How-To: Profile in Python

P. Fowler-Wright

2023-10-19

Profiling in Python is provided by the `cProfile` module. We additionally use the SnakeViz graphical viewer (`pip install snakeviz`) to visualise the call stack, but if you do not have access to packages outside the standard Python library or a graphical user interface (SnakeViz is browser based) the `pstats` module can be used to inspect the profile instead.[1] To profile a Python program contained in `myscript.py`:

```
python -m cProfile -o myscript.prof myscript.py
```

The output is saved to `myscript.prof`, and may be viewed with

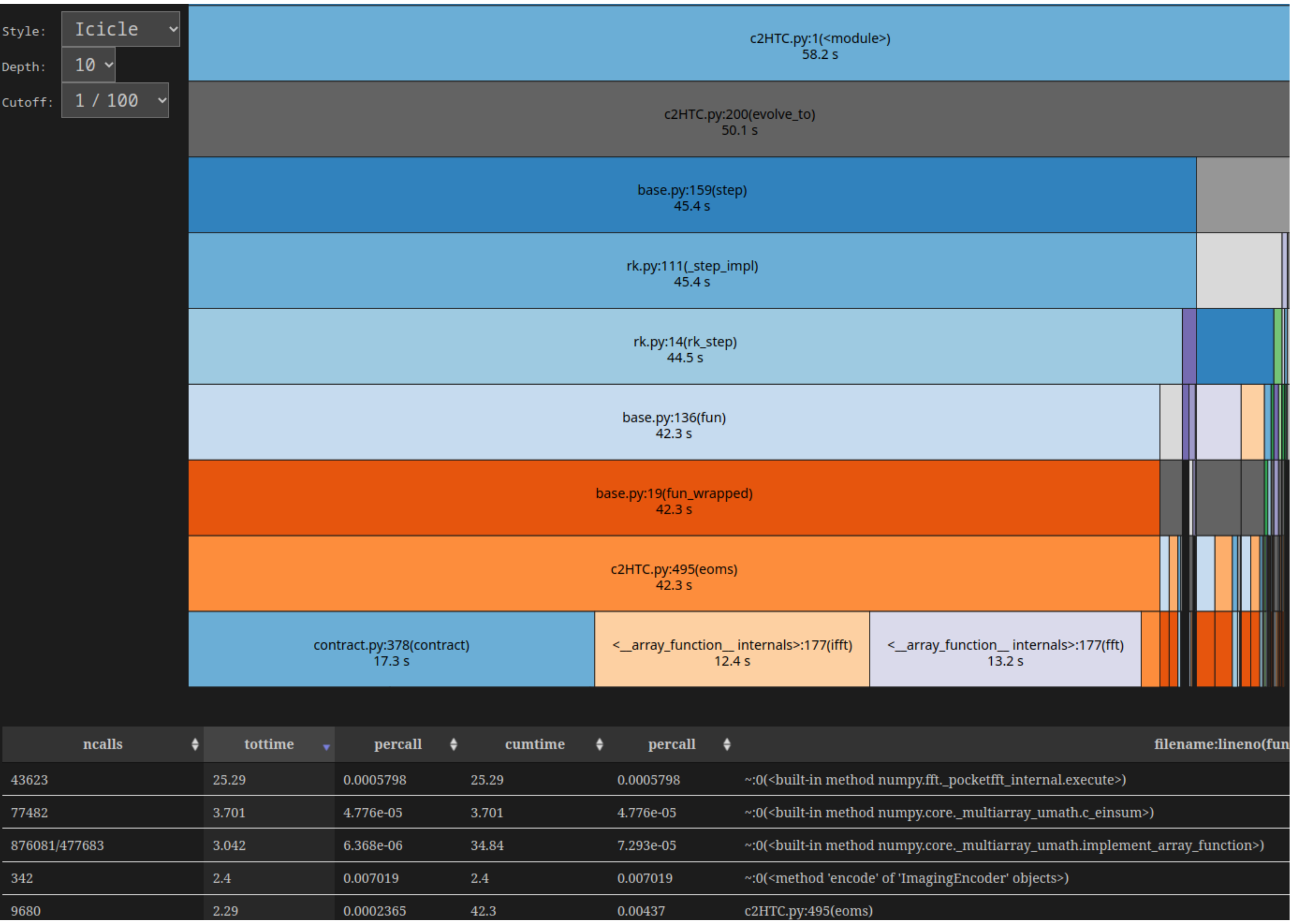
```
snakeviz myscript.prof
```

This should open a view in a new browser window. Below an extract for a script `c2HTC.py` whose main function `evolve_to` uses the Runge-Kutta solver (`rk.py`) from the `scipy.integrate` submodule to integrate a system of equations `eoms` over a fixed interval.

[1]To print the 10 most significant function calls by cumulative time,

```
import pstats
p = pstats.Stats("myscript.prof")
print(p.sort_stats(pstats.SortKey.CUMULATIVE).print_stats(10))
```

For further details see the `pstats.Stats` class on the The Python Profiles documentation page.



ncalls	tottime	percall	cumtime	percall	filename:lineno(fun
43623	25.29	0.0005798	25.29	0.0005798	~:0(<built-in method numpy.fft._pocketfft_internal.execute>)
77482	3.701	4.776e-05	3.701	4.776e-05	~:0(<built-in method numpy.core._multiarray_umath.c_einsum>)
876081/477683	3.042	6.368e-06	34.84	7.293e-05	~:0(<built-in method numpy.core._multiarray_umath.implement_array_function>)
342	2.4	0.007019	2.4	0.007019	~:0(<method 'encode' of 'ImagingEncoder' objects>)
9680	2.29	0.0002365	42.3	0.00437	c2HTC.py:495(eoms)

The main two quantifiers are `tottime`, the total time spent in a function alone i.e. not including calls to sub-functions and `cumtime`, the cumulative time spent in this and all subfunctions (from invocation to exit). It is the latter that is shown above using nested bars.

Here we see out of 58 second total runtime the main function `evolve_to` took approximately 50; there was some initialisation and cleanup around this function. Moving the cursor over any block provides additional information (displayed in the sidebar): doing so for the first generically named `base.py:159` reveals this is the base script in the `scipy.integrate` submodule.

Going down to the final line, what this profiling has revealed is that the majority of the runtime is split between performing tensor contractions with the `contract` function of `opt_einsum` and performing Fast Fourier Transforms with `numpy`'s `fft/ifft`. In general one might check for unexpected timesinks (due to purely written or extraneous code) or at least identify the main computational bottleneck(s) which could be targeted for optimisation e.g. can a more efficient pathing algorithm be chosen for the tensor contractions?

Further considerations Typically you run the profiler on a ‘standard’ mode for your script. If your Python program has multiple modes of operations, you probably want to profile each ‘mode’ separately. Another useful strategy is to generate profiles whilst changing a parameter to identity the scaling of different parts of the computation with that parameter to e.g. determine the cause of runaway growth of computation time.