1 & 10^0 \\ (1000) & (100) & (10) & (1) \end{array} \qquad\qquad \begin{array}{cccc} 7^3 & 7^2 & 7^1 & 7^0 \\ (343) & (49) & (7) & (1) \end{array}$$

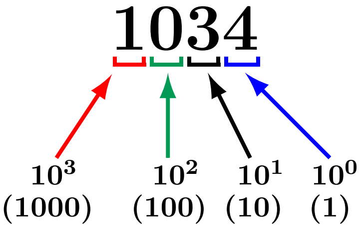
$$\mathbf{1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)}$$

$$\mathbf{1034 = (3 \times 7^3) + (0 \times 7^2) + (0 \times 7^1) + (5 \times 7^0)}$$

In base 7 the numerals have the range 0-6.

Who cares?

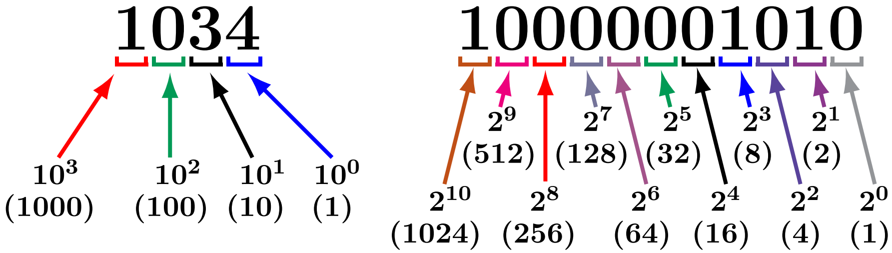
The reason we care is because computers do not use base 10 to store their data. Computers use base 2 (binary). The numerals have the range 0-1.



$$1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

Who cares?

The reason we care is because computers do not use base 10 to store their data. Computers use base 2 (binary). The numerals have the range 0-1.



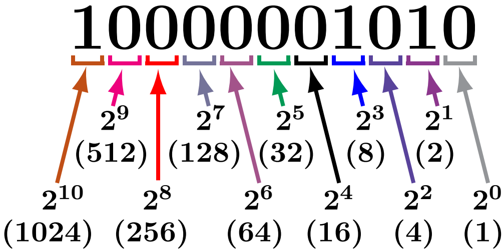
$$1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

$$\begin{aligned} 1034 \;&=\; (1 \times 2^{10}) + (0 \times 2^9) + (0 \times 2^8) + (0 \times 2^7) \\ &\quad + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) \\ &\quad + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) \end{aligned}$$

Why do computers use binary numbers?

Why use binary?

- Modern computers operate using circuits that have one of two states: 'on' or 'off'.
- This choice is related to the complexity and cost of building binary versus ternary circuitry.
- Binary numbers are like series of 'switches': each digit is either 'on' or 'off'.
- Each 'switch' in the number is called a 'bit'.



Count to 16 on one hand in binary!

How do computers store numbers?

Understanding how numbers are stored in a computer's memory is necessary to properly understand where problems can occur.

- Numbers are stored in binary format. This means base two.
- Rather than each column being a power of 10, each column is a power of 2.
- Each element of memory is called a 'bit'. The numbers to the right are eight-bit in size.

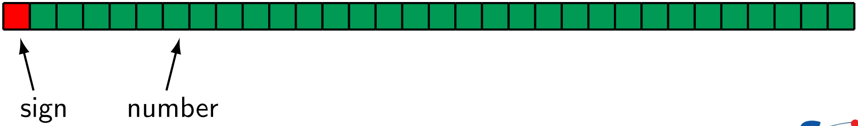
Base 10	Base 2
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
149	10010101

$$\begin{aligned} \mathbf{149} &= (\mathbf{1} \times \mathbf{2^7}) + (\mathbf{0} \times \mathbf{2^6}) + (\mathbf{0} \times \mathbf{2^5}) + (\mathbf{1} \times \mathbf{2^4}) + (\mathbf{0} \times \mathbf{2^3}) \\ &\quad + (\mathbf{1} \times \mathbf{2^2}) + (\mathbf{0} \times \mathbf{2^1}) + (\mathbf{1} \times \mathbf{2^0}) \\ &= \mathbf{128 + 16 + 4 + 1} \end{aligned}$$

Integers

- All integers are exactly representable.
- Different sizes of integer variables are available, depending on your hardware, OS, and programming language.
- A typical int is 32 bits, 1 bit for the sign.

- Finite range: can go from -$\mathbf{2^{31}}$ to $\mathbf{2^{31}-1}$ (-2,147,483,648 to 2,147,483,647).
- Unsigned integers: $\mathbf{0 \ldots 2^{32}-1}$.
- All operations (+, -, *) between representable integers are represented unless there is overflow.

