Python Profiling

Feb 29, 2020

Demo:
https://nbviewer.jupyter.org/github/chaupmcs/show_cpu_mem_info/blob/master/CPU_MEM_info.ipynb?flush_cache=true

How to use python to systems information

- **psutil**: https://psutil.readthedocs.io/en/latest/index.html#cpu

  A cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python. It is useful mainly for system monitoring, profiling, limiting process resources and the management of running processes. It implements many functionalities offered by UNIX command-line tools such as *ps, top, lsof, netstat, ifconfig, kill,* etc. **psutil** currently supports many platforms.

  Python: *pip3 install psutil*

Overall of OS

- **Memory**
  - **total**: total physical memory (exclusive swap).
  - **available**: the memory that can be given instantly to processes without the system going into swap. This is calculated by summing different memory values depending on the platform and it is supposed to be used to monitor actual memory usage in a cross platform fashion.
  - **used**: memory used, calculated differently depending on the platform and designed for informational purposes only. **total - free** does not necessarily match **used**.
  - **free**: memory not being used at all (zeroed) that is readily available; note that this doesn't reflect the actual memory available (use **available** instead). **total - used** does not necessarily match **free**.
  - **active** *(UNIX)*: memory currently in use or very recently used, and so it is in RAM.
  - **inactive** *(UNIX)*: memory that is marked as not used.
  - **buffers** *(Linux, BSD)*: cache for things like file system metadata.
  - **cached** *(Linux, BSD)*: cache for various things.
  - **shared** *(Linux, BSD)*: memory that may be simultaneously accessed by multiple processes.
  - **slab** *(Linux)*: in-kernel data structures cache.
  - **wired** *(BSD, macOS)*: memory that is marked to always stay in RAM. It is never moved to disk.
  - **swap_memory**
- **Disk**
  - Disk partitions

- Disk usage
- Disk IO count
    - **read_count**: number of reads
    - **write_count**: number of writes
    - **read_bytes**: number of bytes read
    - **write_bytes**: number of bytes written

  Platform-specific fields:

    - **read_time**: (all except *NetBSD* and *OpenBSD*) time spent reading from disk (in milliseconds)
    - **write_time**: (all except *NetBSD* and *OpenBSD*) time spent writing to disk (in milliseconds)
    - **busy_time**: (*Linux*, *FreeBSD*) time spent doing actual I/Os (in milliseconds)
    - **read_merged_count** (*Linux*): number of merged reads (see iostats doc)
    - **write_merged_count** (*Linux*): number of merged writes (see iostats doc)

```
psutil.disk_io_counters()
```

```
sdiskio(read_count=11080428, write_count=4114846, read_bytes=269957799424, write_bytes=288751243264, read_time=212749
345, write_time=37767288)
```

- **CPU**
    - Num physical & logical cores

```
##  #logical cores= #physical cores * #threads of each core (Hyper thread)
psutil.cpu_count(logical=True)
```

```
4
```

```
psutil.cpu_count(logical=False) #return the number of physical cores only
```

```
4
```

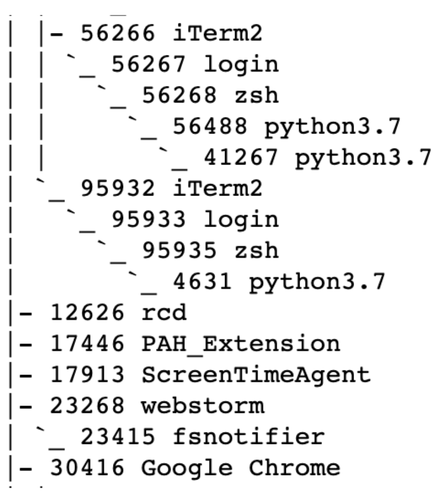
    - CPU Time
        - **user**: time spent by normal processes executing in user mode; on Linux this also includes **guest** time
        - **system**: time spent by processes executing in kernel mode
        - **idle**: time spent doing nothing
        - Some more, depends on platform-specific fields (nice, iowait, etc.)
    - CPU percent: Return a float representing the current system-wide CPU utilization as a percentage
    - CPU start: Return various CPU statistics as the number of context switches, the number of interrupts since boot, etc.
    - CPU frequency
- **Network**
- **Sensors**
    - Hardware temperatures

- Fan speed
- Battery

> Note: *Sensors* depend heavily on OS/Hardware Devices to get these info. Most of them are not available for MacOS (except a workaround using *osx-cpu-temp* for CPU temperature).

