Numerical Computing with Python, Lecture 1: Numerics

Ramses van Zon   Alexey Fedoseev

November 5, 2019

1

Introduction to the Course

About the course

- Mini graduate-style course on numerical research computing
- Using Python 3 as the programming language.
- 4 weeks with 2 lectures per week
- Can be taken for credit by (astro)physics, chemistry, and possibly others, as mini/modular course.
- There will be an assignment each week.

Lectures and Office Hours

Lecture dates

Nov 5, 7, 12, 14, 26, 28, and Dec 3 and 5, 2019
1 pm - 2 pm, SciNet classroom
(suite 1140A of the MaRS building, 661 University Avenue, Toronto, ON M5G 1M1)
Lectures will be recorded and put online within a couple of days.

Office hours

Wednesdays 2 pm - 3 pm
SciNet Offices, suite 1140 of the MaRS building

Course Topics

- Numerics
- NumPy and SciPy
- Integration and ordinary differential equations
- Visualization
- Linear algebra and partial differential equations
- Binary file input and output
- Markov chain Monte Carlo
- Machine Learning

Details

- **Prerequisites:**

  You should be comfortable programming in Python (3).

- **Software that you'll need:**

  Python with numpy, scipy, matplotlib, and sklearn.

  Easiest to get (and preferred): anaconda

- **Instructors**

  - Ramses van Zon
  - Alexey Fedoseev

  Contact us at courses@scinet.utoronto.ca

- **Please fill out the sign-up sheet!**

Details - Assignments and Grading

- **Assignments**

  Programming assignments will given every week on Thursday.

  These assignments will be due the following week. Late submissions are allowed upto one week later, at a penalty per day of 5 points out of 100.

  The assignments should be handed in online in the 'dropbox' on the course website: https://courses.scinet.utoronto.ca/473

- **Grading scheme**

  The grading scheme will be based on the average of the four homework assignments.

2

Installing Python

Getting Python

- With Python 3, we will use a number of packages such as `numpy`, `scipy`, `matplotlib`, and `sklearn`.

Getting Python

- With Python 3, we will use a number of packages such as `numpy`, `scipy`, `matplotlib`, and `sklearn`.
- The easiest way to get all of these is to install one of the Anaconda distributions:

Getting Python

- With Python 3, we will use a number of packages such as `numpy`, `scipy`, `matplotlib`, and `sklearn`.
- The easiest way to get all of these is to install one of the Anaconda distributions:

Getting Python 3 on SciNet

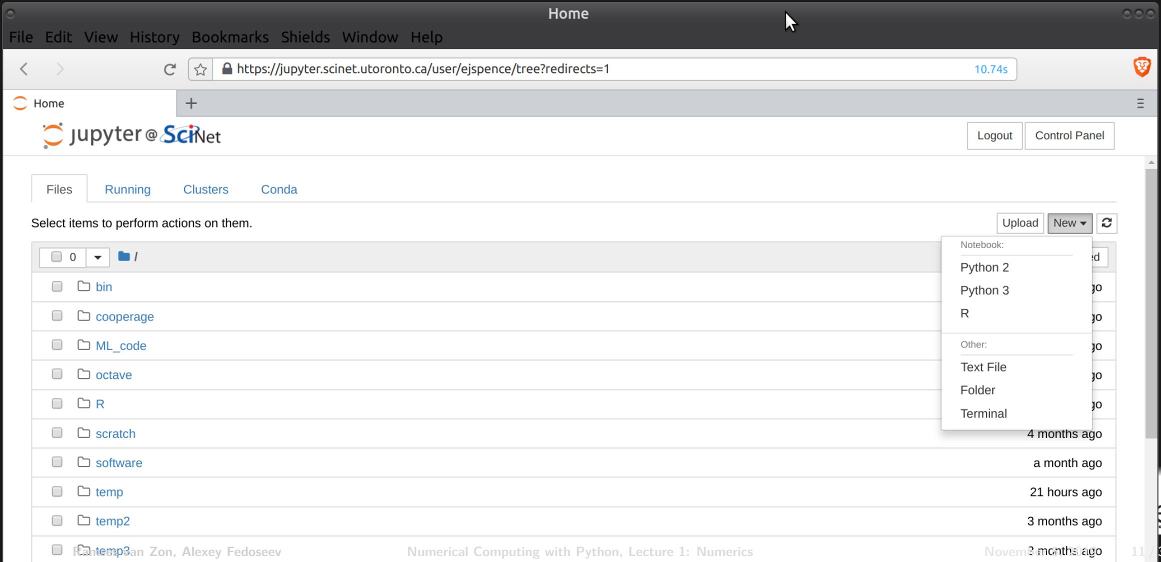
- You of course can also work on SciNet's Niagara.
- We'd suggest using the 'intelpython3' module.

```
$ ssh -Y USERNAME@niagara.scinet.utoronto.ca
Last login: Tue Nov  6 12:02:57 2018 from
....

$ module load intelpython3

$ python3 #(or ipython3, or ipython3 --pylab)
Python 3.6.3 | Intel Corporation| (default, Feb 12 2018, 06:37:09)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
Intel(R) Distribution for Python is brought to you by Intel Corporation.
Please check out: https://software.intel.com/en-us/python-distribution
>>>
```

