Problem set #1

Release 0.1

Brian Granger, John Hunter and Fernando Pérez.

February 04, 2010

Contents

1 Wallis' formula

Wallis' formula is a slowly converging infinite product that approximates pi as

$$\pi = \lim_{n\to\infty} 2\prod_{i=1}^{n} \frac{4i^2}{4i^2-1}. \tag{1}$$

While this isn't a particularly good way of computing π from a numerical standpoint, it provides for an excellent illustration of how Python's integers are more flexible and powerful than those typically found by default in compiled languages like C and Fortran. The problem is that for (1) to be even remotely accurate, one must evaluate it for fairly large values of n, where both the numerator and the denominator will easily overflow the limits of 64-bit integers. It is only after taking the ratio of these two huge numbers that the value is small (close to π).

Fortunately for us, Python integers automatically allocate as many digits as necessary (within the limits of physically available memory) to hold their result. So while impementing (1) in C or Fortran (without auxilary libraries like GMP) would be fairly tricky, in Python it's very straightforward.

For this exercise, write a program that implements the above formula. Note that Python's `math` module already contains π in double precision, so you can use this value to compare your results:

```
In [94]: import math

In [95]: math.pi
Out[95]: 3.1415926535897931
```

Keep in mind that Python by default divides integers by truncating:

```
In [96]: 3/4
Out[96]: 0
```

this will change in the future, and the new behavior can be activated even today:

```
In [97]: from __future__ import division

In [98]: 3/4
Out[98]: 0.75
```

so you should put this `from __future__` statement as the first non-comment, non-docstring statement in your program, to ensure that you can get the division to produce a floating point number in the end.

2 Numerical chaos in the logistic map and floating point error

One of the most classic examples of chaotic behavior in non-linear systems is the iteration of the logistic map

$$x_{n+1} = f(x_n) = rx_n(1 - x_n) \tag{2}$$

which for $x \in (0, 1)$ and $r \in (0, 4)$ can produce very surprising behavior. We'll revisit this system later with some more sophisticated tools, but for now we simply want use it to illustrate numerical roundoff error.

Computers, when performing almost any floating point operation, must by necessity throw away information from the digits that can't be stored at any finite precision. This has a simple implication that is nonetheless often ovelooked: algebraically equivalent forms of the same expression aren't necessarily always numerically equivalent. A simple illustration shows the problem very easily:

```
In [75]: x = 1

In [76]: y = 1e-18

In [77]: x+y-x
Out[77]: 0.0

In [78]: x-x+y
Out[78]: 1.0000000000000001e-18
```

For this exercise, try to find three different ways to express $f(x)$ in (2) and compute the evolution of the same initial condition after a few hundred iterations. For this problem, it will be extremely useful to look at your results graphically; simply build lists of numbers and call matplotlib's `plot` function to look at how each trace evolves.

The following snippet can be used as a starting point, and it includes some hints on what values of r to look at:

```
"""Illustrating error propagation by iterating the logistic map.

f(x) = r*x*(1-x)

Write the above function in three algebraically equivalent forms, and study
their behavior under iteration.  See for what values of r all forms evolve
identically and for which ones they don't.
"""

import matplotlib.pyplot as plt

# Interesting values to try for r:
# [1.9, 2.9, 3.1, 3.5, 3.9]
x0 = 0.6  # any number in [0,1] will do here
numpoints = 100
```

Note: This was inspired by this very nice presentation about Python's `turtle` module, which includes some great numerical examples.

3 Dictionaries for counting words

A common task in text processing is to produce a count of word frequencies. While NumPy has a builtin histogram function for doing numerical histograms, it won't work out of the box for couting discrete items, since it is a binning histogram for a range of real values.

But the Python language provides very powerful string manipulation capabilities, as well as a very flexible and efficiently implemented builtin data type, the *dictionary*, that makes this task a very simple one.

In this problem, you will need to count the frequencies of all the words contained in a compressed text file supplied as input. Load and read the data file `HISTORY.gz` (without uncompressing it on the filesystem separately), and then use a dictionary count the frequency of each word in the file. Then, display the 20 most and 20 least frequent words in the text.

3.1 Hints

- To read the compressed file `HISTORY.gz` without uncompressing it first, see the `gzip` module.
- Consider 'words' simply the result of splitting the input text into a list, using any form of whitespace as a separator. This is obviously a very naive definition of 'word', but it shall suffice for the purposes of this exercise.

- Python strings have a `.split()` method that allows for very flexible splitting. You can easily get more details on it in IPython:

```
In [2]: a = 'somestring'

In [3]: a.split?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
Namespace:      Interactive
Docstring:
    S.split([sep [,maxsplit]]) -> list of strings

    Return a list of the words in the string S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator.
```

The complete set of methods of Python strings can be viewed by hitting the TAB key in IPython after typing `a.`, and each of them can be similarly queried with the `?` operator as above. For more details on Python strings and their companion sequence types, see here.