- Processes

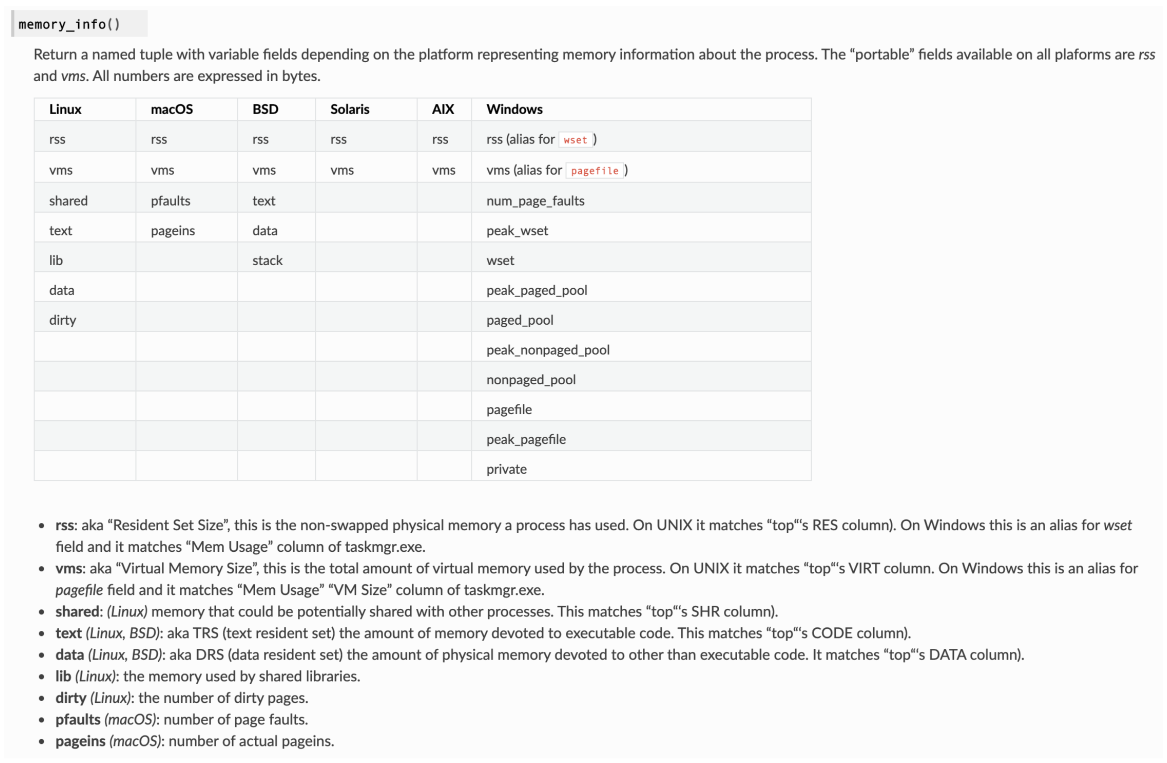
  Show the tree of all processes (PID, name)



```
| |- 56266 iTerm2
| | `_ 56267 login
| |   `_ 56268 zsh
| |     `_ 56488 python3.7
| |       `_ 41267 python3.7
| `_ 95932 iTerm2
|   `_ 95933 login
|     `_ 95935 zsh
|       `_ 4631 python3.7
|- 12626 rcd
|- 17446 PAH_Extension
|- 17913 ScreenTimeAgent
|- 23268 webstorm
| `_ 23415 fsnotifier
|- 30416 Google Chrome
```

For each process

- PID: Process ID
- Memory



```
memory_info()
```

Return a named tuple with variable fields depending on the platform representing memory information about the process. The "portable" fields available on all plaforms are *rss* and *vms*. All numbers are expressed in bytes.

