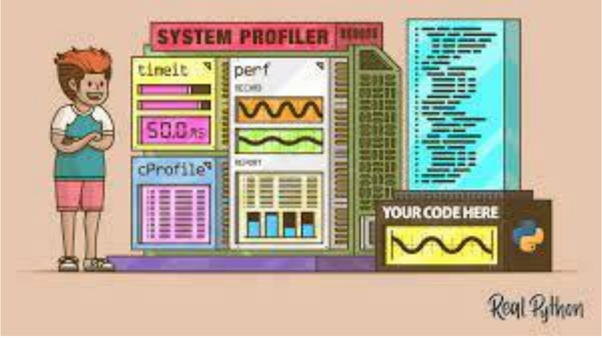

Profiling Python code

Di Gioia Serafina

What is software profiling?

Profiling is a process where we analyze **'time' usage (time complexity)** and **space complexity** (memory consumption) of a program/process/application.

The tools used to profile software applications can be divided into three classes:

- time profilers / performance profilers
- memory profilers
- complete profilers

Time and memory profilers can helps us make better decisions to utilize underlying resources efficiently.

Memory profiling is particularly necessary in scientific computing to avoid unnecessary crashes of application for “out of memory” errors!!

When does profiling comes into play?

When I think to good practices for code developping I think always to this list of actions:

1) Testing: Have you tested your code to prove that it works as expected and without errors?
2) Refactoring: Does your code need some cleanup to become more maintainable and Pythonic?
3) Profiling: Have you identified the most inefficient parts of your code?

So I would not suggest to use any serious profile before having tested the code on which you are working on for output errors, and I would suggest also to make the code a bit more nicer, refactoring it, because refactoring usually highlights repetitions avoiding errors and also leading to more optimized code.

Two ways to approach time profiling

Deterministic profiling

Deterministic profiling is meant to reflect the fact that all *function call*, *function return*, and *exception* events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing).

Statistical profiling

statistical profiling (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

Python time profilers

The Python standard library provides two different implementations of the same profiling interface:

1. `cProfile` is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on `lsprof`, contributed by Brett Rosen and Ted Czotter.
2. `profile`, a pure Python module whose interface is imitated by `cProfile`, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

The **cProfile** and **profile** provide profiling results on a function basis but don't give us information on line by line basis of function. The results generated by these libraries have time taken by function calls but no information about time taken by individual lines of each function. Python has a library called **line_profiler** which can help us better understand the time taken by individual lines of our code.

Apart from the standard we can consider other options:

line_profiler, Scalene, yappi, pprofile, Snakeviz, Pyinstrument

line_profiler is an useful alternative to cProfile because it allows user to measure the time spent in the single code line execution

Bad aspects of cProfile/Profile

- Low accuracy . underlying “clock” is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock.
- overhead latency (this impacts more Profile than cProfile, and Profile can be calibrated to reduce this bias)

Memory profiling

We have a long list of different memory profilers in Python

memory_profiler, memprof, memray, guppy/hpy, tracemalloc, Scalene, Pympler etc.

but we will focus on memory_profiler and Scalene.

Memory_profiler allows both to record stats on memory occupancy of different functions, and to plot them nicely. However it really can slow down significantly the analyzed code.

A competitive alternative only for memory profiling is memray.

I will show you Scalene, because this profiler is very powerful, since allows simultaneously time profiling and memory profiling both on CPUs and GPUs.

Profilers comparison

Profiler	Slowdown	Lines or Functions	Unmodified Code	Threads	Multi-processing	Python vs. C Time	System Time	Profiles Memory	Python vs. C Memory	GPU	Memory Trends	Copy Volume	Detects Leaks
						CPU-only profilers							
`pprofile (stat.)`	1.0×	lines	✓	✓	-	-	-	-	-	-	-	-	-
`py-spy`	1.0×	lines	✓	✓	✓	-	-	-	-	-	-	-	-
`pyinstrument`	1.7×	functions	✓	-	-	-	-	-	-	-	-	-	-
`cProfile`	1.7×	functions	✓	-	-	-	-	-	-	-	-	-	-
`yappi wallclock`	3.2×	functions	✓	✓	-	-	-	-	-	-	-	-	-
`yappi CPU`	3.6×	functions	✓	✓	-	-	-	-	-	-	-	-	-
`line_profiler`	2.2×	lines	-	-	-	-	-	-	-	-	-	-	-
`Profile`	15.1×	functions	✓	-	-	-	-	-	-	-	-	-	-
`pprofile (det.)`	36.8×	lines	✓	✓	-	-	-	-	-	-	-	-	-
						memory-only profilers							
`fil`	2.7×	lines	-	-	-	-	-	*peak only*	-	-	-	-	-
`memory_profiler`	≥37.1×	lines	-	-	-	-	-	*RSS*	-	-	-	-	-
`memray`	4.0×	lines	-	✓	-	-	-	*peak only*	✓	-	-	-	-
						CPU+memory profilers							
`Austin (CPU+mem)`	1.0×	lines	✓	✓	✓	-	-	*RSS*	-	-	-	-	-
`Scalene (CPU+GPU)`	1.0×	**both**	✓	✓	✓	✓	✓	-	-	✓	-	-	-
`Scalene (all)`	1.3×	**both**	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓


SO LONG AND...
Thanks for all the fish!