Python 3.9 StatsProfile — My first OSS Contribution to cPython

Daniel Olshansky

Feb 17 · 4 min read

You can try out all of the code in this article yourself using this Google Colaboratory notebook.

If you've ever tried to debug and optimize your python application, it's likely that you stumbled upon Python Profiles to understand where most of the execution time is being spent. You enable the profiler at the beginning of a code segment you're interested in profiling with `pr.enable()` , and call `pr.create_stats()` at the end.

```
import cProfile, pstats
import time
from random import randint

def sleep1():
    time.sleep(0.1)

def sleep2():
    time.sleep(0.2)

def sleep3():
    time.sleep(0.3)

pr = cProfile.Profile()
pr.enable()
```

```python
pr.enable()

for _ in range(10):
    for _ in range(randint(1, 10)):
        sleep1()

    for _ in range(randint(1, 10)):
        sleep2()

    for _ in range(randint(1, 10)):
        sleep3()

pr.create_stats()
ps = pstats.Stats(pr)
ps.print_stats()
```

get_stats_profile_example_print_stats.py hosted with ❤ by GitHub view raw

Afterwards, you can create a Stats object, and print the results in a human readable format with `ps.print_stats()` .

```
         66 function calls in 4.507 seconds

   Ordered by: internal time, cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       24    4.507    0.188    4.507    0.188 {built-in method time.sleep}
       10    0.000    0.000    1.002    0.100 <ipython-input-7-96f3d7189345>:12(sleep1)
        7    0.000    0.000    1.402    0.200 <ipython-input-7-96f3d7189345>:15(sleep2)
        7    0.000    0.000    2.103    0.300 <ipython-input-7-96f3d7189345>:18(sleep3)
        3    0.000    0.000    0.000    0.000 /usr/lib/python3.6/random.py:173(randrange)
        3    0.000    0.000    0.000    0.000 /usr/lib/python3.6/random.py:223(_randbelow)
        3    0.000    0.000    0.000    0.000 /usr/lib/python3.6/random.py:217(randint)
        1    0.000    0.000    0.000    0.000 /usr/lib/python3.6/cProfile.py:50(create_stats)
        4    0.000    0.000    0.000    0.000 {method 'getrandbits' of '_random.Random' objects}
        3    0.000    0.000    0.000    0.000 {method 'bit_length' of 'int' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


<pstats.Stats at 0x7f5d60f80a20>
```

The output above is quite useful and can take you a long way. However, what if you don't know what kind of data inputs cause a bottleneck in your application? What if you're interested in aggregating and evaluating profiling data over some period of time? What if you want to profile your application while your team is dogfooding it? I found that there isn't an easy way to use this data in an ETL pipeline where you'd be able to do further offline analysis over a larger dataset.

I recently made my first open source cPython contribution which adds a dataclass called `StatsProfile` to address this. After you created your stats object, you can retrieve all the information in the above screenshot by calling `ps.get_stats_profile()` and analyze it in a programmatic way.

If you're not on Python3.9 yet, the following code snippet is a slightly modified version of the code in the pull request that you can start using today by importing it directly into your project.

```python
from pstats import func_std_string, f8
from dataclasses import dataclass
from typing import Dict

@dataclass(unsafe_hash=True)
class FunctionProfile:
    ncalls: int
    tottime: float
    percall_tottime: float
    cumtime: float
    percall_cumtime: float
    file_name: str
    line_number: int

@dataclass(unsafe_hash=True)
class StatsProfile:
    '''Class for keeping track of an item in inventory.'''
    total_tt: float
    func_profiles: Dict[str, FunctionProfile]

def get_stats_profile(stats):
    """This method returns an instance of StatsProfile, which contains a mapping
    of function names to instances of FunctionProfile. Each FunctionProfile
    instance holds information related to the function's profile such as how
    long the function took to run, how many times it was called, etc...
    """
    func_list = stats.fcn_list[:] if stats.fcn_list else list(stats.stats.keys())
    if not func_list:
        return StatsProfile(0, {})
```

```
    total_tt = float(f8(stats.total_tt))
    func_profiles = {}
    stats_profile = StatsProfile(total_tt, func_profiles)

    for func in func_list:
        cc, nc, tt, ct, callers = stats.stats[func]
        file_name, line_number, func_name = func
        ncalls = str(nc) if nc == cc else (str(nc) + '/' + str(cc))
        tottime = float(f8(tt))
        percall_tottime = -1 if nc == 0 else float(f8(tt/nc))
        cumtime = float(f8(ct))
        percall_cumtime = -1 if cc == 0 else float(f8(ct/cc))
        func_profile = FunctionProfile(
            ncalls,
            tottime, # time spent in this function alone
            percall_tottime,
            cumtime, # time spent in the function plus all functions that this function
            percall_cumtime,
            file_name,
            line_number
        )
        func_profiles[func_name] = func_profile

    return stats_profile
```

Now, rather than inspecting the profile of a single execution of our code snippet, we can aggregate and analyze the profiles over several different iterations. In a real production service, depending on which logging tool you use, you would likely need to format and stringify the `StatsProfile` dataclass before logging it, but for the purposes of this example, everything is stored in memory.