Linux	macOS	BSD	Solaris	AIX	Windows
rss	rss	rss	rss	rss	rss (alias for `wset`)
vms	vms	vms	vms	vms	vms (alias for `pagefile`)
shared	pfaults	text			num_page_faults
text	pageins	data			peak_wset
lib		stack			wset
data					peak_paged_pool
dirty					paged_pool
					peak_nonpaged_pool
					nonpaged_pool
					pagefile
					peak_pagefile
					private

- **rss**: aka "Resident Set Size", this is the non-swapped physical memory a process has used. On UNIX it matches "top"'s RES column). On Windows this is an alias for *wset* field and it matches "Mem Usage" column of taskmgr.exe.
- **vms**: aka "Virtual Memory Size", this is the total amount of virtual memory used by the process. On UNIX it matches "top"'s VIRT column. On Windows this is an alias for *pagefile* field and it matches "Mem Usage" "VM Size" column of taskmgr.exe.
- **shared**: *(Linux)* memory that could be potentially shared with other processes. This matches "top"'s SHR column).
- **text** *(Linux, BSD)*: aka TRS (text resident set) the amount of memory devoted to executable code. This matches "top"'s CODE column).
- **data** *(Linux, BSD)*: aka DRS (data resident set) the amount of physical memory devoted to other than executable code. It matches "top"'s DATA column).
- **lib** *(Linux)*: the memory used by shared libraries.
- **dirty** *(Linux)*: the number of dirty pages.
- **pfaults** *(macOS)*: number of page faults.
- **pageins** *(macOS)*: number of actual pageins.

- Thread
- Status
- Connection
- CPU

Examples:

```
py.memory_full_info()
```

```
pfullmem(rss=152358912, vms=4909830144, pfaults=41647, pageins=855, uss=133885952)
```

```
py.memory_percent()
```

```
1.7775535583496094
```

```
py.is_running()
```

```
True
```

```
py.cpu_percent()
```

```
0.0
```

```
py.children()
```

```
[]
```

```
py.status()
```

```
'running'
```

```
py.parent()
```

```
psutil.Process(pid=56488, name='python3.7', started='2020-02-28 11:21:49')
```

```
py.num_threads()
```

```
16
```

Basic packages to profile python code

- **cProfile** is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs.
- **profile** a pure Python module whose interface is imitated by cProfile

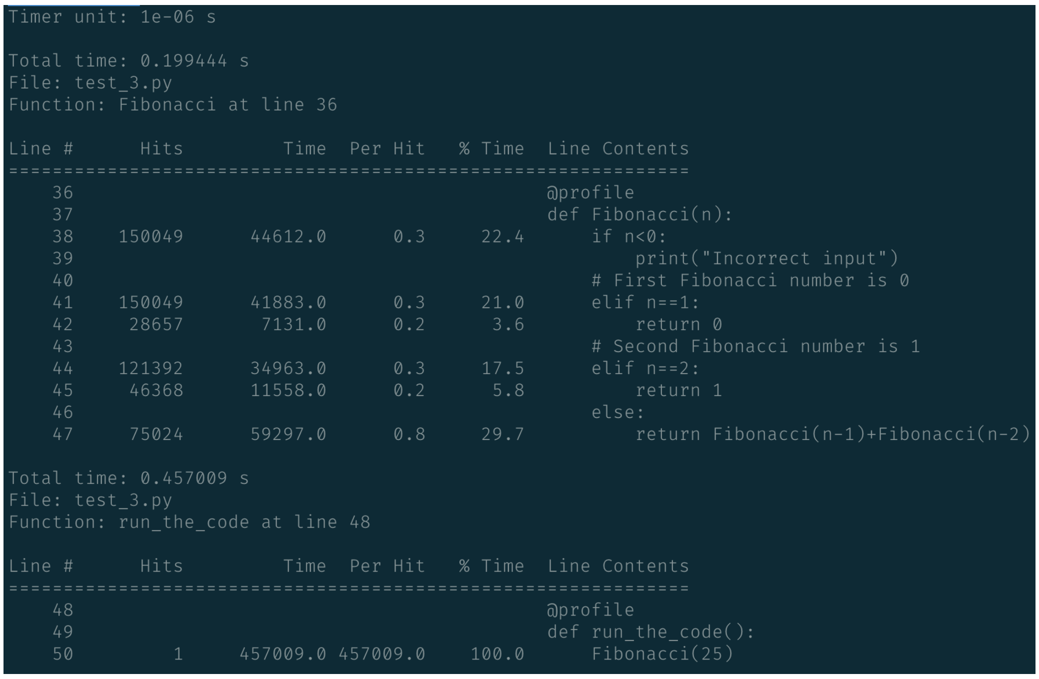
```
import cProfile
cProfile.run('Fibonacci(35)')
```

```
         18454932 function calls (4 primitive calls) in 4.718 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
18454929/1    4.718    0.000    4.718    4.718 <ipython-input-34-a865922b7858>:42(Fibonacci)
        1    0.000    0.000    4.718    4.718 <string>:1(<module>)
        1    0.000    0.000    4.718    4.718 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

The **profile** and **cProfile** modules export the same interface, so they are mostly interchangeable; **cProfile** has a much lower overhead compared to **Profile**. *https://docs.python.org/2/library/profile.html*

- **line_profiler**: https://github.com/rkern/line_profiler

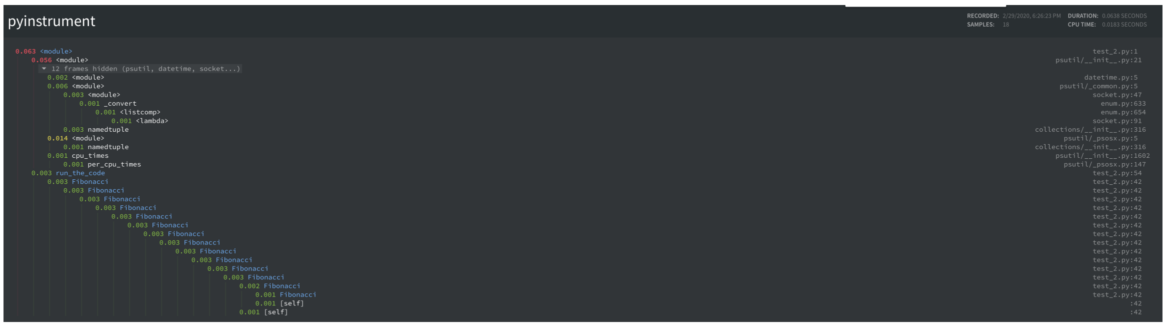
  line_profiler is a module for doing line-by-line profiling of functions.



