line_profiler

There's a neat 3rd party project called **line_profiler** that is designed to profile the time each individual line takes to execute. It also includes a script called **kernprof** for profiling Python applications and scripts using line_profiler. Just use pip to install the package. Here's how:

```
pip install line_profiler
```

To actually use the line_profiler, we will need some code to profile. But first, I need to explain how line_profiler works when you call it on the command line. You will actually be calling line_profiler by calling the kernprof script. I thought that was a bit confusing the first time I used it, but that's just the way it works. Here's the normal way to use it:

```
kernprof -l silly_functions.py
```

This will print out the following message when it finishes: *Wrote profile results to silly_functions.py.lprof*. This is a binary file that we can't view directly. When we run kernprof though, it will actually inject an instance of **LineProfiler** into your script's **__builtins__** namespace. The instance will be named **profile** and is meant to be used as a decorator. With that in mind, we can actually write our script:

```
# silly_functions.py
import time

#@profile
def fast_function():
    print("I'm a fast function!")

#@profile
def slow_function():
    time.sleep(2)
    print("I'm a slow function")
```

```
if __name__ == '__main__':
    fast_function()
    slow_function()
```

So now we have two decorated functions that are decorated with something that isn't imported. If you actually try to run this script as is, you will get a **NameError** because "profile" is not defined. So always remember to remove your decorators after you have profiled your code!

Let's back up and learn how to actually view the results of our profiler. There are two methods we can use. The first is to use the line_profiler module to read our results file:

```
python -m line_profiler silly_functions.py.lprof
```

The alternate method is to just use kernprof in verbose mode by passing is **-v**:

```
kernprof -l -v silly_functions.py
```

Regardless which method you use, you should end up seeing something like the following get printed to your screen:

```
I'm a fast function!
I'm a slow function
Wrote profile results to silly_functions.py.lprof
Timer unit: 1e-06 s

Total time: 3.4e-05 s
File: silly_functions.py
Function: fast_function at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def fast_function():
     5         1           34     34.0    100.0      print("I'm a fast function!")

Total time: 2.001 s
File: silly_functions.py
Function: slow_function at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def slow_function():
```

```
     8                                           def slow_function():
     9         1      2000942 2000942.0    100.0      time.sleep(2)
    10         1           59     59.0      0.0      print("I'm a slow function")
```

You will notice that the source code is printed out with the timing information for each line. There are six columns of information here. Let’s find out what each one means.

> - **Line #** - The line number of the code that was profiled
> - **Hits** - The number of times that particular line was executed
> - **Time** - The total amount of time the line took to execute (in the timer’s unit). The timer unit can be seen at the beginning of the output
> - **Per Hit** - The average amount of time that line of code took to execute (in timer units)
> - **% Time** - The percentage of time spent on the line relative to the total amount of time spent in said function
> - **Line Contents** - The actual source code that was executed

If you happen to be an IPython user, then you might want to know that IPython has a magic command (%lprun) that allows you to specify functions to profile and even which statement to execute.