A Survey of Open-Source Python Profilers

PETER NORTON

Peter works on automating cloud environments. He loves using Python to solve problems. He has contributed to books on Linux and Python, helped with the New York Linux Users Group, and helped to organize past DevOpsDays NYC events. In addition to Python, Peter is slowly improving his knowledge of Rust, Clojure, and maybe other fun things. Even though he is a native New Yorker, he is currently living in and working from home in the northeast of Brazil. pcnorton@rbox.co.

In my day-to-day work, I am fortunate enough to have access to a lot of tools that give me insight into our running services, including data collection and visualizations of distributed traces. Distributed tracing at its core shows the flow of requests through the various services that they are handled by and usually includes the information about the time they take in each service, along with some special plumbing that allows a particular connection, or event, or action to be tracked across those different systems so they can be correlated. Based on the sheer volume of the data of tracking and tracing requests across different processes on different systems in a network, the thing that makes the traced data useful is good data being fed into good visualization tools.

Working on a smaller scale—for example, when I'm writing tools for my own use or writing standalone scripts—doesn't feed into the same tracing infrastructure (often called Application Performance Monitoring, or APM), and so I don't get the benefit of the tooling and the visualizations that I'm used to.

In order to get similar insight into my own smaller use cases, I usually take a side trip that involves researching the available profiler and then visualization options. Since I use them infrequently, I have tried a few profiling tools over the years, and I haven't settled on a single best tool.

So I'm going to use this opportunity to survey the field, focus on my opinions of the most important features, and summarize what I see as the pros and cons for the profilers I think will be the most useful to me when I don't have to or don't want to work within a larger infrastructure.

In addition, I'm going to favor mentioning available options that can be invoked without having to modify the code being run. While there are times when adding profiling code to your program may be the proper approach, that's something that would be done after some planning and discussion, and I'm hoping to look at useful first options here.

Why Profiling Is Done

When we release a program for use by others, the goal is to create a self-contained experience that works in isolation. By this I mean that it's considered a flaw, and confusing to the user, when a program fails with a stack trace, memory contents, or in some other manner that exposes its internal state, code, or anything that breaks the fourth wall of software.

On the other hand, when we write a program, we need to see it as a complex composition of a whole lot of independent pieces that have to be carefully arranged to work as intended. Confirming that it is working involves being able to review each piece of the infrastructure and ensuring that its expected form and function are in place.

The contrast between what we present to the user and how we work on the inside leads to our having to solve the conundrum of making the same program that we don't want to expose also able to give those who understand it (us, me, you, the developer, or the technical user) the capability to look into the running program and characterize it in whole and in its parts.

There are a few different methods of gaining insight. The lowest-effort way, and often the first, is some variations on printing or logging the internal state you're interested in. This is always an important method, but if you have code that's out of your control (like in a library), logging isn't an option since you can only coarsely select what is logged. Also, if you have code in a fast loop, logging isn't usually an option because it has a way of causing more problems than it solves—either filling disks or just making it impossible to see any other context.

Applying a debugger is the other method that is always good to use to understand a program. Debuggers accomplish their primary purpose, which is to look inside the state of the program without having to modify it at all. The debugger is always a good choice when the program in question is misbehaving. One of the key points of a debugger is that it will stop the program, and keep everything stopped, so that you can inspect the state of the program, or it will work off of a core dump—either way the program will not make progress while you work in the debugger. That is absolutely what you want, but it is a potential problem if you are interested in investigating the uninterrupted behavior of a program as it continues running.

Other methods include recording and emitting metrics. This is always useful in the long run, but it requires modifying the program and, to be really useful, summarizing and visualizing the trends. So it requires infrastructure, which we'd like to not have to wire up if there are lower-effort methods.

Profiling

A profiler is another way of investigating the way a program works. Both logging/printing and using a debugger allow us to look at specific isolated bits of code as needed—for example, with the addition of unusual numbers of print statements or logging lines or when the program's dead and its only traces are a core dump after a failure. In contrast, profilers work by characterizing the performance of the functions as the program continues to run.

There are two approaches that are taken to do this in general and in the Python scene. The first way is that the actual running program is modified in an automated fashion when the profiler is invoked in a way that each invoked function is enriched so that the time it takes is recorded; the aggregate time spent in each function is thus recorded so that you are presented with cold hard facts about where your program spent its time and where it really didn't.