```
Timer unit: 1e-06 s

Total time: 0.199444 s
File: test_3.py
Function: Fibonacci at line 36

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    36                                           @profile
    37                                           def Fibonacci(n):
    38    150049      44612.0      0.3     22.4      if n<0:
    39                                                   print("Incorrect input")
    40                                               # First Fibonacci number is 0
    41    150049      41883.0      0.3     21.0      elif n==1:
    42     28657       7131.0      0.2      3.6          return 0
    43                                               # Second Fibonacci number is 1
    44    121392      34963.0      0.3     17.5      elif n==2:
    45     46368      11558.0      0.2      5.8          return 1
    46                                               else:
    47     75024      59297.0      0.8     29.7          return Fibonacci(n-1)+Fibonacci(n-2)

Total time: 0.457009 s
File: test_3.py
Function: run_the_code at line 48

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    48                                           @profile
    49                                           def run_the_code():
    50         1     457009.0 457009.0    100.0      Fibonacci(25)
```

- **pyinstrument:** https://github.com/joerick/pyinstrument

  Pyinstrument is a statistical profiler - it doesn't track every function call that your program makes. Instead, it's recording the call stack every 1ms to reduce the overhead.



```
pyinstrument                              RECORDED: 2/29/2020, 6:26:23 PM   DURATION: 0.0638 SECONDS
                                          SAMPLES:  18                      CPU TIME: 0.0183 SECONDS

