# profile and pstats — Performance Analysis

**Purpose:** Performance analysis of Python programs.

The `profile` module provides APIs for collecting and analyzing statistics about how Python source consumes processor resources.

> **Note**
>
> This output reports in this section have been reformatted to fit on the page. Lines ending with backslash (\) are continued on the next line.

## Running the Profiler

The most basic starting point in the `profile` module is `run()`. It takes a string statement as argument, and creates a report of the time spent executing different lines of code while running the statement.

```
# profile_fibonacci_raw.py

import profile


def fib(n):
    # from literateprograms.org
    # http://bit.ly/hlOQ5m
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


def fib_seq(n):
    seq = []
    if n > 0:
        seq.extend(fib_seq(n - 1))
    seq.append(fib(n))
    return seq


profile.run('print(fib_seq(20)); print()')
```

This recursive version of a Fibonacci sequence calculator is especially useful for demonstrating the profile because the performance can be improved significantly. The standard report format shows a summary and then details for each function executed.

```
$ python3 profile_fibonacci_raw.py

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 98\
7, 1597, 2584, 4181, 6765]

         57359 function calls (69 primitive calls) in 0.127 seco\
nds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(fu\
nction)
       21    0.000    0.000    0.000    0.000 :0(append)
        1    0.000    0.000    0.127    0.127 :0(exec)
       20    0.000    0.000    0.000    0.000 :0(extend)
```

```
        2    0.000    0.000    0.000    0.000 :0(print)
        1    0.001    0.001    0.001    0.001 :0(setprofile)
        1    0.000    0.000    0.127    0.127 <string>:1(<module\
>)
        1    0.000    0.000    0.127    0.127 profile:0(print(fi\
b_seq(20)); print())
        0    0.000             0.000          profile:0(profiler\
)
 57291/21    0.126    0.000    0.126    0.006 profile_fibonacci_\
raw.py:11(fib)
     21/1    0.000    0.000    0.127    0.127 profile_fibonacci_\
raw.py:22(fib_seq)
```

The raw version takes 57359 separate function calls and 0.127 seconds to run. The fact that there are only 69 *primitive* calls says that the vast majority of those 57k calls were recursive. The details about where time was spent are broken out by function in the listing showing the number of calls, total time spent in the function, time per call (tottime/ncalls), cumulative time spent in a function, and the ratio of cumulative time to primitive calls.

Not surprisingly, most of the time here is spent calling `fib()` repeatedly. Adding a cache decorator reduces the number of recursive calls, and has a big impact on the performance of this function.

```python
# profile_fibonacci_memoized.py

import functools
import profile


@functools.lru_cache(maxsize=None)
def fib(n):
    # from literateprograms.org
    # http://bit.ly/hlOQ5m
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


def fib_seq(n):
    seq = []
    if n > 0:
        seq.extend(fib_seq(n - 1))
    seq.append(fib(n))
    return seq


if __name__ == '__main__':
    profile.run('print(fib_seq(20)); print()')
```

By remembering the Fibonacci value at each level, most of the recursion is avoided and the run drops down to 89 calls that only take 0.001 seconds. The ncalls count for `fib()` shows that it *never* recurses.

```
$ python3 profile_fibonacci_memoized.py

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 98\
7, 1597, 2584, 4181, 6765]

         89 function calls (69 primitive calls) in 0.001 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(fu\
nction)
       21    0.000    0.000    0.000    0.000 :0(append)
        1    0.000    0.000    0.000    0.000 :0(exec)
       20    0.000    0.000    0.000    0.000 :0(extend)
        2    0.000    0.000    0.000    0.000 :0(print)
        1    0.001    0.001    0.001    0.001 :0(setprofile)
        1    0.000    0.000    0.000    0.000 <string>:1(<module\
>)
        1    0.000    0.000    0.001    0.001 profile:0(print(fi\
b_seq(20)); print())
        0    0.000             0.000          profile:0(profiler\
)
       21    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:12(fib)
     21/1    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:24(fib_seq)
```

## Running in a Context

Sometimes, instead of constructing a complex expression for `run()`, it is easier to build a simple expression and pass it parameters through a context, using `runctx()`.

```python
# profile_runctx.py

import profile
from profile_fibonacci_memoized import fib, fib_seq

if __name__ == '__main__':
    profile.runctx(
        'print(fib_seq(n)); print()',
        globals(),
        {'n': 20},
    )
```

