Profiling with Python

Performance matters in code. There is a qualitative difference between an application that will run immediately compared to one you have to wait for an hour to complete. But sometimes an application is very easily fixed! It was not optimized for performance.

A lot of code spends a lot of time doing things it doesn't really need to be doing.

How do we know what to optimize? We can rely on our intuition and experience. Which is valid at any stage. However, a more measured approach is that of *profiling*. Profiling is vital for intensive applications on the scientific stack.

Here, we will examine how to profile Python code using the standard library module `cProfile` (yes, that is a lower case 'c' and capital 'P').

Using cProfile

There are two modes of use for cProfile. One is within your script, while the other runs on a standalone script you have written. The former is good for testing a complex function you're in the process of writing, while the latter tests the performance bottlenecks of a perhaps production-ready script.

The documentation for cProfile and others may be found here:
https://docs.python.org/3/library/profile.html

The profiler takes the function or code in question and runs it, while obtaining detailed timing information about each call. The standard format is:

ncalls	Number of calls
tottime	Total time spent running the particular function
percall	tottime/ncalls
cumtime	Cumulative time spent in the function and all those it calls (including recursion)
percall	cumtime/primitive calls
filename:lineno(function)	Information about each function call

Profiling within your code

You need to import cProfile before using it:

```
import cProfile
```

You then run it on a specific function defined in scope:

```
cProfile.run("function_name(parameters)")
```

You may specify an output filename too:

```
cProfile.run("function_name(parameters)", "outfile_name.cprof")
```

Profiling a production-ready script

You can also call cProfile on a Python script. This is achieved using the module flag `-m` from the Python run-time:

```
» python -m cProfile script_name.py
```

You can specify an output file to document the gory statistics. This is a more test-driven approach:

```
» python -m cProfile -o output.cprof script_name.py
```

You can also select which output to sort by

```
» python -m cProfile -s sort_by script_name.py
```

1. Define a function for the factorial operation, using the most naive recursion you can think of.

2. Profile this function within your code. You may have to adjust the input parameter to, say, 1000 or more to see non-zero timings.

3. Compare the performance of your function to the `math.factorial()` function. Is there any difference? Why/how?

4. Take the code you wrote for the automation section (or any other script you have handy) and profile it.

5. Save the profiling output to a file.

6. Where is the performance bottleneck? In other words, which operation is taking the longest to execute? Can you optimize this further? Why/why not?