To simulate timestamped logging, `(timestamp, stats_profile)` tuples are appended to a `timestamped_stats_profile` list with every execution of the loop.

```
import cProfile, pstats
import time
from random import randint

START_TIME = int(time.time())

timestamped_stats_profiles = []
```

```python

def sleep1():
    time.sleep(0.1)

def sleep2():
    time.sleep(0.2)

def sleep3():
    time.sleep(0.3)

for _ in range(10):
    pr = cProfile.Profile()
    pr.enable()

    for _ in range(randint(1, 10)):
        sleep1()

    for _ in range(randint(1, 10)):
        sleep2()

    for _ in range(randint(1, 10)):
        sleep3()

    pr.create_stats()
    ps = pstats.Stats(pr)

    stats_profile = get_stats_profile(ps)
    timestamped_stats_profiles.append((int(time.time()), stats_profile))
```

After the data is logged, it needs to be aggregated over a certain timeslice. Most logging/visualization platforms have their functions to process timeseries data, so this would be platform specific. Sumologic has the timeslice function, Elasticsearch has examples of how to do date histogram aggregation, Datadog has an aggregate across time dropdown, etc…

For the purposes of this example, I'm doing the aggregation manually in python. I bucket all the logged (i.e. saved) `StatsProfile` objects over 10 second intervals, aggregate the cumulative execution time, `cumtime` , per function call and store the resultant counters in `time_slices_counters` . If you're interested in inspecting the number of calls to certain functions rather than the cumulative execution time spent in it, you would simply modify the parameter being access on `line 21` in the code snippet below.

```
1  from collections import Counter
2  import itertools
3
4  TIME_SLICE = 10 # Aggregate logs every 10 seconds
5
6  def time_to_bucket(time):
7      return (time-START_TIME) // TIME_SLICE
8
9  def bucket_to_time(bucket):
10     return bucket * TIME_SLICE + START_TIME
11
12 time_sliced_counters = []
13 headers = set()
14
15 for bucket, grp in itertools.groupby(timestamped_stats_profiles,key=lambda timestamped_s
16     time_slice_counter = Counter()
17     for (timestamp, stats_profile) in grp:
18         for f_name, f_profile in stats_profile.func_profiles.items():
19             f_name = f_name.lstrip("_") # matplotlib can't allow legend values to start
20             headers.add(f_name)
21             time_slice_counter[f_name] += f_profile.cumtime
22     time_sliced_counters.append((bucket_to_time(bucket), time_slice_counter))
```

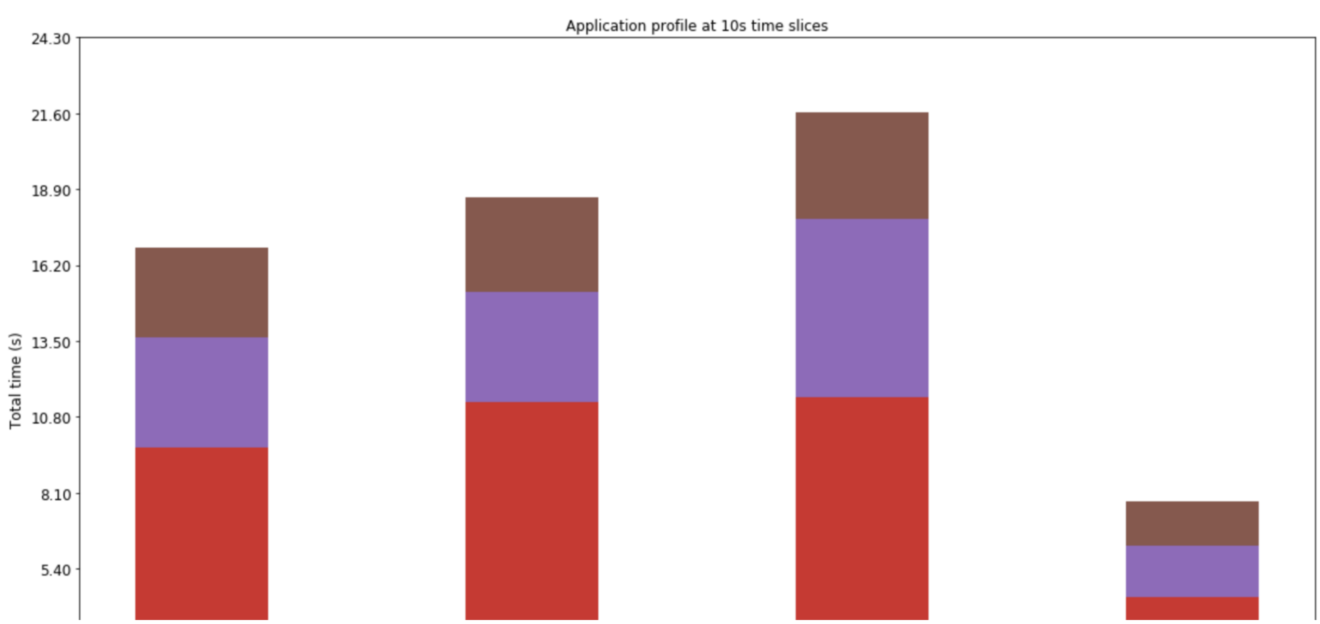
get_stats_profile_log_aggregation.py hosted with ❤ by GitHub view raw

