DD2358 – Using the `cProfile` Module

Stefano Markidis

KTH Royal Institute of Technology

Intended Learning Outcomes

- To obtain profiling information about the most used functions in the code with the `cProfile` Python tool
- To analyze the `cProfile` output and statistics and understand the impact of different function on the overall application timing
- Use the `SnakeViz` tool to visualize the results of `cProfile`

The `cProfile` tool

- `cProfile` is a built-in profiling tool in the standard library.
  - It hooks into the python interpreter in CPython to measure the time taken to run every function that it sees.
- `profile` is the original and slower pure Python profiler
- `cProfile` has the same interface as profile and is **written in C for a lower overhead**.
- If you're curious about the history of these libraries, see Armin Rigo's 2005 request to include `cProfile` in the standard library.

[Python-Dev] s/hotshot/lsprof

Armin Rigo arigo at tunes.org
Sat Nov 19 19:08:55 CET 2005

- Previous message: [Python-Dev] How to stay almost backwards compatible with al
- Next message: [Python-Dev] s/hotshot/lsprof
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Hi!

The current Python profilers situation is a mess.

'profile.Profile' is the ages-old pure Python profiler.  At the end of a
run, it builds a dict that is inspected by 'pstats.Stats'.  It has some
recent support for profiling C calls, which however make it crash in
some cases [1].  And of course it's slow (makes a run take about 10x
longer).

'hotshot', new from 2.2, is quite faster (reportedly, only 30% added
overhead).  The log file is then loaded and turned into an instance of
```

Hypothesis when Profiling

- A good practice when profiling is to generate a ***hypothesis*** about the speed of parts of your code before you profile it.
- Let's hypothesize that `calculate_z_serial_purepython` is the slowest part of the code.
  - In that function, we do a lot of **dereferencing** and make **many calls to basic arithmetic operators** and the **abs** function.
    - *These will probably show up as consumers of CPU resources.*
- Here, we'll use the `cProfile` module to run the code.
  - The output is spartan but helps us figure out where to analyze further.

```
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while abs(z) < 2 and n < maxiter:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

```
python -m cProfile -s cumulative JuliaSet.py
```

The -s cumulative flag tells `cProfile` to **sort by cumulative time spent** inside each function; this gives us a view into the slowest parts of a section of code. The `cProfile output` is written to screen directly after our usual print results

```
stef@Stefs-MacBook-Air Codes % python -m cProfile -s cumulative JuliaSet.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 4.945113182067871 seconds
         36221995 function calls in 5.263 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    5.263    5.263 {built-in method builtins.exec}
        1    0.013    0.013    5.263    5.263 JuliaSet.py:1(<module>)
        1    0.252    0.252    5.249    5.249 JuliaSet.py:21(calc_pure_python)
        1    3.975    3.975    4.945    4.945 JuliaSet.py:59(calculate_z_serial_purepython)
 34219980    0.970    0.000    0.970    0.000 {built-in method builtins.abs}
  2002000    0.048    0.000    0.048    0.000 {method 'append' of 'list' objects}
        1    0.004    0.004    0.004    0.004 {built-in method builtins.sum}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

```
stef@Stefs-MacBook-Air Codes % python -m cProfile -s cumulative JuliaSet.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 4.945113182067871 seconds
         36221995 function calls in 5.263 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    5.263    5.263 {built-in method builtins.exec}
        1    0.013    0.013    5.263    5.263 JuliaSet.py:1(<module>)
        1    0.252    0.252    5.249    5.249 JuliaSet.py:21(calc_pure_python)
        1    3.975    3.975    4.945    4.945 JuliaSet.py:59(calculate_z_serial_purepython)
 34219980    0.970    0.000    0.970    0.000 {built-in method builtins.abs}
  2002000    0.048    0.000    0.048    0.000 {method 'append' of 'list' objects}
        1    0.004    0.004    0.004    0.004 {built-in method builtins.sum}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- We can see that the entry point to the code `JuliasSet.py` on line 1 takes a total of 5.2 seconds.
- This is just the `__main__` call to `calc_pure_python`.
- ncalls is 1, indicating that this line is executed only once.

```
stef@Stefs-MacBook-Air Codes % python -m cProfile -s cumulative JuliaSet.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 4.945113182067871 seconds
         36221995 function calls in 5.263 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    5.263    5.263 {built-in method builtins.exec}
        1    0.013    0.013    5.263    5.263 JuliaSet.py:1(<module>)
        1    0.252    0.252    5.249    5.249 JuliaSet.py:21(calc_pure_python)
        1    3.975    3.975    4.945    4.945 JuliaSet.py:59(calculate_z_serial_purepython)
 34219980    0.970    0.000    0.970    0.000 {built-in method builtins.abs}
  2002000    0.048    0.000    0.048    0.000 {method 'append' of 'list' objects}
        1    0.004    0.004    0.004    0.004 {built-in method builtins.sum}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- Inside calc_pure_python, the call to `calculate_z_serial_purepython` consumes 4.95 seconds.
  - *Both functions are called only once.*
- We can derive that approximately 0.3 second is spent on lines of code inside calc_pure_python, separate to calling the CPU-intensive calculate_z_serial_purepython function.
  - However, we can't derive *which* lines take the time inside the function using cProfile.