Alternatively, instead of recording the fact of every action taken on the stack, the profiler will look at the state of the stack by sampling at a steady interval, say every 100 ms. By showing you over time what was on the stack, it can infer approximately the same information as watching every function entry and exit, but without imposing as much overhead. This is usually called statistical profiling or sampling.

In either case, the generated statistics, together, are called the profile, showing you the program's metaphorical outline and contours—importantly, where a lot of its time is spent. Since it's rare to get much benefit from optimizing things that aren't taking a lot of time, the profile is the lens that lets you focus on your performance. Once a profile is created it gives you real data that is the starting point for forming a hypothesis about your next steps, helping you follow the scientific method to continue to improve your program by making changes, then re-running your code and looking for differences between the before and after profiles.

Some Code to Profile

I spent some time looking through some of the available AWS public data sets and decided to use the NEXRAD data, modifying one of the example Jupyter notebooks as the base of a small bit of example code that just loads data through a few layers of libraries.

To make the graphs and screenshots in this column reproducible, I have put up a Docker file on GitHub that you can use to run the same steps I have (https://github.com/pcn/ogin-some-pyprofiling).

The Available Profilers

Using the Built-in Profilers

The Python documentation describes in good detail the `profile` and `cProfile` modules. They ship in the standard library (if you're reading the 2.x documentation, ignore the unmaintained hotshot module). In practice the `cProfile` module will be the only one of these that you'll ever use.

Because these are part of the base Python, it's an easy first thing to reach for. I almost always invoke the profiler from the command line and record output to a profile file, which saves the info about the run. This is so much more versatile than the default of having the profiler print out its result once the run has finished.

Since the provided documentation really does cover the modules well, I'll just present my take on the tradeoffs when using the `cProfile` or `profile` modules. Basically, they require that you stop your program and invoke it in a one-off manner.

One less-used feature is that you can enable and disable the profiler modules via their `enable()` and `disable()` methods; you can choose to run your program normally and turn on profiling when some condition is hit, e.g., if you hit it with a signal, send a specific message, or if the program itself notices that it's slowing down. Then you can turn profiling off after some amount of time.

On the downside, no matter what else you do, the profiling is done in the same process as your code. You can imagine that the profiling is conceptually done by decorating each function entrance and exit on the stack with time, resource info, etc., so it's unfortunately not necessarily appropriate for high-performance situations where the loss of cycles in production is not allowable. So profiling is frequently done by enabling the profiler while running a representative chunk of code with a representative chunk of data in a QA or staging situation, and using that to simulate production, which can work as well.

Let's look at an example:

```
$ python -m cProfile -o generate.prof generate_data.py
```

This runs the `generate_data.py` script and records profile data in `generate.prof`, which we can process using the profilers `pstats` module.

The default output from the profiler isn't very useful and requires a lot of cogitation. Instead, you pretty much always need to use the `pstats` module in order to start to find actionable info.

```
import pstats
from pstats import SortKey
p = pstats.Stats('generate.prof')
p.strip_dirs().sort_stats(1).print_stats()
```

This is a slight modification of the example from the standard library docs, which prints the most impactful function invocations at the top of the list instead of the end, which is just my preference. The output looks like this:

```
$ python basic_stats.py | head -20 2>/dev/null
Sun Sep  8 14:56:36 2019    generate.prof

  1459508 function calls (1424219 primitive calls) in
24.581 seconds

  Ordered by: internal time

  ncalls  tottime  percall  cumtime  percall  filename:lineno
(function)
      257   19.124   0.074   19.124   0.074  {method 'recv_into'
of '_socket.socket' objects}
        6    3.988   0.665    3.988   0.665  {method 'connect'
of '_socket.socket' objects}
     1022    0.148   0.000    0.148   0.000  {built-in method
marshal.loads}
     2307    0.057   0.000    0.057   0.000  {built-in method
builtins.compile}
   52/133    0.050   0.000    0.091   0.001  {built-in method _imp.
create_dynamic}
  33/2629    0.044   0.000    0.163   0.000  {built-in method
builtins.__build_class__}
1652/322    0.031   0.000    0.085   0.000  sre_parse.py:475(_parse)
    5787    0.028   0.000    0.028   0.000  {built-in method
posix.stat}
    4745    0.027   0.000    0.031   0.000  {method 'sub' of
're.Pattern' objects}
    2467    0.023   0.000    0.039   0.000  inspect.py:613(cleandoc)
       2    0.022   0.011    0.026   0.013  core.py:1005(__call__)
  146627    0.020   0.000    0.020   0.000  {method 'startswith'
of 'str' objects}
  129897    0.020   0.000    0.026   0.000  {built-in method
builtins.isinstance}
```