- You can also use SciNet's jupyterhub at https://jupyter.scinet.utoronto.ca

Jupyter on SciNet

https://jupyter.scinet.utoronto.ca



Python, IPython, Jupyter: Which one to use?

Totally up to you, as long as:

- You use Python 3.
- If your favorite (or your friends favorite) python environment fails, you are able to switch to plain command-line python.
- You are able to write plain python scripts that run regardless of the environment.

  That means that opening a terminal and running the script with the command `python3 SCRIPTNAME` must work.
- This is important because you have to submit your homework as plain python scripts that you have tested and that we can run. No jupyter notebooks, no ipython extensions for the assignments.

3

Enough preliminaries, let's get started...

Research Computing

A.K.A.: Computational Science, Scientific Computing.

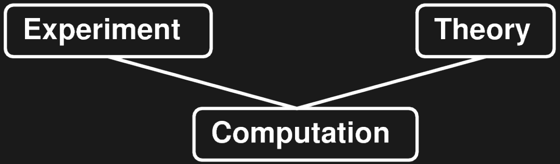
Using a computing device (computer) to figure out values of quantities of interest in the scientific endevour.

One computes for a variety of reasons, such as

- Large data processing/data mining
- Investigating behaviour of models too complex to deal with on paper
- Interpret experimental results using a theoretical model
- Finding simpler models from more complex ones
- Visualization

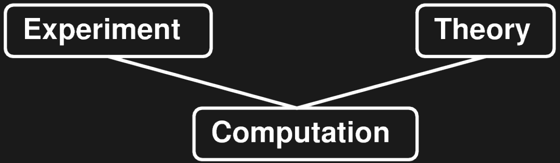
Third Leg?

Research Computing is often called the third leg of science:



Third Leg?

Research Computing is often called the third leg of science:

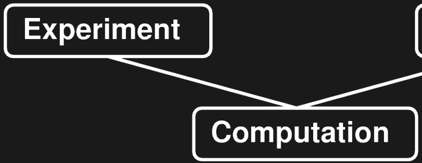


Won't get into philosophical matters.

From a practical perspective:

Third Leg?

Research Computing is often called the third leg of science:



Won't get into philosopical matters.

From a practical perspective:

- Computation is used by experiment and theory.
- Research Computing can learn from best practices in both theoretical and experimental science.
- It is often closer to a well controlled experiment.
- Requires some knowledge and skills unique to computing.

4

Numerics

Numerics

- Numerical analysis will be one of the themes of this mini-course.

  (Data analysis is the other.)

- Today we'll look at numbers:
  - How they are represented and why.
  - How computers store different types of numbers.
  - The kind of errors that can creep into numerical calculations.

How do we represent quantities?