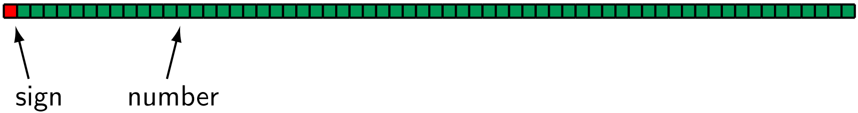


A typical int = 32 bits = 4 bytes.

Long integers

- Long integers are like regular integers, just with a bigger memory size, usually 64 bits.
- And consequently a bigger range of numbers.

- One bit for sign.
- can go from -$\mathbf{2^{63}}$ to $\mathbf{2^{63} - 1}$
- -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
- Unsigned long integers: $\mathbf{0 \ldots 2^{64} - 1}$.



A typical long int = 64 bits = 8 bytes.

Fixed point numbers

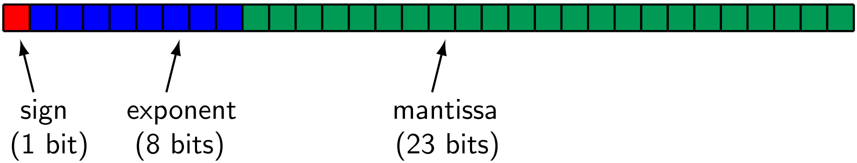
How do we deal with decimal places?

- We could treat real numbers like integers: 0 ... INT_MAX, and only keep, say, the last two digits behind the decimal point.
- This is known as 'fixed point' numbers, since the decimal place is always in the same spot.
- It is often used for financial timeseries data, since they only use a finite number of decimal places.
- But this is terrible for scientific computing. Relative precision varies with magnitude; we need to be able to represent small and large numbers at the same time.

Floating point numbers

- Analog of numbers in scientific notation.
- Inclusion of an exponent means the decimal point is 'floating'.
- Again, one bit is dedicated to sign.

$$\underbrace{-}_{\text{sign}}\underbrace{1.34}_{\text{mantissa}} \times \underbrace{10}_{\text{base}}{}^{\overbrace{-7}^{\text{exponent}}}$$



A typical single precision real = 32 bits = 4 bytes.
A typical double precision real = 64 bits = 8 bytes.

Special "numbers"

This format for storing floating point numbers comes from the IEEE 754 standard.

There's room in the format for the storing of a few special numbers.

- Signed infinities (+Inf, -Inf): result of overflow, or divide by zero.
- Signed zeros: signed underflow, or divide by +/-Inf.
- Not a Number (NaN): Sqrt of a negative number, 0/0, Inf/Inf, *etc.*
- The events which lead to these are usually errors, and can be made to cause exceptions.

Errors in floating point mathematics

There are errors inherent in using finite-length floating point variables.

- Except for numbers that fit exactly into a base two representation, assigning a real number to a floating point variable involves truncation.
- Think about how you represent 1/3. Is it 0.3? 0.33? 0.333?
- You end up with an error of 1/2 ULP (Unit in Last Place).

```
In [1]: a = 0.1

In [2]: print a
Out[2]: 0.1

In [3]: a
Out[3]: 0.10000000000000001
```

In base two, 0.1 is an infinitely repeating fraction: 0.00011001100110011001100110011...

Single precision: 1 part in $\mathbf{2^{-24}} \sim$ 6e-8.
Double precision: 1 part in $\mathbf{2^{-53}} \sim$ 1e-16.

Testing for equality

Never ever ever ever test for equality with floating point numbers!

- Because of rounding errors in floating point numbers, you don't know exactly what you're going to get.
- Instead, test to see if the difference is below some tolerance that is near epsilon.
- Testing for equality with integers is ok, however, because integers are exact.

```
In [4]: a = 0.1 * 0.1

In [5]: b = 0.01

In [6]: (a == b)
Out[6]: False

In [7]: a
Out[7]: 0.010000000000000002

In [8]: b
Out[8]: 0.01

In [9]: (abs(a - b) < 1e-15)
Out[9]: True
```

Floating point mathematics

One must be very careful when doing floating point mathematics.

Fire up Python and try the examples on the right.

What went wrong?

```
In [10]: print 1.
Out[10]: 1.0

In [11]: print 1.e-18
Out[11]: 1e-18

In [12]: print (1. - 1.) + 1.e-18
???

In [13]: print (1. + 1.e-18) - 1.
???

In [14]: print 1. + 1.e-18
???
```

Machine epsilon

Let's do some addition, to demonstrate what went wrong.