If you know that this is downloading data, and you know that means it's getting data over a socket, this is telling you that most of the actual time spent waiting for the program was spent receiving data from a socket.

While this is good information, it doesn't try to take on the responsibility of helping you to understand the code and the relationships between bits of code. It'd be much more useful if it could tell you where in the call stack these were invoked to some extent—in short if it could provide more context. In a way it can—in `basic_stats.py`, which is printing the `pstats` data, you can iterate and choose which functions to print out and how to describe whether to print their callers, their callees, etc. This means that in order to get some really useful data, you're required to step out of the problem you're really trying to solve (getting more performance out of your code) and think about how to get better data out of the profiling module. It seems like there should be a better way.

So the state of the built-in profiler is that it definitely profiles functions, but it relies on you inferring the state of the stack, and in order to get a useful overview and to zoom in on what you want, you will need to become familiar with the `pstats` module.

Let's look at some other profilers that include more batteries.

A Little Bit About Sampling vs. Deterministic Profilers

The built-in Python profiling modules call themselves "deterministic," which basically means that they will completely encompass every function entry and return—you can read a lot more about that in the standard library documentation. The determinism is in the fact that if you run the same program twice with the same inputs, it's guaranteed that you'll get the same functions profiled both times.

However ideal this seems, it is not always the appropriate approach. The approach of statistical or sampling for profiling can have some pretty attractive advantages. First and foremost, the mechanism can be implemented both within the process and, with some fancy work, externally.

When done internally (that is, within the same interpreter), it can result in much less performance impact on the running process. When done externally, it can result in even less impact by running the profiler into a separate CPU entirely while it's doing its work.

There can be some doubt whether it's appropriate to switch from a deterministic problem-solving method that completely covers all possibilities if there's a chance that something could be missed—for example, an invocation of a function going unnoticed. In practice, for a profiler this should almost never be a real problem since the point of profiling is not to describe every detail of a program's running, but to help determine where the program is spending significant amounts of time. A statistical or sampling profiler is very unlikely to miss functions, and the call stacks leading to them using a lot more CPU time than expected, for example, because these should clearly stick out when the sampling process is collecting data.

One more almost incidental advantage is that since there is a series of events being triggered, it's also useful to potentially gather other environmental factors with a statistical profiler, that is, overall system health indicators like CPU load, I/O utilization, etc.

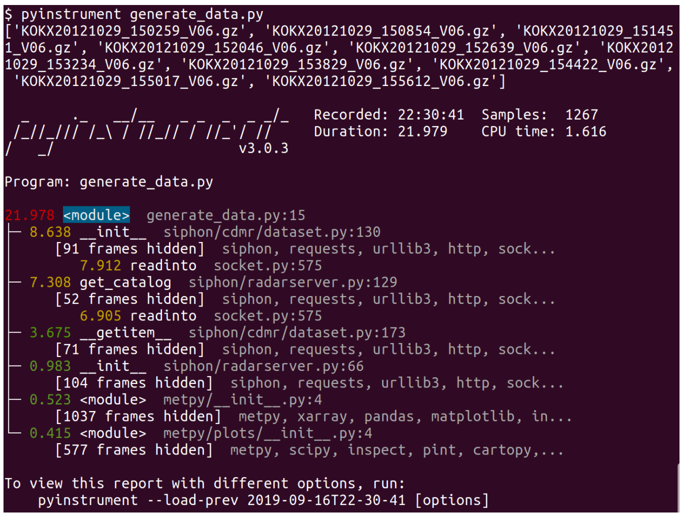
pyinstrument

`pyinstrument` is the first of these open-source projects I've found recently (https://github.com/joerick/pyinstrument). It's simple to invoke as the built-in profiler, it's installable via `pypi`, and it's been releasing versions since 2014. Unlike the built-in profiler, it not only prints output at the end but also records a profile you can use to rerun it with different display parameters. All you need to do is run `pyinstrument` with your command after any options (Figure 1).