0.063 <module>                                                      test_2.py:1
   0.056 <module>                                                   psutil/__init__.py:21
      ▾ 12 frames hidden (psutil, datetime, socket...)
         0.002 <module>                                             datetime.py:5
         0.006 <module>                                             psutil/_common.py:5
            0.003 <module>                                          socket.py:47
               0.001 _convert                                       enum.py:633
                  0.001 <listcomp>                                  enum.py:654
                     0.001 <lambda>                                 socket.py:91
            0.003 namedtuple                                        collections/__init__.py:316
         0.014 <module>                                             psutil/_psosx.py:5
            0.001 namedtuple                                        collections/__init__.py:316
         0.001 cpu_times                                            psutil/__init__.py:1602
            0.001 per_cpu_times                                     psutil/_psosx.py:147
   0.003 run_the_code                                               test_2.py:54
      0.003 Fibonacci                                               test_2.py:42
         0.003 Fibonacci                                            test_2.py:42
            0.003 Fibonacci                                         test_2.py:42
               0.003 Fibonacci                                      test_2.py:42
                  0.003 Fibonacci                                   test_2.py:42
                     0.003 Fibonacci                                test_2.py:42
                        0.003 Fibonacci                             test_2.py:42
                           0.003 Fibonacci                          test_2.py:42
                              0.003 Fibonacci                       test_2.py:42
                                 0.003 Fibonacci                    test_2.py:42
                                    0.003 Fibonacci                 test_2.py:42
                                       0.003 Fibonacci              test_2.py:42
                                          0.002 Fibonacci           test_2.py:42
                                             0.001 Fibonacci        test_2.py:42
                                             0.001 [self]           :42
                                          0.001 [self]              :42
```

Some more visualization packages to profile Python code

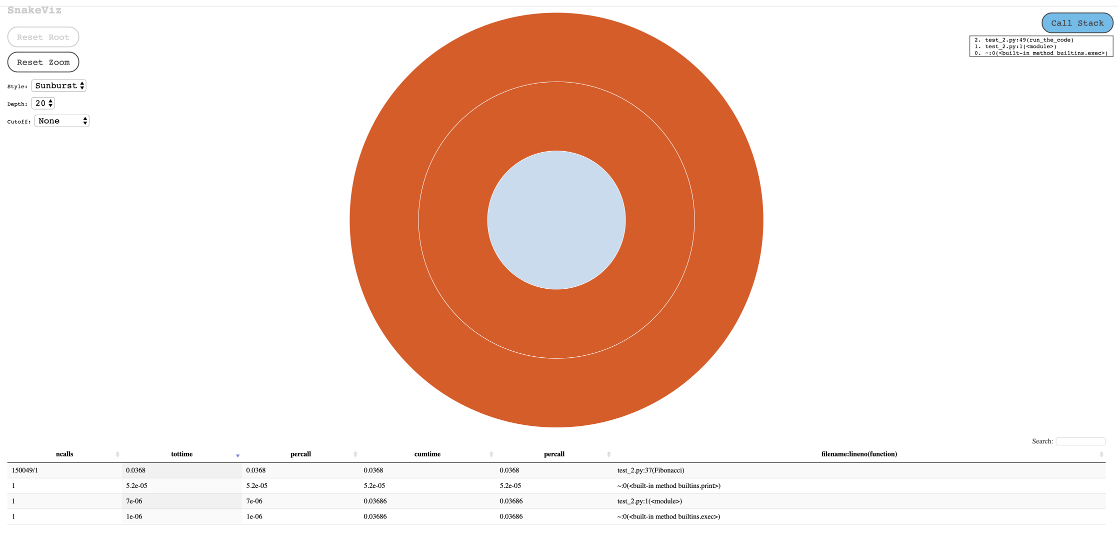
💡 Most of these tools below are wrappers of ***cProfile***

- **Snakeviz**: https://jiffyclub.github.io/snakeviz/

  SnakeViz is a browser based graphical viewer for the output of Python's **cProfile** module.

```
## Terminal
python -m cProfile -o program.prof test.py
```

```
snakeviz program.prof
```



Style: Sunburst
Depth: 20
Cutoff: None

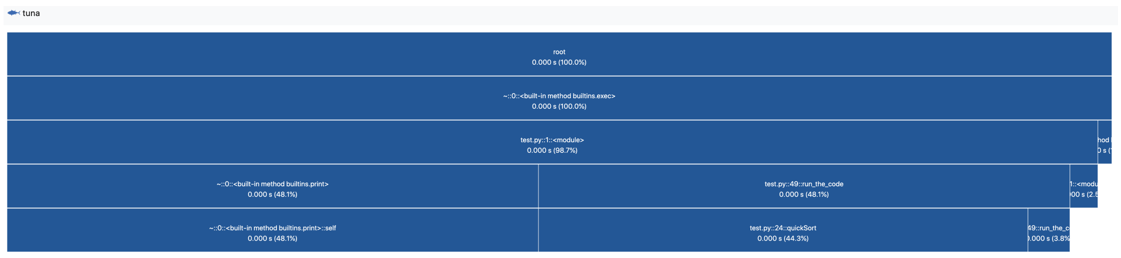
```
2. test_2.py:49(run_the_code)
1. test_2.py:1(<module>)
0. ~:0(<built-in method builtins.exec>)
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
150049/1	0.0368	0.0368	0.0368	0.0368	test_2.py:37(Fibonacci)
1	5.2e-05	5.2e-05	5.2e-05	5.2e-05	~:0(<built-in method builtins.print>)
1	7e-06	7e-06	0.03686	0.03686	test_2.py:1(<module>)
1	1e-06	1e-06	0.03686	0.03686	~:0(<built-in method builtins.exec>)