- Problem: $1.0 + 0.001$
- Let's work in base 10.
- Let's assume that we only have a mantissa precision of 3, and exponent precision of 2.

$$\begin{array}{r} 1.00 \times 10^{0} \\ +\,1.00 \times 10^{-3} \\ \hline \end{array}$$

$$\begin{array}{r} 1.00 \times 10^{0} \\ +\,0.001 \times 10^{0} \\ \hline 1.00 \times 10^{0} \end{array}$$

Machine epsilon

Let's do some addition, to demonstrate what went wrong.

- Problem: $1.0 + 0.001$
- Let's work in base 10.
- Let's assume that we only have a mantissa precision of 3, and exponent precision of 2.
- So what happened?
- Mantissa only has a precision of 3! The final answer is beyond the range of the mantissa!

$$\begin{array}{r} 1.00 \times 10^{0} \\ +\,1.00 \times 10^{-3} \\ \hline \end{array}$$

$$\begin{array}{r} 1.00 \times 10^{0} \\ +\,0.001 \times 10^{0} \\ \hline 1.00 \times 10^{0} \end{array}$$

Machine epsilon

Machine epsilon gives you the limits of the precision of the machine.

- Machine epsilon is defined to be the smallest x such that $1 + x \neq 1$.
- (or sometimes, the largest x such that $1 + x = 1$.)
- Machine epsilon is named after the mathematical term for a small positive infinitesimal.

```
In [15]: print 1.
Out[15]: 1.0

In [16]: print 1.e-18
Out[16]: 1e-18

In [17]: print (1. - 1.) + 1.e-18
Out[17]: 1e-18

In [18]: print (1. + 1.e-18) - 1.
Out[18]: 0.0

In [19]: print 1. + 1.e-18
Out[19]: 1.0
```

What's your epsilon?

You can find your approximate machine epsilon by repeatedly halving a number and testing it.

```
# myepsilon.py
def myepsilon():

  # Initialize our epsilon.
  eps = 1.0

  # Is (1 + eps) > 1?
  while ((1. + eps) > 1.):
    # If it is, divide and print it.
    eps = eps / 2.
    # Change the number of digits
    # printed so we can see them
    # all.
    print '%1.8e %1.18f' % \
      (eps, (1. + eps))
```

```
In [20]: import myepsilon
In [21]: myepsilon.myepsilon()
.
.
.
1.77635684e-15 1.000000000000001776
8.88178420e-16 1.000000000000000888
4.44089210e-16 1.000000000000000444
2.22044605e-16 1.000000000000000222
1.11022302e-16 1.000000000000000000

In [22]:
```

The epsilon is about 1e-16 for my desktop, as expected for double precision.

Underflow: look out below!

Underflow occurs when the result of a calculation is smaller than machine epsilon.

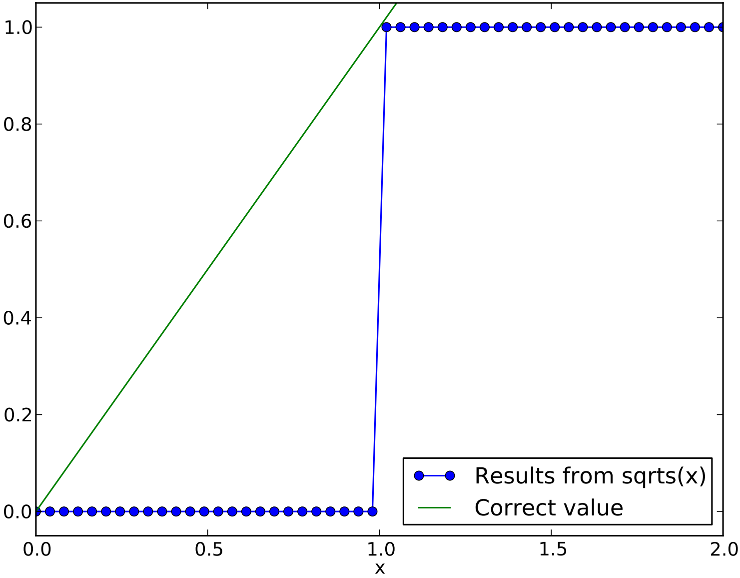
Try the following:

- Repeatedly take sqrts, then square the number.
- Plot this from 0..2.
- What should you get? What do you get?
- Loss of precision in early stages of a calculation can cause problems.

```
# underflow.py
from numpy import sqrt
def sqrts(x):

  # Make a copy of the argument.
  y = x

  # Repeatedly sqrt, then square.
  for i in xrange(128):
    y = sqrt(y)
  for i in xrange(128):
    y = y * y

  return y
```

```
In [22]: import underflow
In [23]: x = linspace(0., 2., 50)
In [24]: y = underflow.sqrts(x)
In [25]: plot(x, y, 'o-')
```

Underflow: uh oh



Underflow: what happened?

```python
# underflow.py
from numpy import sqrt
def sqrts(x):
  y = x
  for i in xrange(128):
    y = sqrt(y)
    print '%1i %1.16f' % (i,y)
  for i in xrange(128):
    y = y * y
    print '%1i %1.16f' % (i,y)
  return y
```

If the argument is below 1.0, sqrt pushes it up to epsilon below 1.0.