`pyinstrument`'s default output starts out as useful. It displays the functions in order from those with the most time seen to the least. It also defaults to hiding library calls in order to help you focus on your own code to start with. In addition, it colors the output red/yellow/green, so you can use that as a starting point for identifying where problems may be found and also for excluding code paths in the profile that you probably don't need to see.

A thoughtful, useful, and simple output option is that it can display function calls in the order they were invoked rather than ordering by their cumulative time; in this way, you can also relate the program's behavior from the user's perspective to long times spent in a particular function.

Because it automatically saves the profile without your having to think about it, `pyinstrument` makes it one step easier to export the profile as JSON, text, or, for simple-ish profiles, a nice self-contained HTML page that is easy to share and to explore interactively by twiddling pull-down triangles.



```
$ pyinstrument generate_data.py
['KOKX20121029_150259_V06.gz', 'KOKX20121029_150854_V06.gz', 'KOKX20121029_15145
1_V06.gz', 'KOKX20121029_152046_V06.gz', 'KOKX20121029_152639_V06.gz', 'KOKX2012
1029_153234_V06.gz', 'KOKX20121029_153829_V06.gz', 'KOKX20121029_154422_V06.gz',
 'KOKX20121029_155017_V06.gz', 'KOKX20121029_155612_V06.gz']

  _     ._   __/__   _ _  _  _ _/_   Recorded: 22:30:41  Samples:  1267
 /_//_/// /_\ / //_// / //_'/ //     Duration: 21.979    CPU time: 1.616
/   _/                      v3.0.3

Program: generate_data.py

21.978 <module>  generate_data.py:15
├─ 8.638 __init__  siphon/cdmr/dataset.py:130
│     [91 frames hidden]  siphon, requests, urllib3, http, sock...
│        7.912 readinto  socket.py:575
├─ 7.308 get_catalog  siphon/radarserver.py:129
│     [52 frames hidden]  siphon, requests, urllib3, http, sock...
│        6.905 readinto  socket.py:575
├─ 3.675 __getitem__  siphon/cdmr/dataset.py:173
│     [71 frames hidden]  siphon, requests, urllib3, http, sock...
├─ 0.983 __init__  siphon/radarserver.py:66
│     [104 frames hidden]  siphon, requests, urllib3, http, sock...
├─ 0.523 <module>  metpy/__init__.py:4
│     [1037 frames hidden]  metpy, xarray, pandas, matplotlib, in...
└─ 0.415 <module>  metpy/plots/__init__.py:4
      [577 frames hidden]  metpy, scipy, inspect, pint, cartopy,...

To view this report with different options, run:
    pyinstrument --load-prev 2019-09-16T22-30-41 [options]
```

Figure 1: A simple run of `pyinstrument` with the `generate_data.py` script

Part of the simplicity that I've found when profiling with `pyinstrument` is that it helps you look in your own code first. This is always a sensible starting point since that should be the only code that's changing, and thus the most likely place you should look at for some kind of performance regression. In line with this bit of common sense, `pyinstrument` will default to not expanding info about functions from files whose file system paths include the string `/lib/` by default, though you can toggle this behavior.

py-spy

The next profiler, py-spy (https://github.com/benfred/py-spy), works entirely differently from the other two profilers I've mentioned so far and, along with the next one, in a way that I think is exciting to have for Python. There are two big distinctions: it focuses on showing you what your program's profile looks like over time, and it operates outside of the process being profiled.

py-spy takes its inspiration from a project called `pyflame`, which appears to be unmaintained at this point. The "flame" part of `pyflame` refers to support for displaying Python profiles as flame graphs, which are a very useful visualization technique that Brendan Gregg has been developing and advocating. Flame graphs are a way to visually represent what the stack looks like over time, which allows answers to questions that are otherwise hard to get.

py-spy is significantly different from the built-in profilers and `pyinstrument` specifically because it now takes the profiling outside of the process being profiled. py-spy has the very interesting approach of using the OS-provided stack inspection calls to look at the process for a vanishingly small amount of time, record the state of the stack, query the Python interpreter, and do a whole lot of frankly very clever work to gather and present that data.

py-spy can attach to a running program and gather information without interrupting a running task or server, so you don't need to modify your code.