- **Tuna**: https://github.com/nschloe/tuna

  Tuna is a modern, lightweight Python profile viewer inspired by **SnakeViz**. It handles runtime and import profiles, has no Python dependencies, uses *d3* and *bootstrap*

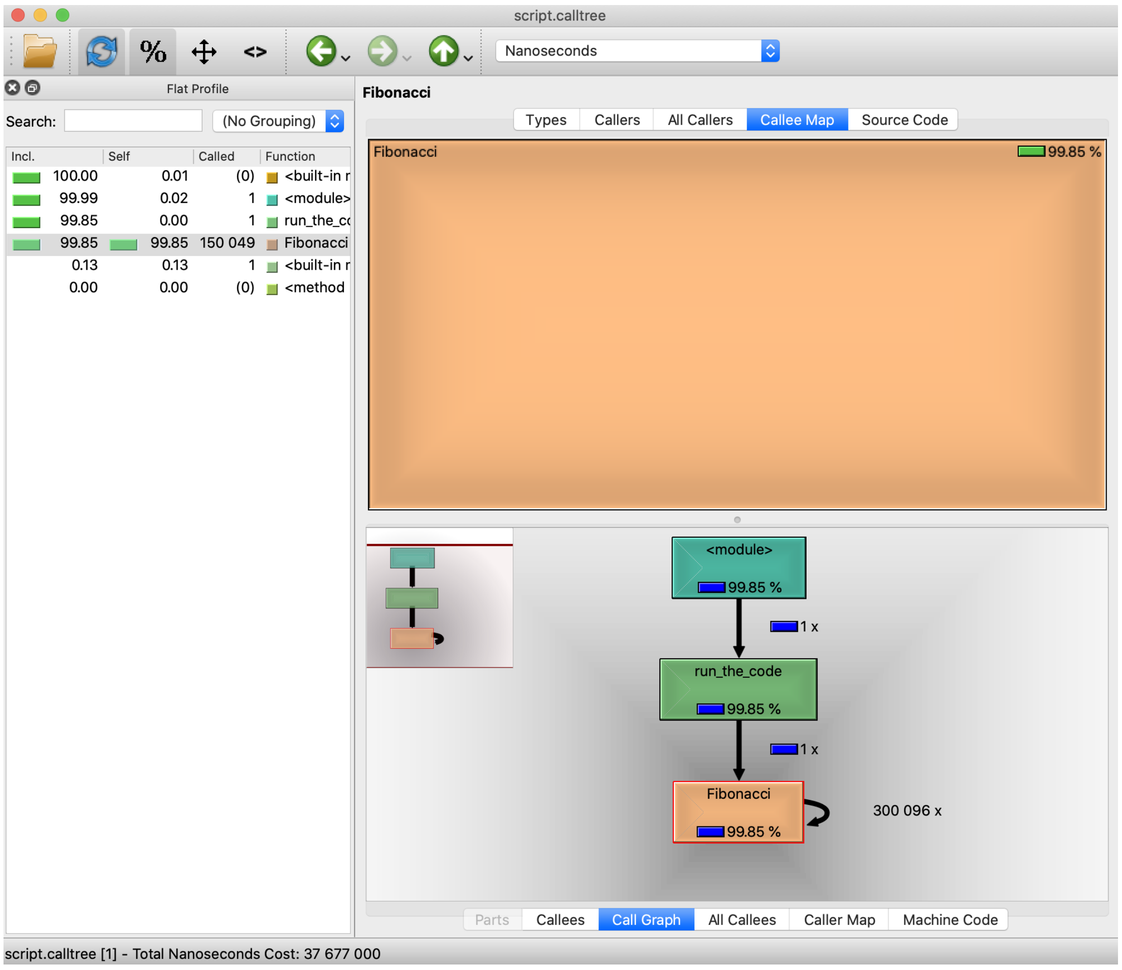
```
python -m cProfile -o program.prof test.py
tuna program.prof
```