In this example, the value of n is passed through the local variable context instead of being embedded directly in the statement passed to `runctx()`.

```
$ python3 profile_runctx.py

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
987, 1597, 2584, 4181, 6765]

        148 function calls (90 primitive calls) in 0.002 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(\
function)
       21    0.000    0.000    0.000    0.000 :0(append)
        1    0.000    0.000    0.001    0.001 :0(exec)
       20    0.000    0.000    0.000    0.000 :0(extend)
        2    0.000    0.000    0.000    0.000 :0(print)
        1    0.001    0.001    0.001    0.001 :0(setprofile)
        1    0.000    0.000    0.001    0.001 <string>:1(<module\
>)
        1    0.000    0.000    0.002    0.002 profile:0(print(fi\
b_seq(n)); print())
        0    0.000             0.000          profile:0(profiler\
)
    59/21    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:19(__call__)
       21    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:27(fib)
     21/1    0.000    0.000    0.001    0.001 profile_fibonacci_\
memoized.py:39(fib_seq)
```

## pstats: Saving and Working With Statistics

The standard report created by the `profile` functions is not very flexible. However, custom reports can be produced by saving the raw profiling data from `run()` and `runctx()` and processing it separately with the `pstats.Stats` class.

This example runs several iterations of the same test and combines the results:

```python
# profile_stats.py
```

```
import cProfile as profile
import pstats
from profile_fibonacci_memoized import fib, fib_seq

# Create 5 set of stats
for i in range(5):
    filename = 'profile_stats_{}.stats'.format(i)
    profile.run('print({}, fib_seq(20))'.format(i), filename)

# Read all 5 stats files into a single object
stats = pstats.Stats('profile_stats_0.stats')
for i in range(1, 5):
    stats.add('profile_stats_{}.stats'.format(i))

# Clean up filenames for the report
stats.strip_dirs()

# Sort the statistics by the cumulative time spent
# in the function
stats.sort_stats('cumulative')

stats.print_stats()
```

The output report is sorted in descending order of cumulative time spent in the function and the directory names are removed from the printed filenames to conserve horizontal space on the page.

```
$ python3 profile_stats.py

0 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \
987, 1597, 2584, 4181, 6765]
1 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \
987, 1597, 2584, 4181, 6765]
2 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \
987, 1597, 2584, 4181, 6765]
3 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \
987, 1597, 2584, 4181, 6765]
4 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \
987, 1597, 2584, 4181, 6765]
Sat Dec 31 07:46:22 2016    profile_stats_0.stats
Sat Dec 31 07:46:22 2016    profile_stats_1.stats
Sat Dec 31 07:46:22 2016    profile_stats_2.stats
Sat Dec 31 07:46:22 2016    profile_stats_3.stats
Sat Dec 31 07:46:22 2016    profile_stats_4.stats

         351 function calls (251 primitive calls) in 0.000 secon\
ds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(fu\
nction)
        5    0.000    0.000    0.000    0.000 {built-in method b\
uiltins.exec}
        5    0.000    0.000    0.000    0.000 <string>:1(<module\
>)
    105/5    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:24(fib_seq)
        5    0.000    0.000    0.000    0.000 {built-in method b\
uiltins.print}
      100    0.000    0.000    0.000    0.000 {method 'extend' o\
f 'list' objects}
       21    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:12(fib)
      105    0.000    0.000    0.000    0.000 {method 'append' o\
f 'list' objects}
        5    0.000    0.000    0.000    0.000 {method 'disable' \
of '_lsprof.Profiler' objects}
```

## Limiting Report Contents

The output can be restricted by function. This version only shows information about the performance of fib() and fib_seq()

The output can be restricted by function. This version only shows information about the performance of `fib()` and `fib_seq()` by using a regular expression to match the desired `filename:lineno(function)` values.

```
# profile_stats_restricted.py

import profile
import pstats
from profile_fibonacci_memoized import fib, fib_seq

# Read all 5 stats files into a single object
stats = pstats.Stats('profile_stats_0.stats')
for i in range(1, 5):
    stats.add('profile_stats_{}.stats'.format(i))
stats.strip_dirs()
stats.sort_stats('cumulative')

# limit output to lines with "(fib" in them
stats.print_stats('\(fib')
```

