Introduction to Testing and Debugging

Note: the code here is not terribly Pythonic, because we're trying to illustrate particular points, so we must know, for instance, exactly what our type is. The examples are also intended to transfer without error between Py 2 and Py3, which leaves some things an-idiomatic

- 01_RoundingAndTruncation

  - Demonstrates how single and double precision floats can be inexact, and what this can do unexpectedly

  Info:

  In Python all floating point numbers are 64 bit

  Summing a rather small number many times often doesn't give quite the expected result. This makes it risky as a loop condition as you can get 1 (or rarely more) fewer or extra loops depending on platform, optimisation levels etc

  Try:

  What we set as 0.01 might not come out that way, and what we do see depends on formatting. Try different numbers and compare the results. In particular, try setting n to a power of 2

- 02_CombiningBigAndSmall

  - This code prints a variety of values found by summing floats. In particular it sums small numbers onto a larger one

  Try:

  Adjust the number or range of values in the array

  Adjust the init value to see the effect of a large number on the accuracy of the final sum

- 03_Overflows

  - This code demonstrates the min and max values of integers and floats and what happens when they're exceeded

  Info:

  In Python 2.7 there are ints and longs, in Python 3 this is hidden from you. Both will automatically promote from int to long on overflow. Note that longs can be a lot slower than ints, so you should still beware

  Floating point numbers also have limits. The IEEE (I triple-E) standards define strictly what behaviour must be. The program 03-05_InfAndNanTable (C so compile with gcc {name} -o {outputname}) shows the required values of operations on Inf and NaN (see also 05) values.

  Note 1: Here FLT_MAX + 1 is still FLT_MAX because the finite Epsilon for a value this large is more than 1. Try finding the minimum needed to add to get inf

  Note 2: Here we are running into what are called denormal numbers. Usually the base part of the number is a number between 0.0 and 1.0 as in normal scientific notation. In denormal numbers this is a pure decimal. This allows the representation of smaller numbers, but at lower precision.

  Try:

  Try adjusting the number or range of values

- 04_MixedMode

  - This code demonstrates the dangers of combining int and float in calculations

  Info:

  Note: in Py 2 this matches C and Fortran. In Py 3 there is automatic promotion. Compare the answers if you have both versions. Also compare the explicit integer calculation on line 20

  Try:

  In general operands are promoted to the "biggest" type involved in a single operation (biggest here means that ints will be boosted to longs, but can also be boosted to floats). Using the order of operations (BODMAS, PEMDAS etc) work out

what is happening. Note that the division operator "associates with" a number to its right. Note also that nothing guarantees the order things are evaluated

adjust the erroneous calculation on line 13 to be correct

- 05_NaNContagion

  - This program explores NaN, how it arises and how it behaves

  Info:

  Note: Divide by zero and sqrt(1) in plain Python are exceptions, which strictly violates the IEEE standards. Numpy does not, but for simplicity here we don't use it. Otherwise Python obeys IEEE

  Try:

  Using the facts about NaN comparisons implement an isNaN function using the stub definition in the code. Test it with some numbers.

  Python math has an isNaN function. Look it up and try it

- No 06 in Python

- 07_AccessErrors

  - This program shows what happens if you access outside the bounds of an array

  Info:

  This code writes beyond its array bounds, which causes an exception. Python has checks for this sort of error, which is part of the reason it is slower than languages like C

  Note that checking the length before access also fails if we have None

  Try:

  Adjust how far out of bounds the write goes.

  If you're familiar with Numpy, you might try the same example with a Numpy array

- No 08 in Python

- 09_Tarpit

  - This code has a logic bug which makes it very very slow.

  Info:

  If you compile and run, you'll see numbers printed. If it takes a very long time or no time at all, adjust the reps parameter until it takes 2-3 seconds

  The code is writing to a file, but accidentally opening it every iteration. You may notice the numbers go in pulses, when the writing actually flushes to disk.

  Use the command-line option '-q' to omit printing i at every step

  Try:

  Supply the argument 'fix' to use a fixed version without the extra work