Since it's grouping the events it records into slices of time, it will default to a `top`-like display, which updates the summary of what functions/stack the program is spending the most time executing.

```
$ py-spy—python ./generate_data.py

Collecting samples from 'python ./generate_data.py' (python v3.7.4)
Total Samples 400
GIL: 0.00%, Active: 2.00%, Threads: 1

%Own   %Total  OwnTime  TotalTime  Function (filename:line)
2.00%   2.00%   0.020s    0.020s   readinto (socket.py:589)
0.00%   0.00%   0.000s    0.010s   <module> (siphon/cdmr/
   cdmrfeature_pb2.py:11)
0.00%   0.00%   0.000s    0.020s   <module> (matplotlib/
   rcsetup.py:25)
0.00%   0.00%   0.000s    0.290s   <module> (siphon/
   catalog.py:21)
    [etc.]
```

One important note is that when sampling and attempting to output a flame graph of the sampled data, a lot of data will be presented, even at the rate of 100 samples per second. By default the flame graph mode will sample limiting output to two seconds. Though you can run it for longer if you need to, the principle is that you should invoke this when you are experiencing a slowdown since the flame graph is very dense. In some cases, you may need to run it for longer periods, but two seconds is forever in CPU time, so when delving into the detail you'll get from a flame graph output, you should usually not need much more. The black lines in the following example represent the actual progress bar.

```
    $ py-spy -n -f pyspyflame.svg -r 100—python ./generate_data.py
    Sampling process 100 times a second for 2 seconds. Press
    Control-C
      to exit.

    ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    ■■■■■■■■■■■■■■■■■■■■■■ 200/200
    Wrote flame graph 'pyspyflame.svg'. Samples: 200 Errors: 0
```

This results in an SVG file that you can explore in your browser, with stack frame info available in tooltips that come up as you mouse over.

The above example demonstrates running a script under py-spy, but it is much more interesting when attaching to a running program.

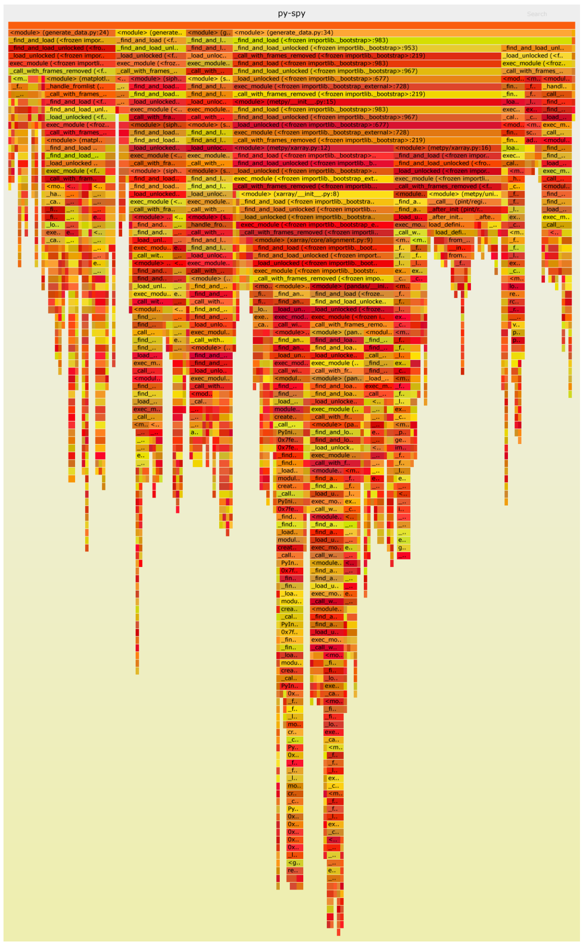


Figure 2: The flame graph for a two-second sample of the `generate_data.py` script

And Lastly, Austin

The last profiler I'll mention is whimsically named after the movie character Austin Powers (I'll venture a guess that like py-spy, it's spying for you, and the movie character is a spy, so I guess that's the connection?), and it's just called Austin. You can get it from https://github.com/P403n1x87/austin, and even though it overlaps with many of the best features of py-spy and `pyinstrument`, it's not exactly as easy to just reach for it. py-spy and `pyinstrument` are both easily installed via `pip install` (though py-spy's author has done some very cool trickery to make that possible). Austin requires you to return to the days of Autotools and Make, but it does ease the way by also offering a few pre-packaged methods of installing it.

