# Demonstrating Profiling

Note: these examples use python3 and are sometimes not terribly Pythonic. Some examples are deliberately inefficient to illustrate how to improve common issues

To run the examples, either start python and import the example, then run the "main" function (with or without the profiler), or run python{3} {name.py} and observe. Inside the interpreter, you can also run individual functions for partial profiles.

- 01_basic_profile.py

  - Demonstrates a nice profile

  **Info:**

  This example works out the amplitude in space (or time) of a series of summed sine waves. It does two different ways, one of which has a nice speed boost

  **Try:**

  Run the code, and observe the profile. What is taking the most time?

  Since there's only a few calls, you can trace the call graph of the two "chunks" of this code. Do the times all add up?

  Before looking deeply at the code, at which level (simple_calc*, calc_wave_amplitude* or neither) have we tweaked things for a time bonus?

- 02_tot_v_cum_prof.py

  - Function level profiling benefits from well named functions

  **Info:**

  This example just finds the mean of a numpy array - it has to first generate a random array and we aren't interested in that timing here

  **Try:**

  Try profiling either "main()" or "processing(create_array())". What's the difference? Which profile makes it easier to find the compute timings?

Notice that most of the functions here have very low or "0.000" tottime, but non-zero cumtime - we really need to see the call graph to work out what's going on. Which method is ultimately taking all the compute time?

Try generating a call graph. You'll want gprof2dot and to put profiling info into a pstats file

- 03_streaming_processing.py

- Data from disk is slow - line-by-line reading in Python is slower

**Info:**

This example reads numbers from a file and calculates either their sum or the sum of Gaussians centred on each value. The data file is generated on first run, or if it doesn't exist.

The code either reads line by line, processing as it goes ("streaming", method 1) or reads the whole file first, then does the calculations ("pre-read" method 2). This is picked randomly, or you can supply a command line arg for method or supply the use_method parameter to main.

You can change the number of elements by tweaking n_els on line 12 and deleting the existing file - it will be regenerated. Be cautious - 1 billion (10^9) elements will be approx 8GB!

**Try:**

Run both versions for the default n_els. Which is quicker? Does the profile indicate why?

What's the minimum time this code might take? If you don't know disk speeds for the machine you're using, assume 100MB/sec.

Does it matter whether we use the "_total" or the "_heavier_processing" functions?

For the pre-read version does it matter if we do the conversions with a list comprehension (convert=True in read_data) or not?