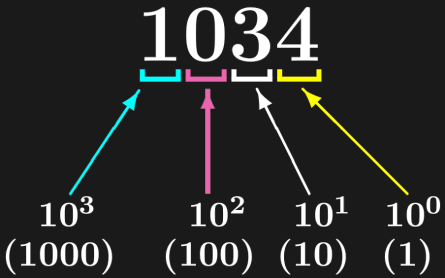
- We use numbers, of course.
- In grade school we are taught that numbers are organized in columns of digits. We learn the names of these columns.
- The numbers are understood as multiplying the digit in the column by the number that names the column.

$$\underbrace{1}_{\text{thousands}}\underbrace{0}_{\text{hundreds}}\underbrace{3}_{\text{tens}}\underbrace{4}_{\text{ones}}$$

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

Other ways to represent a quantity

- Instead of using tens' andhundreds', 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 (septenary)?

$$\underbrace{1}_{\substack{10^3\\(1000)}}\underbrace{0}_{\substack{10^2\\(100)}}\underbrace{3}_{\substack{10^1\\(10)}}\underbrace{4}_{\substack{10^0\\(1)}}$$

You can choose any base you want

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

$$\underbrace{1}_{10^3\,(1000)}\underbrace{0}_{10^2\,(100)}\underbrace{3}_{10^1\,(10)}\underbrace{4}_{10^0\,(1)}$$

$$\underbrace{3}_{7^3\,(343)}\underbrace{0}_{7^2\,(49)}\underbrace{0}_{7^1\,(7)}\underbrace{5}_{7^0\,(1)}$$

You can choose any base you want

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

$$\underbrace{1}_{\substack{10^3\\(1000)}}\underbrace{0}_{\substack{10^2\\(100)}}\underbrace{3}_{\substack{10^1\\(10)}}\underbrace{4}_{\substack{10^0\\(1)}} \qquad \underbrace{3}_{\substack{7^3\\(343)}}\underbrace{0}_{\substack{7^2\\(49)}}\underbrace{0}_{\substack{7^1\\(7)}}\underbrace{5}_{\substack{7^0\\(1)}}$$

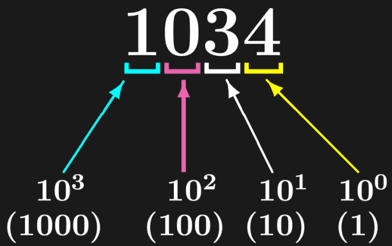
$$1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$
$$1034 = (3 \times 7^3) + (0 \times 7^2) + (0 \times 7^1) + (5 \times 7^0)$$

Note: 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.



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.

$$\mathbf{1034}$$

10^3 (1000) $\quad 10^2$ (100) $\quad 10^1$ (10) $\quad 10^0$ (1)

$$\mathbf{10000001010}$$

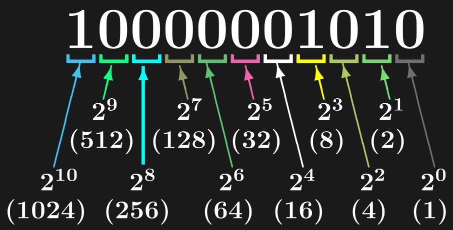
2^{10} (1024) $\quad 2^9$ (512) $\quad 2^8$ (256) $\quad 2^7$ (128) $\quad 2^6$ (64) $\quad 2^5$ (32) $\quad 2^4$ (16) $\quad 2^3$ (8) $\quad 2^2$ (4) $\quad 2^1$ (2) $\quad 2^0$ (1)

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

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”.



Integers

All integers are exactly representable.

- Different sizes of integer variables are available, depending on your hardware, OS, and programming language.
- For most languages, a typical integer is 32 bits.
- Negative numbers often represented using "two's complement". $(-x \equiv 2^{32} - x)$
- Finite range: can go from -2^{31} to $2^{31} - 1$ (-2,147,483,648 to 2,147,483,647).
- Unsigned integers: 0...$2^{32} - 1$.
- All operations (+, -, *) between representable integers are represented unless there is overflow.

Integers

All integers are exactly representable.