For people who've been around for a while, this shouldn't get in the way of trying out a useful tool, but I find that most of my colleagues over the past decade or so are not excited about anything with instructions that include `autoconf` and `make`. That said, the Austin maintainer has made it available as a snap package, which is certainly a step in the right direction.

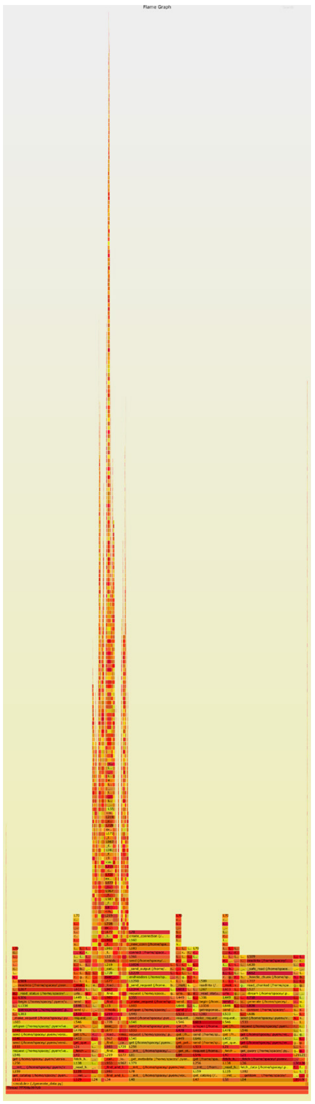


Figure 3: The flame graph for a full run, sampled by Austin.

Austin is as simple to run as `pyinstrument` or py-spy. Austin is very similar in features and scope to py-spy, with some additional modules that are provided when installing with pip/setuptools—a terminal top-like view similar to py-spy, as well as a web UI that lets you observe a process being traced in a browser, which is a nice touch. While these are both promising directions, in my testing I haven't found a use case for these features in my workflow.

There are two major distinctions in my mind between py-spy and Austin. py-spy limits the amount of time you can sample a trace when outputting to a flame graph, while Austin is happy to continue tracing until you stop it. I'm not sure which is right—both could be a best practice, but I think I would lean towards py-spy in this aspect.

The other major distinction is that Austin attempts to record the changes in memory usage by the process while it's tracing, which is a very nice touch—unfortunately, I haven't found a way to visualize that alongside the flame graphs yet.

Once it's installed, Austin will output profile data to `STDOUT`, which can be read by the `flamegraph.pl` tool. The same data can be saved to a file with the `-o` flag, and then post-processed as well:

```
$ austin -s -i 500 -o austin.profile -f -m python
  ./generate_data.py
$ cat austin.profile | flamegraph.pl --countname=us
  > austin_generated.svg
Ignored 11 lines with invalid format
```

Something Like a Conclusion

I think that the most interesting thing I've realized is how much easier the profiling tools I've described here are when it comes to getting some insight—they're much more useful than the default profiling modules. I've got a definite preference among these tools:

First, I would probably not bother loading up the default profile modules anymore. They provide a feeling of poorly made flatpack furniture—a bunch of pieces with a guiding document and a rough idea of what you could accomplish if only you had already done this a lot.

For a quick overview of what a program is doing, if I could stop and start it in isolation, I would reach for `pyinstrument` given its simplicity and easy formatting capabilities.

For a running service, I'd currently reach for py-spy. I think it covers the important features of sampling, understanding the stack, and outputting useful data and visualizations.

I am interested in Austin for one of the distinctions that I mentioned about it: one of the data points it can collect is the memory usage of the process being profiled on each tick. The more I think about this feature, the more I think that there's a good case to be made for tracking increasing memory usage and other information usually provided by `vmstat`/`mpstat`/`iostat` as part of the profile. Relating the profile of the program to the profile of the system that's running it is very useful and would bring Python profilers closer to the capabilities of APM products.