

## Navigation

- [index](#)
- [modules](#) |
- [next](#) |
- [previous](#) |
- [Python v2.6.4 documentation](#) »
- [The Python Standard Library](#) »
- [27. Debugging and Profiling](#) »

## 27.4. The Python Profilers¶

Copyright © 1994, by InfoSeek Corporation, all rights reserved.

Written by James Roskind. [1]

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of InfoSeek not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C) wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### 27.4.1. Introduction to the profilers¶

A *profiler* is a program that describes the run time performance of a program, providing a variety of statistics. This documentation describes the profiler functionality provided in the modules `cProfile`, `profile` and `pstats`. This profiler provides *deterministic profiling* of Python programs. It also provides a series of report generation tools to allow users to rapidly examine the results of a profile operation.

The Python standard library provides three different profilers:

`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.

New in version 2.5.

`profile`, a pure Python module whose interface is imitated by `cProfile`. 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. Copyright © 1994, by InfoSeek Corporation.

Changed in version 2.4: Now also reports the time spent in calls to built-in functions and methods.

[hotshot](#) was an experimental C module that focused on minimizing the overhead of profiling, at the expense of longer data post-processing times. It is no longer maintained and may be dropped in a future version of Python.

Changed in version 2.5: The results should be more meaningful than in the past: the timing core contained a critical bug.

The `profile` and `cProfile` modules export the same interface, so they are mostly interchangeable; `cProfile` has a much lower overhead but is newer and might not be available on all systems. `cProfile` is really a compatibility layer on top of the internal `_lsprof` module. The [hotshot](#) module is reserved for specialized usage.

### 27.4.2. Instant User's Manual¶

This section is provided for users that “don't want to read the manual.” It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile an application with a main entry point of `foo()`, you would add the following to your module:

```
import cProfile
cProfile.run('foo()')
```

(Use `profile` instead of `cProfile` if the latter is not available on your system.)

The above action would cause `foo()` to be run, and a series of informative lines (the profile) to be printed. The above approach is most useful when working with the interpreter. If you would like to save the results of a profile into a file for later examination, you can supply a file name as the second argument to the `run()` function:

```
import cProfile
cProfile.run('foo()', 'fooprof')
```

The file `cProfile.py` can also be invoked as a script to profile another script. For example:

```
python -m cProfile myscript.py
```

`cProfile.py` accepts two optional arguments on the command line:

```
cProfile.py [-o output_file] [-s sort_order]
```

`-s` only applies to standard output (`-o` is not supplied). Look in the `Stats` documentation for valid sort values.

When you wish to review the profile, you should use the methods in the `pstats` module. Typically you would load the statistics data as follows:

```
import pstats
p = pstats.Stats('fooprof')
```

The class `Stats` (the above code just created an instance of this class) has a variety of methods for manipulating and printing the data that was just read into `p`. When you ran `cProfile.run()` above, what was printed was the result of three method calls:

```
p.strip_dirs().sort_stats(-1).print_stats()
```

The first method removed the extraneous path from all the module names. The second method sorted all the entries according to the standard module/line/name string that is printed. The third method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats('cumulative').print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats('time').print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats('file').print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats('time', 'cum').print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: .5) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p` is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('fooprof')
```

Invoked as a script, the `pstats` module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using `cmd`) and interactive help.

### 27.4.3. What Is 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). In contrast, *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.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a *hook* (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify “hot loops” that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

#### 27.4.4. Reference Manual – `profile` and `cProfile`

The primary entry point for the profiler is the global function `profile.run()` (resp. `cProfile.run()`). It is typically used to create any profile information. The reports are formatted and printed using methods of the class `pstats.Stats`. The following is a description of all of these standard entry points and functions. For a more in-depth view of some of the code, consider reading the later section on Profiler Extensions, which includes discussion of how to derive “better” profilers from the classes presented, or reading the source code for these modules.

`cProfile.run(command[, filename])`

This function takes a single argument that can be passed to the `exec` statement, and an optional file name. In all cases this routine attempts to `exec` its first argument, and gather profiling statistics from the execution. If no file name is present, then this function automatically prints a simple profiling report, sorted by the standard name string (file/line/function-name) that is presented in each line. The following is a typical output from such a call:

```
2706 function calls (2004 primitive calls) in 4.504 CPU seconds
```

Ordered by: standard name

