Profiling Your Code with cProfile

Profiling code with cProfile is really quite easy. All you need to do is import the module and call its **run** function. Let's look at a simple example:

```
import hashlib
import cProfile
cProfile.run("hashlib.md5(b'abcdefghijkl').digest()")
#     4 function calls in 0.000 CPU seconds

#   Ordered by: standard name

#   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
#        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
#        1    0.000    0.000    0.000    0.000 {_hashlib.openssl_md5}
#        1    0.000    0.000    0.000    0.000 {method 'digest' of '_hashlib.HASH' objects}
#        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' object
```

Here we import the **hashlib** module and use cProfile to profile the creation of an MD5 hash. The first line shows that there were 4 function calls. The next line tells us how the results are ordered. According to the documentation, standard name refers to the far right column. There are a number of columns here.

- **ncalls** is the number of calls made.
- **tottime** is a total of the time spent in the given function.
- **percall** refers to the quotient of tottime divided by ncalls
- **cumtime** is the cumulative time spent in this and all subfunctions. It's even accurate for recursive functions!

- The second **percall** column is the quotient of cumtime divided by primitive calls
- **filename:lineno(function)** provides the respective data of each function

|

A primitive call is one that was not induced via recursion.

This isn't a very interesting example as there are no obvious bottlenecks. Let's create a piece of code with some built in bottlenecks and see if the profiler detects them.

```
import time

def fast():
    """"""
    print("I run fast!")

def slow():
    """"""
    time.sleep(3)
    print("I run slow!")

def medium():
    """"""
    time.sleep(0.5)
    print("I run a little slowly...")

def main():
    """"""
    fast()
    slow()
    medium()

if __name__ == '__main__':
    main()
```

In this example, we create four functions. The first three run at different rates. The **fast** function will run at normal speed; the **medium** function will take approximately half a second to run and the **slow** function will take around 3 seconds to run. The **main** function calls the other three. Now let's run cProfile against this silly little program:

```
import cProfile
import ptest
cProfile.run('main()')

#I run fast!
#I run slow!
#I run a little slowly...
#         8 function calls in 3.500 seconds

#   Ordered by: standard name

#   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
#        1    0.000    0.000    3.500    3.500 <string>:1(<module>)
#        1    0.000    0.000    0.500    0.500 ptest.py:15(medium)
#        1    0.000    0.000    3.500    3.500 ptest.py:21(main)
#        1    0.000    0.000    0.000    0.000 ptest.py:4(fast)
#        1    0.000    0.000    3.000    3.000 ptest.py:9(slow)
#        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' object
#        2    3.499    1.750    3.499    1.750 {time.sleep}
```

This time around we see the program took 3.5 seconds to run. If you examine the results, you will see that cProfile has identified the **slow** function as taking 3 seconds to run. That's the biggest bottleneck after the **main** function. Normally when you find a bottleneck like this, you would try to find a faster way to execute your code or perhaps decide that the runtime was acceptable. In this example, we know that the best way to speed up the function is to remove the **time.sleep** call or at least reduce the sleep length.

You can also call cProfile on the command line rather than using it in the interpreter. Here's one way to do it:

```
python -m cProfile ptest.py
```

This will run cProfile against your script in much the same way as we did before. But what if you want to save the profiler's output? Well, that's easy with cProfile! All you need to do is pass it the **-o** command followed by the name (or path) of the output file. Here's an example:

```
python -m cProfile -o output.txt ptest.py
```

Unfortunately, the file it outputs isn't exactly human-readable. If you want to read the file, then you'll need to use Python's **pstats** module. You can use pstats to format the output in various ways. Here's some code that shows how to get some output that's similar to what we've seen so far:

```
import pstats
p = pstats.Stats("output.txt")
p.strip_dirs().sort_stats(-1).print_stats()
#Thu Mar 20 18:32:16 2014    output.txt

#         8 function calls in 3.501 seconds

#   Ordered by: standard name

#   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
#        1    0.000    0.000    3.501    3.501 ptest.py:1(<module>)
#        1    0.001    0.001    0.500    0.500 ptest.py:15(medium)
#        1    0.000    0.000    3.501    3.501 ptest.py:21(main)
#        1    0.001    0.001    0.001    0.001 ptest.py:4(fast)
#        1    0.001    0.001    3.000    3.000 ptest.py:9(slow)
#        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' object
#        2    3.499    1.750    3.499    1.750 {time.sleep}

#<pstats.Stats instance at 0x017C9030>
```

The **strip_dirs** call will strip out all the paths to the modules from the output while the **sort_stats** call does the sorting that we're used to seeing. There are a bunch of really interesting examples in the cProfile documentation showing different ways to extract information using the pstats module.

Wrapping Up

At this point you should be able to use the **cProfile** module to help you diagnose why your code is so slow. You might also want to take a look at Python's **timeit** module. It allows you to time small pieces of your code if you don't want to deal with the complexities involved with profiling. There are also several other 3rd party modules that are good for profiling such as the line_profiler and the memory_profiler projects.