memory_profiler

Another great 3rd party profiling package is **memory_profiler**. The memory_profiler module can be used for monitoring memory consumption in a process or you can use it for a line-by-line analysis of the memory consumption of your code. Since it’s not included with Python, we’ll have to install it. You can use pip for this:

```
pip install memory_profiler
```

Once it’s installed, we need some code to run it against. The memory_profiler actually works in much the same way as line_profiler in that when you run it, memory_profiler will inject an instance of itself into __builtins__ named profile that you are supposed to use as a decorator on the function you are profiling. Here’s a simple example:

```
# memo_prof.py
#@profile
def mem_func():
    lots_of_numbers = list(range(1500))
    x = ['letters'] * (5 ** 10)
    del lots_of_numbers
    return None

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

In this example, we create a list that contains 1500 integers. Then we create a list with 9765625 (5 to the 10 power) instances of a string. Finally we delete the first list and return. The memory_profiler doesn’t have another script you need to run to do the actual profiling like line_profiler did. Instead you can just run Python and use its **-m** parameter on the command line to load the module and run it against our script:

```
python -m memory_profiler memo_prof.py
```

```
Filename: memo_prof.py

Line #    Mem usage    Increment   Line Contents
================================================
     1   16.672 MiB    0.000 MiB   @profile
     2                             def mem_func():
     3   16.707 MiB    0.035 MiB       lots_of_numbers = list(range(1500))
     4   91.215 MiB   74.508 MiB       x = ['letters'] * (5 ** 10)
     5   91.215 MiB    0.000 MiB       del lots_of_numbers
     6   91.215 MiB    0.000 MiB       return None
```

The columns are pretty self-explanatory this time around. We have our line numbers and then the amount of memory used after said line was executed. Next we have an increment field which tells us the difference in memory of the current line versus the line previous. The very last column is for the code itself.

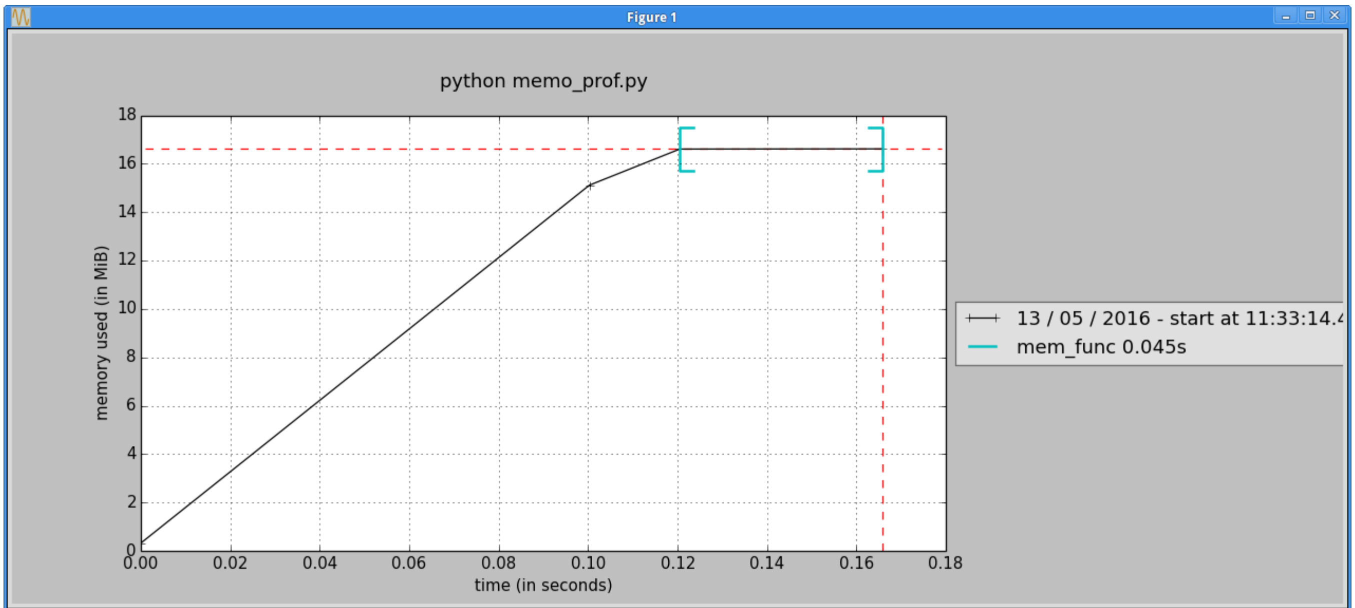
The memory_profiler also includes **mprof** which can be used to create full memory usage reports over time instead of line-by-line. It's very easy to use; just take a look:

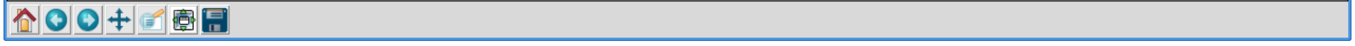
```
$ mprof run memo_prof.py
mprof: Sampling memory every 0.1s
running as a Python program...
```

mprof can also create a graph that shows you how your application consumed memory over time. To get the graph, all you need to do is:

```
$ mprof plot
```

For the silly example we created earlier, I got the following graph:





You should try running it yourself against a much more complex example to see a more interesting plot.