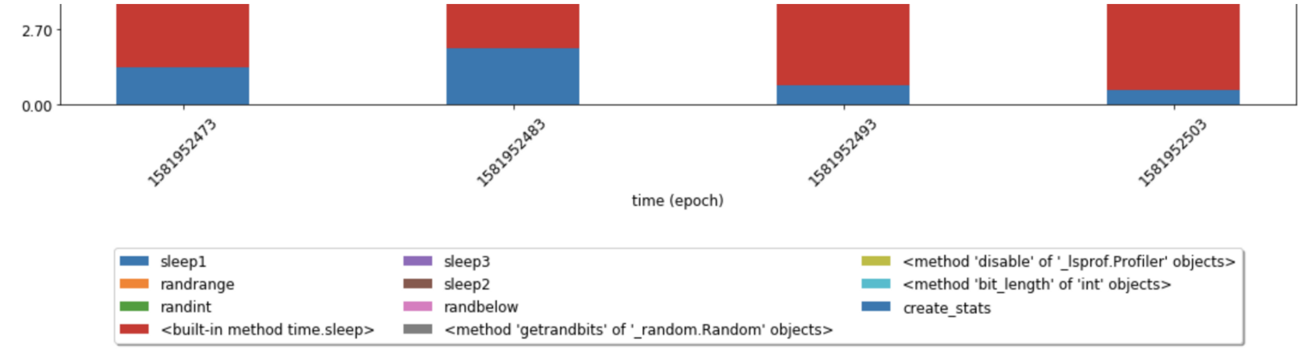
In my opinion, a stacked bar graph is a great way to visualize and easily interpret this data. Using the following code snippet:

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.ticker import FormatStrFormatter

WIDTH = 0.4

ind = np.arange(len(time_sliced_counters))

x_axis = tuple(time for (time, c) in time_sliced_counters)
y_axis = [[] for _ in range(len(headers))]
for idx, header in enumerate(headers):
```

```
    for (time, counter) in time_sliced_counters:
        y_axis[idx].append(counter[header])

fig = matplotlib.pyplot.gcf()
fig.set_size_inches(18.5, 10.5)

boxes = []
titles = []
bottom = np.zeros(len(y_axis[0]))
for idx, header in enumerate(headers):
    p = plt.bar(ind, tuple(y_axis[idx]), WIDTH, bottom=bottom)
    bottom += np.array(y_axis[idx])
    boxes.append(p[0])
    titles.append(header)

plt.ylabel("Total time (s)", fontsize=12)
plt.yticks(np.arange(0, round(max(bottom) + 5), 1), fontsize=12)
plt.gca().yaxis.set_major_formatter(FormatStrFormatter('%.2f'))

plt.xlabel('time (epoch)', fontsize=12)
plt.xticks(ind, x_axis, fontsize=12, rotation=45)

plt.title(f"Application profile at {TIME_SLICE}s time slices")
plt.legend(tuple(boxes), tuple(titles), fontsize=12, ncol=3, fancybox=True, loc='upper c

plt.show()
```

We can generate a graph that looks like this:





The results aren't very interesting or surprising given the simplicity of a script calling `sleep` a bunch of times, but hopefully it'll be more useful in more complex applications.

. . .

It's important to note that you probably should not be doing this in production. It could be useful on your local or development environments, and might be worth enabling in a single canary, but could have adverse effects in prod. You would be polluting your logs with large `StatsProfile` structures, and I have not investigated if running `cProfile` in prod could potentially downgrade your service's performance.

. . .

As a side note, though there is some overhead and a small learning curve, I was very pleased with how easy it is to contribute to cPython. Aside from publishing the actual PR, you have to sign the PSF Contributor Agreement, open a on bugs.python.org, and nudge a few people to make get your code looked at. There is a great developer guide on how to run things locally and execute tests. I recently also came by this doc, which is a good starting point if you've never contributed to cPython before.

Huge thanks Gregory P. Smith for reviewing and approving my cPython PR! Also, thank you to Сергей Яркин for proofreading my article, and a special shoutout to Manuel Dell'Elce who built a really nice chrome extension that made embedding code snippets in this medium article a breeze.

Python Profiling Stacked Bar Chart Stats