```
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
     2    0.006    0.003    0.953    0.477  pobject.py:75(save_objects)
 43/3    0.533    0.012    0.749    0.250  pobject.py:99(evaluate)
...
```

The first line indicates that 2706 calls were monitored. Of those calls, 2004 were *primitive*. We define *primitive* to mean that the call was not induced via recursion. The next line: `Ordered by: standard name`, indicates that the text string in the far right column was used to sort the output. The column headings include:

`ncalls`

for the number of calls,

`tottime`

for the total time spent in the given function (and excluding time made in calls to sub-functions),

`percall`

is the quotient of `tottime` divided by `ncalls`

`cumtime`

is the total time spent in this and all subfunctions (from invocation till exit). This figure is accurate *even* for recursive functions.

`percall`

is the quotient of `cumtime` divided by primitive calls

`filename:lineno(function)`

provides the respective data of each function

When there are two numbers in the first column (for example, 43/3), then the latter is the number of primitive calls, and the former is the actual number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

`cProfile.runcx(command, globals, locals[, filename])`

This function is similar to `run()`, with added arguments to supply the globals and locals dictionaries for the `command` string.

Analysis of the profiler data is done using the `Stats` class.

Note

The `Stats` class is defined in the `pstats` module.

`class pstats.Stats(filename[, stream=sys.stdout[, ...]])`

This class constructor creates an instance of a “statistics object” from a `filename` (or set of filenames). `Stats` objects are manipulated by methods, in order to print useful reports. You may specify an alternate output stream by giving the keyword argument, `stream`.

The file selected by the above constructor must have been created by the corresponding version of `profile` or `cProfile`. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers. If several files are provided, all the

statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing `Stats` object, the `add()` method can be used.

Changed in version 2.5: The `stream` parameter was added.

#### 27.4.4.1. The `Stats` Class

`Stats` objects have the following methods:

`Stats.strip_dirs()`

This method for the `Stats` class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If `strip_dirs()` causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

`Stats.add(filename[, ...])`

This method of the `Stats` class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of `profile.run()` or `cProfile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

`Stats.dump_stats(filename)`

Save the data loaded into the `Stats` object to a file named `filename`. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the `profile.Profile` and `cProfile.Profile` classes.

New in version 2.3.

`Stats.sort_stats(key[, ...])`

This method modifies the `Stats` object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example: 'time' or 'name').

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats('name', 'file')` will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

Valid Arg	Meaning
'calls'	call count
'cumulative'	cumulative time
'file'	file name
'module'	file name
'pcalls'	primitive call count
'line'	line number
'name'	function name
'nfl'	name/file/line
'stdname'	standard name
'time'	internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between 'nfl' and 'stdname' is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, 'nfl' does a numeric compare of the line numbers. In fact, `sort_stats('nfl')` is the same as `sort_stats('name', 'file', 'line')`.

For backward-compatibility reasons, the numeric arguments -1, 0, 1, and 2 are permitted. They are interpreted as 'stdname', 'calls', 'time', and 'cumulative' respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

`Stats.reverse_order()`

This method for the `Stats` class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

`Stats.print_stats([restriction, ...])`

This method for the `Stats` class prints out a report as described in the `profile.run()` definition.

The order of the printing is based on the last `sort_stats()` operation done on the object (subject to caveats in `add()` and `strip_dirs()`).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a regular expression (to pattern match the standard name that is printed; as of Python 1.5b1, this uses the Perl-style regular expression syntax defined by the `re` module). If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `. *foo:`. In contrast, the command:

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names `. *foo:`, and then proceed to only print the first 10% of them.

```
Stats.print_callers([restriction, ...])
```

This method for the `Stats` class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by `print_stats()`, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With `profile`, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With `cProfile`, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

```
Stats.print_callees([restriction, ...])
```

This method for the `Stats` class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the `print_callers()` method.

### 27.4.5. Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying “clock” is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the “error” will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it “takes a while” from when an event is dispatched until the profiler’s call to get the time actually gets the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock’s value was obtained (and then squirreled away), until the user’s code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it *can* accumulate and become very significant.

The problem is more important with `profile` than with the lower-overhead `cProfile`. For this reason, `profile` provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-). ) Do *not* be alarmed by negative numbers in the profile. They should *only* appear if you have calibrated your profiler, and the results are actually better than without calibration.

### 27.4.6. Calibration

The profiler of the `profile` module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see discussion in section Limitations above).

```
import profile
pr = profile.Profile()
for i in range(5):
    print pr.calibrate(10000)
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on an 800 MHz Pentium running Windows 2000, and using Python’s `time.clock()` as the timer, the magical number is about 12.5e-6.

The object of this exercise is to get a fairly consistent result. If your computer is *very* fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it: [\[2\]](#)

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will “less often” show up as negative in profile statistics.

### 27.4.7. Extensions — Deriving Better Profilers<sup>[1]</sup>

The `Profile` class of both modules, `profile` and `cProfile`, were written so that derived classes could be developed to extend the profiler. The details are not described here, as doing this successfully requires an expert understanding of how the `Profile` class works internally. Study the source code of the module carefully if you want to pursue this.

If all you want to do is change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the `Profile` class constructor:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func()`.

```
profile.Profile
```

`your_time_func()` should return a single number, or a list of numbers whose sum is the current time (like what `os.times()` returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose. For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (`os.times()` is *pretty* bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

```
cProfile.Profile
```

`your_time_func()` should return a single number. If it returns plain integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if `your_integer_time_func()` returns times measured in thousands of seconds, you would construct the `Profile` instance as follows:

```
pr = profile.Profile(your_integer_time_func, 0.001)
```

As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprof` module.

Footnotes

[1]

Updated and converted to LaTeX by Guido van Rossum. Further updated by Armin Rigo to integrate the documentation for the new `cProfile` module of Python 2.5.

[2]

Prior to Python 2.2, it was necessary to edit the profiler source code to embed the bias as a literal number. You still can, but that method is no longer described, because no longer needed.

### [Table Of Contents](#)

#### [27.4. The Python Profilers](#)

- [27.4.1. Introduction to the profilers](#)
- [27.4.2. Instant User's Manual](#)
- [27.4.3. What Is Deterministic Profiling?](#)
- [27.4.4. Reference Manual – `profile` and `cProfile`](#)
  - [27.4.4.1. The `Stats` Class](#)
- [27.4.5. Limitations](#)
- [27.4.6. Calibration](#)
- [27.4.7. Extensions — Deriving Better Profilers](#)

Previous topic

[27.2. `pdb` — The Python Debugger](#)

Next topic

[27.5. `hotshot` — High performance logging profiler](#)

This Page

- [Show Source](#)

Navigation

- [index](#)
- [modules](#) |
- [next](#) |
- [previous](#) |
- [Python v2.6.4 documentation](#) »
- [The Python Standard Library](#) »
- [27. Debugging and Profiling](#) »

© [Copyright](#) 1990-2010, Python Software Foundation.

The Python Software Foundation is a non-profit corporation. [Please donate.](#)

Last updated on Feb 26, 2010. Created using [Sphinx](#) 0.6.3.