The regular expression includes a literal left parenthesis (`(`) to match against the function name portion of the location value.

```
$ python3 profile_stats_restricted.py

Sat Dec 31 07:46:22 2016    profile_stats_0.stats
Sat Dec 31 07:46:22 2016    profile_stats_1.stats
Sat Dec 31 07:46:22 2016    profile_stats_2.stats
Sat Dec 31 07:46:22 2016    profile_stats_3.stats
Sat Dec 31 07:46:22 2016    profile_stats_4.stats

         351 function calls (251 primitive calls) in 0.000 secon\
ds

   Ordered by: cumulative time
   List reduced from 8 to 2 due to restriction <'\\(fib'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(fu\
nction)
    105/5    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:24(fib_seq)
       21    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:12(fib)
```

## Caller / Callee Graphs

`Stats` also includes methods for printing the callers and callees of functions.

```
# profile_stats_callers.py

import cProfile as profile
import pstats
from profile_fibonacci_memoized import fib, fib_seq

# Read all 5 stats files into a single object
stats = pstats.Stats('profile_stats_0.stats')
for i in range(1, 5):
    stats.add('profile_stats_{}.stats'.format(i))
stats.strip_dirs()
stats.sort_stats('cumulative')

print('INCOMING CALLERS:')
stats.print_callers('\(fib')

print('OUTGOING CALLEES:')
stats.print_callees('\(fib')
```

The arguments to `print_callers()` and `print_callees()` work the same as the restriction arguments to `print_stats()`. The output shows the caller, callee, number of calls, and cumulative time.

```
$ python3 profile_stats_callers.py
$ python3 profile_stats_callers.py

INCOMING CALLERS:
   Ordered by: cumulative time
   List reduced from 8 to 2 due to restriction <'\\(fib'>

Function                                   was called by...
                                               ncalls  tottime  \
cumtime
profile_fibonacci_memoized.py:24(fib_seq)  <-       5    0.000  \
  0.000  <string>:1(<module>)
                                                100/5    0.000  \
  0.000  profile_fibonacci_memoized.py:24(fib_seq)
profile_fibonacci_memoized.py:12(fib)      <-      21    0.000  \
  0.000  profile_fibonacci_memoized.py:24(fib_seq)


OUTGOING CALLEES:
   Ordered by: cumulative time
   List reduced from 8 to 2 due to restriction <'\\(fib'>

Function                                   called...
                                               ncalls  tottime  \
cumtime
profile_fibonacci_memoized.py:24(fib_seq)  ->      21    0.000  \
  0.000  profile_fibonacci_memoized.py:12(fib)
                                                100/5    0.000  \
  0.000  profile_fibonacci_memoized.py:24(fib_seq)
                                                  105    0.000  \
  0.000  {method 'append' of 'list' objects}
                                                  100    0.000  \
  0.000  {method 'extend' of 'list' objects}
profile_fibonacci_memoized.py:12(fib)      ->
```

### See also

- Standard library documentation for profile
- functools.lru_cache() – The cache decorator used to improve performance in this example.
- The Stats Class – Standard library documentation for `pstats.Stats`.
- Gprof2Dot – Visualization tool for profile output data.
- Python Decorators: Syntactic Sugar | avinash.vora – Another memoized Fibonacci sequence generator in Python.
- Smiley – Python Application Tracer

← pdb — Interactive Debugger

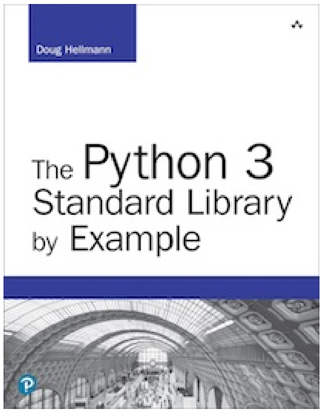
timeit — Time the execution of small bits of Python code. →

**Quick Links**

*This page was last updated 2017-01-29.*

**Navigation**

pdb — Interactive Debugger
timeit — Time the execution of small bits of Python code.



*The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.*

*Looking for examples for Python 2?*

**This Site**

Module Index
Index

**Other Writing**

Blog
The Python Standard Library By Example