```
stef@Stefs-MacBook-Air Codes % python -m cProfile -s cumulative JuliaSet.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 4.945113182067871 seconds
         36221995 function calls in 5.263 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    5.263    5.263 {built-in method builtins.exec}
        1    0.013    0.013    5.263    5.263 JuliaSet.py:1(<module>)
        1    0.252    0.252    5.249    5.249 JuliaSet.py:21(calc_pure_python)
        1    3.975    3.975    4.945    4.945 JuliaSet.py:59(calculate_z_serial_purepython)
 34219980    0.970    0.000    0.970    0.000 {built-in method builtins.abs}
  2002000    0.048    0.000    0.048    0.000 {method 'append' of 'list' objects}
        1    0.004    0.004    0.004    0.004 {built-in method builtins.sum}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- Inside `calculate_z_serial_purepython`, the time spent on lines of code (without calling other functions) is 3.9 seconds.
- This function makes 34,219,980 calls to `abs`, which take a total of 0.9 seconds, along with other calls that do not cost much time.

```
stef@Stefs-MacBook-Air Codes % python -m cProfile -s cumulative JuliaSet.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 4.945113182067871 seconds
         36221995 function calls in 5.263 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    5.263    5.263 {built-in method builtins.exec}
        1    0.013    0.013    5.263    5.263 JuliaSet.py:1(<module>)
        1    0.252    0.252    5.249    5.249 JuliaSet.py:21(calc_pure_python)
        1    3.975    3.975    4.945    4.945 JuliaSet.py:59(calculate_z_serial_purepython)
 34219980    0.970            0.970    0.000 {built-in method builtins.abs}
  2002000    0.048    0.000    0.048    0.000 {method 'append' of 'list' objects}
        1    0.004    0.004    0.004    0.004 {built-in method builtins.sum}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- While the per-call cost is negligible (it is recorded as 0.000 seconds), the total time for 34,219,980 calls is 0.97 seconds.
- We couldn’t predict in advance exactly how many calls would be made to abs, as the Julia function has unpredictable dynamics

```
stef@Stefs-MacBook-Air Codes % python -m cProfile -s cumulative JuliaSet.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 4.945113182067871 seconds
         36221995 function calls in 5.263 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    5.263    5.263 {built-in method builtins.exec}
        1    0.013    0.013    5.263    5.263 JuliaSet.py:1(<module>)
        1    0.252    0.252    5.249    5.249 JuliaSet.py:21(calc_pure_python)
        1    3.975    3.975    4.945    4.945 JuliaSet.py:59(calculate_z_serial_purepython)
 34219980    0.970    0.000    0.970    0.000 {built-in method builtins.abs}
  2002000          0.000    0.048    0.000 {method 'append' of 'list' objects}
        1    0.004    0.004    0.004    0.004 {built-in method builtins.sum}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- The next line in the profiled output, `{method 'append' of 'list' objects}`, details the creation of 2,002,000 list items.
- This creation of 2,002,000 items is occurring in `calc_pure_python` during the setup phase.
- The `zs` and `cs` lists will be 1000*1000 items each (generating 1,000,000 * 2 calls), and these are built from a list of 1,000 *x* and 1,000 *y* coordinates. In total, this is 2,002,000 calls to append.

```
stef@Stefs-MacBook-Air Codes % python -m cProfile -s cumulative JuliaSet.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 4.945113182067871 seconds
         36221995 function calls in 5.263 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    5.263    5.263 {built-in method builtins.exec}
        1    0.013    0.013    5.263    5.263 JuliaSet.py:1(<module>)
        1    0.252    0.252    5.249    5.249 JuliaSet.py:21(calc_pure_python)
        1    3.975    3.975    4.945    4.945 JuliaSet.py:59(calculate_z_serial_purepython)
 34219980    0.970    0.000    0.970    0.000 {built-in method builtins.abs}
  2002000    0.048    0.000    0.048    0.000 {method 'append' of 'list' objects}
        1    0.004    0.004    0.004    0.004 {built-in method builtins.sum}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- The final line of the profiling output refers to `lsprof`
  - This is the original name of the tool that evolved into `cProfile` and can be ignored.

Write a Statistics File with cProfile

1 `python -m cProfile -o` **`profile.stats`** `JuliaSet.py` to obtain output file

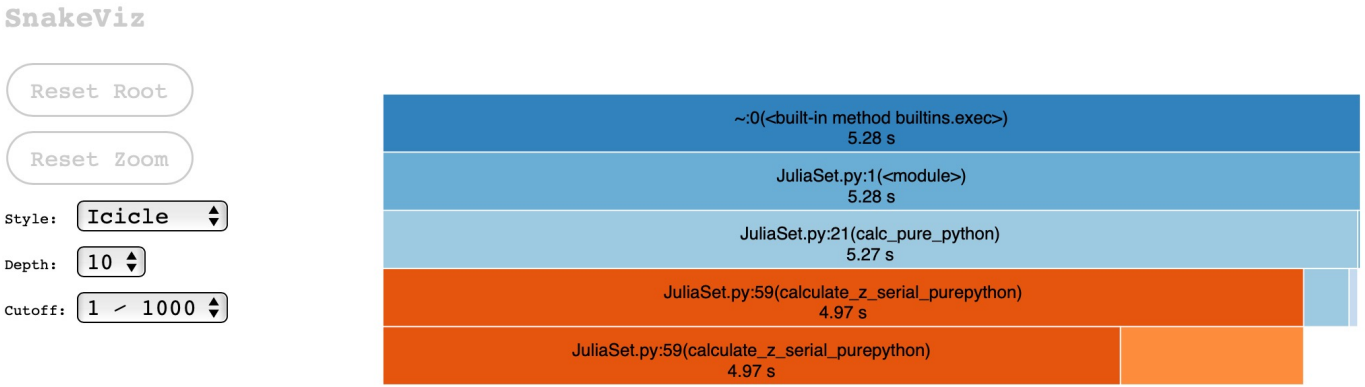
```
stef@Stefs-MacBook-Air Codes % python
Python 3.8.9 (default, Aug  3 2021, 19:21:54)
[Clang 13.0.0 (clang-1300.0.29.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import pstats
>>> p = pstats.Stats("profile.stats")
>>> p.sort_stats("cumulative")
<pstats.Stats object at 0x100439910>
>>> p.print_stats()
Sun Jan 16 15:41:03 2022    profile.stats