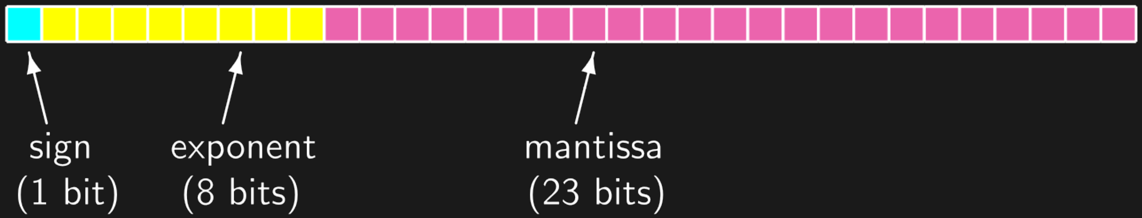
- Different sizes of integer variables are available, depending on your hardware, OS, and programming language.
- For most languages, a typical integer is 32 bits.
- Negative numbers often represented using "two's complement". $(-x \equiv 2^{32} - x)$
- Finite range: can go from -2^{31} to $2^{31} - 1$ (-2,147,483,648 to 2,147,483,647).
- Unsigned integers: 0...$2^{32} - 1$.
- All operations (+, -, *) between representable integers are represented unless there is overflow.

- The CPU has dedicated circuitry to deal with integers as described above.
- But Python integers have infinite range;
  That's convenient, but means integer arithmetic is done in software.
- To get the faster, low-level integers back, we can use datatypes from `numpy` (next lecture).

Floating point numbers

- Analog of numbers in scientific notation.
- Inclusion of an exponent means the decimal point is "floating".
- One bit is dedicated to sign.

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



A typical single precision real = 32 bits = 4 bytes.

A typical double precision real = 64 bits = 8 bytes.

Floats in Python

Floating point numbers

- Based on the 64 bits floating point type.
- You can specify the exponent by putting "e" in your number.
- Information about floats on your system can be found in `sys.float_info`.

Complex numbers

- Have a real and imaginary part, both of which are floats.
- Use z.real and z.imag to access individual parts.

```
import sys
print(sys.float_info)
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308,
min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

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): square root 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.
- Notice that there is no “NA” option. The pandas package uses NaN; there is no standard approach.

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).

```
>>> a = 0.1
>>> print(a)
0.1
>>> print( '%.18f' % a )
0.100000000000000006
```

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

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

Double precision: 1 part in $2^{-53} \sim$ 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 okay, however, because integers are exact.

```
>>> a = 0.1 * 0.1
>>> b = 0.01
>>> print(a == b)
False
>>> print(a)
0.010000000000000002
>>> print(b)
0.01
>>> print(abs(a - b) < 1e-15)
True
```

Floating point mathematics

One must be very careful when doing floating point mathematics.
In Python, try the examples on the right.
What went wrong?

```
>>> print(1.)
1.0
>>> print(1.e-18)
1e-18
>>> print( (1. - 1.) + 1.e-18 )
1e-18
>>> print( (1. + 1.e-18) - 1. )
0.0
>>> print( 1. + 1.e-18 )
1.0
```

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.

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.

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

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

- So what happened?
- Mantissa only has a precision of 3! The final answer is beyond the range of the mantissa!

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 number.

```
>>> print(1.)
1.0
>>> print(1.e-18)
1e-18
>>>
>>> print( (1. - 1.) + 1.e-18 )
1e-18
>>> print( (1. + 1.e-18) - 1. )
0.0
>>> print( 1. + 1.e-18 )
1.0
```

What's your epsilon?

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