If the argument is above 1.0, sqrt pulls it down to exactly 1.0.

```
In [26]: sqrts(0.1)
0 0.3162277660168379
1 0.5623413251903491
.
.
126 0.9999999999999999
127 0.9999999999999999
0 0.9999999999999998
1 0.9999999999999996
2 0.9999999999999991
3 0.9999999999999982
.
.
126 0.0000000000000000
127 0.0000000000000000
Out[26]: 0.0
```

```
In [27]: sqrts(1.9)
0 1.3784048752090221
1 1.1740548859440185
.
.
126 1.0000000000000000
127 1.0000000000000000
0 1.0000000000000000
1 1.0000000000000000
2 1.0000000000000000
3 1.0000000000000000
.
.
126 1.0000000000000000
127 1.0000000000000000
Out[27]: 1.0
```

Beware: subtraction

Be very wary of subtracting very similar numbers.

- Problem: subtract 1.22 from 1.23.
- Assume that we only have a mantissa precision of 3, and exponent precision of 2.
- By performing this subtraction, we eliminate most of the information, and end up with 'catastrophic cancellation'.
- We go from 3 significant digits to 1.
- Dangerous in intermediate results.

3 sig. digits

$$\begin{array}{r} 1.23 \times 10^{0} \\ -\ 1.22 \times 10^{0} \\ \hline 1.00 \times 10^{-2} \end{array}$$

1 sig. digit

Overflow

Overflow occurs when the result of a calculation exceeds the memory size of the variable.

- 8-bit integers have a range of -128 to 127.
- When Python calculates a quantity, it up-casts all of the variables to the 'largest' variable type in the calculation.
    - int are converted to long ints
    - ints are converted to floats
    - single precision floats are converted to double.
- Always be sure to use variables that are big enough for what you're doing.

```
In [28]: a = int8(10)

In [29]: a
Out[29]: 10

In [30]: a.dtype
Out[30]: dtype('int8')

In [31]: a * a
Out[31]: 100

In [32]: a * a * a
Out[32]: -24

In [33]: a * a * int16(a)
Out[33]: 1000

In [34]: a * float(a) * int16(a)
Out[34]: 1000.0
```

Summary: things to remember

- Integers are stored exactly.
- Floating point numbers are, in general, NOT stored exactly. Rounding error will cause the number to be slightly off.
- DO NOT test floating point numbers for equality. Instead test `(abs(a - b) < cutoff)`.
- Know the approximate value of epsilon for the machine that you are using.
- Be aware of underflow: if your calculations get too close to epsilon you've lost all your precision.
- Try not to subtract numbers that are very close to one another. 'Catastrophic cancellation' leads to loss of precision.
- Be aware of overflow: use variable sizes that are appropriate for your problem.

Homework 1

1. Write a program, called DecimalToBinary, which takes as its argument a base-10 integer and returns array which contains the argument's binary form.

```
In [35]: DecimalToBinary(149)
Out[35]: array([1, 0, 0, 1, 0, 1, 0, 1])
```

2. Consider the *single precision* sequence of numbers: 1 followed by $\mathbf{10^8}$ values of $\mathbf{10^{-8}}$.
    - This sequence should sum to 2.
    - Write a program code which sums the sequence in order, and returns it.
    - Add to the program a routine which sums up values in reverse order, and returns that.
    - Add a routine which returns a pairwise sum (a sum which adds pairs of numbers, followed by the pairs of the resulting sequence, etc.)?

```
In [36]: import array
In [37]: sequence = array.array('f')
In [38]: sequence.append(1.0)
In [39]: for i in xrange(10**8): sequence.append(10**-8)
```

Homework 1, continued

3. Write a program, called OverflowUnderflow, which, given an argument $m > 1.0$, returns
   1. the minimum value of integer n that generates an overflow error when calculating m^n.
   2. the minimum value of integer p that generates an underflow error when calculating m^{-p}.

Be sure to convert the argument to single precision before performing the tests.

```
In [40]:  a = float32(10.)

In [41]:  a.dtype
Out[41]:  dtype('float32')
```