         36221995 function calls in 5.284 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    5.284    5.284 {built-in method builtins.exec}
        1    0.014    0.014    5.284    5.284 JuliaSet.py:1(<module>)
        1    0.245    0.245    5.270    5.270 JuliaSet.py:21(calc_pure_python)
        1    3.983    3.983    4.973    4.973 JuliaSet.py:59(calculate_z_serial_purepython)
 34219980    0.990    0.000    0.990    0.000 {built-in method builtins.abs}
  2002000    0.048    0.000    0.048    0.000 {method 'append' of 'list' objects}
        1    0.004    0.004    0.004    0.004 {built-in method builtins.sum}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


<pstats.Stats object at 0x100439910>
```

2 Python to read the output `profile.stats`

Visualizing `cProfile` Output with `SnakeViz`

- `snakeviz` is a visualizer that draws the output of `cProfile` as a diagram in which larger boxes are areas of code that take longer to run.
- Use `snakeviz` to get a high-level understanding of a `cProfile` statistics file, particularly if you're investigating a new project for which you have little intuition.
  - The diagram will help you visualize the CPU-usage behavior of the system, and it may highlight areas that you hadn't expected to be expensive.
- To install `SnakeViz`, use `$ pip install snakeviz`
- To run `python -m snakeviz profile.stats` **`--server`**
- You will need the browser to open page given by `snakeviz`

SnakeViz Output



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	3.983	3.983	4.973	4.973	JuliaSet.py:59(calculate_z_serial_purepython)
34219980	0.9902	2.894e-08	0.9902	2.894e-08	~:0(<built-in method builtins.abs>)
1	0.2446	0.2446	5.27	5.27	JuliaSet.py:21(calc_pure_python)
2002000	0.04771	2.383e-08	0.04771	2.383e-08	~:0(<method 'append' of 'list' objects>)
1	0.01426	0.01426	5.284	5.284	JuliaSet.py:1(<module>)
1	0.004439	0.004439	0.004439	0.004439	~:0(<built-in method builtins.sum>)
3	9.175e-05	3.058e-05	9.175e-05	3.058e-05	~:0(<built-in method builtins.print>)
2	2.875e-06	1.438e-06	2.875e-06	1.438e-06	~:0(<built-in method time.time>)
1	1.417e-06	1.417e-06	5.284	5.284	~:0(<built-in method builtins.exec>)
4	5.84e-07	1.46e-07	5.84e-07	1.46e-07	~:0(<built-in method builtins.len>)
1	3.33e-07	3.33e-07	3.33e-07	3.33e-07	~:0(<method 'disable' of '_lsprof.Profiler' objects>)

To Summarize

- `cProfile` is a built-in Python profiler to measure the time taken to run every function.
- `cProfile` is ideal to find the most computational-intensive parts of the code.
- It is typically called from the command line with `-m cProfile` and can print to stdout or to a file
- The ouput file can be visualized with the `SnakeViz` tool.