```
#file: 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)))
```

```
>>> import myepsilon
>>> 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
>>>
>>> import sys
>>> print(sys.float_info.epsilon)
>>> 2.220446049250313e-16
```

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

The limits of precision: look out below!

Problems will occur when the result of a calculation spans more orders of magnitude than the inverse of machine epsilon.

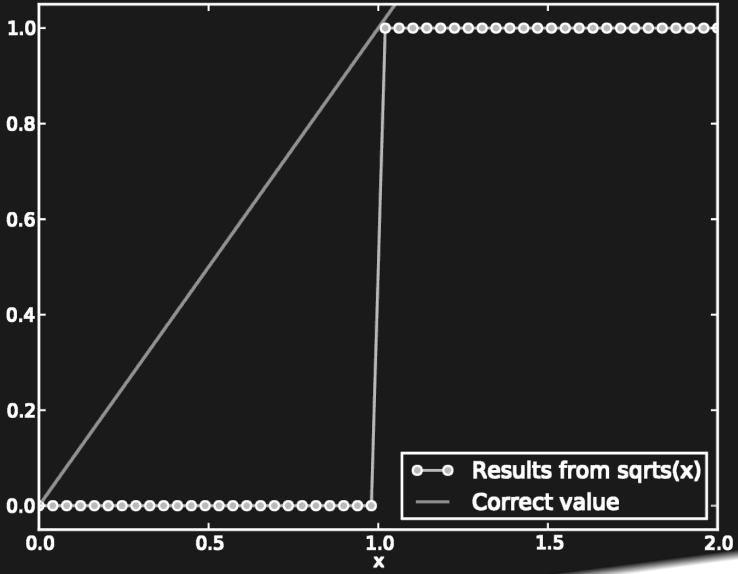
Try the following:

- For the range of numbers between 0 and 2, repeatedly take square roots of the numbers, then repeatedly square the numbers.
- Plot the result, from 0..2.
- What should you get? What do you get?
- Loss of precision in early stages of a calculation causes problems.

```
from numpy import sqrt
def sqrts(x):
 #Make a copy of the argument.
 y = x
 #Repeatedly sqrt, then square.
 for i in range(128):
  y = sqrt(y)
 for i in range(128):
  y = y * y
 return y
```

```
from numpy import linspace
from matplotlib.pyplot import plot,show
x = linspace(0., 2., 50)
y = precision.sqrts(x)
plot(x, y, 'o-')
show()
```

Precision problem: uh oh



Precision problem: what happened?

```
from numpy import sqrt
def sqrts(x):
 y = x
 for i in range(128):
  y = sqrt(y)
  print('%1i %1.16f'%(i,y))
 for i in range(128):
  y = y * y
  print('%1i %1.16f'%(i,y))
 return y
```

```
>>> 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
0.0
>>>
```

```
>>> 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
1.0
>>>
```

If the argument is below 1, sqrt pushes it up to epsilon below 1. If the argument is above 1, sqrt pulls it down to exactly 1.

Beware: catastrophic cancelation

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

The same problem can occur when dividing large numbers.

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 are doing.

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 are doing.

```
>>> from numpy import int8, int16
>>> a = int8(10)
>>> print( a )
10
>>> print( a.dtype )
int8
>>> print( type(a) )
<class 'numpy.int8'>
>>> print( a * a )
100
>>> print( a * a * a )
__main__:1: RuntimeWarning: overflow encountered in
byte_scalars
-24
>>> print( int8(1000) )
-24
>>> print( a * a * int16(a) )
1000
>>> print( a * float(a) * int16(a) )
1000.0
```

Underflow

An underflow error is the opposite of an overflow error: you are attempting to make a number which is smaller than the variable can hold.

- 32-bit floats integers have a range of -3.4e38 to +3.4e38
- An overflow error will result if you attempt to go beyond this range.
- An underflow error results if you try to go too small.

```
>>> from numpy import float32
>>>
>>> print(float32(-1e35))
-1e+35
>>> print(float32(-1e44))
-inf
>>>
>>> print(float32(1e-40))
9.9999461e-41
>>> print(float32(1e-44))
9.8090893e-45
>>> print(float32(1e-46))
0.0
>>>
```

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.
- Know the limits of your precision: if your calculations span as many orders of magnitude as the inverse of epsilon you're going to lose precision.
- Try not to subtract floating point numbers that are very close to one another.
- Be aware of overflow and underflow: use variable sizes that are appropriate for your problem.