- **pyprof2calltree:** https://github.com/pwaller/pyprof2calltree

  Script to help visualize profiling data collected with the **cProfile** Python module with the *qcachegrind* graphical calltree analyser.

```
python -m cProfile -o program.profile test.py
pyprof2calltree -i program.profile -o program.calltree
qcachegrind program.calltree
```



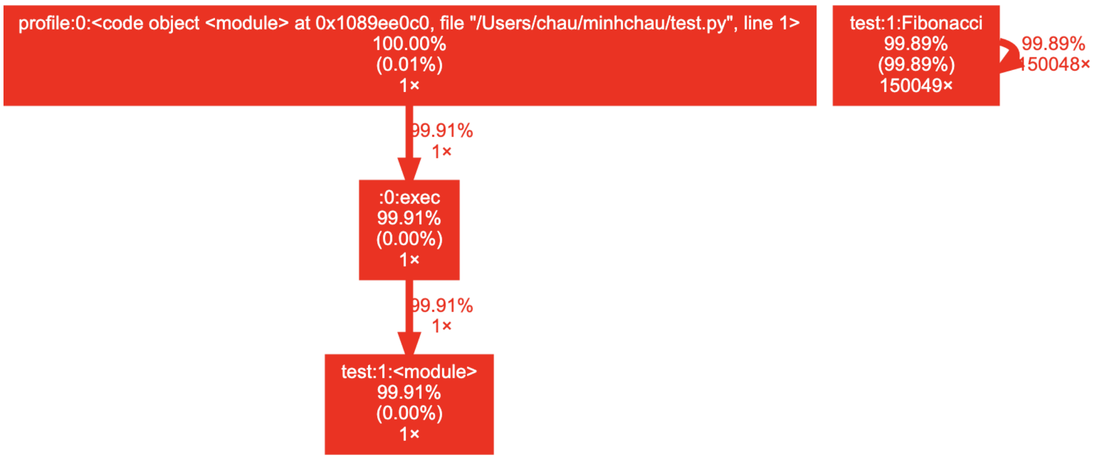
Incl.	Self	Called	Function
🟩 100.00	0.01	(0)	<built-in r
🟩 99.99	0.02	1	<module>
🟩 99.85	0.00	1	run_the_cc
🟩 99.85	🟩 99.85	150 049	Fibonacci
0.13	0.13	1	<built-in r
0.00	0.00	(0)	<method

script.calltree [1] - Total Nanoseconds Cost: 37 677 000

- **gprof2dot**: https://github.com/jrfonseca/gprof2dot

  A tool to convert the output from many profilers into a dot graph.

```
gprof2dot -f pstats program.prof | dot -Tsvg -o callgraph.svg
```



Note 1

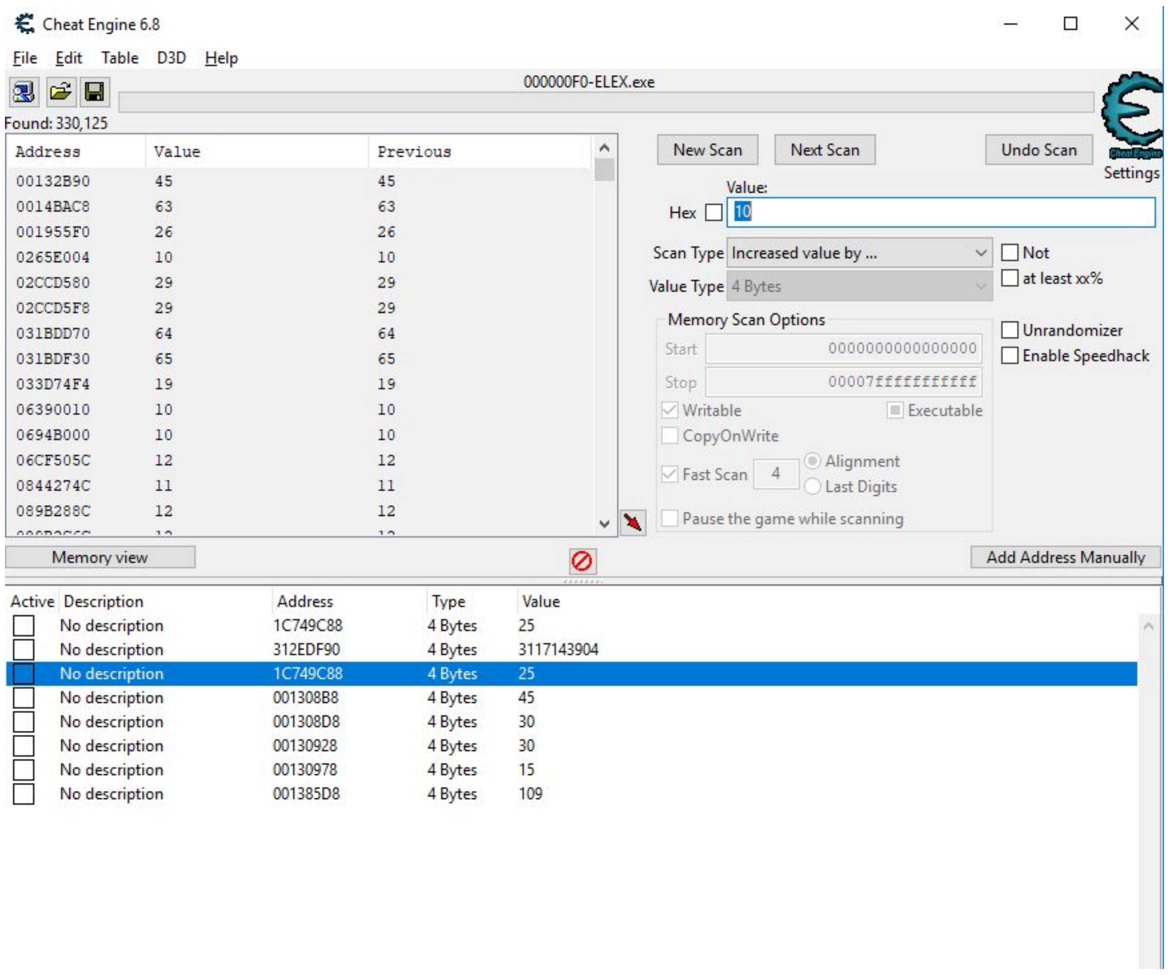
> The whole timed call tree cannot be retrieved from profile data. Python developers made the decision to only store parent data in profiles because it can be computed with little overhead. You may notice this when looking at the tree from ***pyinstrument***.

Note 2

> In python, package ***energyusage*** (https://pypi.org/project/energyusage/) can be used for measuring the environmental impact of computation, but it works with some hardware devices (RALP and Nvidia only).

Note 3

> Some softwares allow user to find and modify value in memory.
> For example: Cheat Engine (https://www.cheatengine.org/)



Found: 330,125

Address	Value	Previous
00132B90	45	45
0014BAC8	63	63
001955F0	26	26
0265E004	10	10
02CCD580	29	29
02CCD5F8	29	29
031BDD70	64	64
031BDF30	65	65
033D74F4	19	19
06390010	10	10
0694B000	10	10
06CF505C	12	12
0844274C	11	11
089B288C	12	12

Value: 10
Hex ☐
Scan Type: Increased value by ...
Value Type: 4 Bytes
☐ Not
☐ at least xx%
Start: 0000000000000000
Stop: 00007fffffffffff
✓ Writable
▨ Executable
☐ CopyOnWrite
✓ Fast Scan 4
● Alignment
○ Last Digits
☐ Pause the game while scanning
☐ Unrandomizer
☐ Enable Speedhack

Active	Description	Address	Type	Value
☐	No description	1C749C88	4 Bytes	25
☐	No description	312EDF90	4 Bytes	3117143904
☐	No description	1C749C88	4 Bytes	25
☐	No description	001308B8	4 Bytes	45
☐	No description	001308D8	4 Bytes	30
☐	No description	00130928	4 Bytes	30
☐	No description	00130978	4 Bytes	15
☐	No description	001385D8	4 Bytes	109