```
In[6]: %%time
       total = 0
       for i in range(1000):
           for j in range(1000):
               total += i * (-1) ** j

CPU times: user 504 ms, sys: 979 µs, total: 505 ms
Wall time: 505 ms
```

For more information on `%time` and `%timeit`, as well as their available options, use the IPython help functionality (i.e., type **`%time?`** at the IPython prompt).

Profiling Full Scripts: %prun

A program is made of many single statements, and sometimes timing these statements in context is more important than timing them on their own. Python contains a built-in code profiler (which you can read about in the Python documentation), but IPython offers a much more convenient way to use this profiler, in the form of the magic function `%prun`.

By way of example, we'll define a simple function that does some calculations:

```
In[7]: def sum_of_lists(N):
           total = 0
           for i in range(5):
               L = [j ^ (j >> i) for j in range(N)]
               total += sum(L)
           return total
```

Now we can call `%prun` with a function call to see the profiled results:

```
In[8]: %prun sum_of_lists(1000000)
```

In the notebook, the output is printed to the pager, and looks something like this:

```
14 function calls in 0.714 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        5    0.599    0.120    0.599    0.120 <ipython-input-19>:4(<listcomp>)
        5    0.064    0.013    0.064    0.013 {built-in method sum}
        1    0.036    0.036    0.699    0.699 <ipython-input-19>:1(sum_of_lists)
        1    0.014    0.014    0.714    0.714 <string>:1(<module>)
        1    0.000    0.000    0.714    0.714 {built-in method exec}
```

The result is a table that indicates, in order of total time on each function call, where the execution is spending the most time. In this case, the bulk of execution time is in the list comprehension inside `sum_of_lists`. From here, we could start thinking about what changes we might make to improve the performance in the algorithm.

For more information on `%prun`, as well as its available options, use the IPython help functionality (i.e., type **`%prun?`** at the IPython prompt).

Line-by-Line Profiling with %lprun

The function-by-function profiling of `%prun` is useful, but sometimes it's more convenient to have a line-by-line profile report. This is not built into Python or IPython, but there is a `line_profiler` package available for installation that can do this. Start by using Python's packaging tool, `pip`, to install the `line_profiler` package:

```
$ pip install line_profiler
```

Next, you can use IPython to load the `line_profiler` IPython extension, offered as part of this package:

```
In[9]: %load_ext line_profiler
```

Now the `%lprun` command will do a line-by-line profiling of any function—in this case, we need to tell it explicitly which functions we're interested in profiling:

```
In[10]: %lprun -f sum_of_lists sum_of_lists(5000)
```

As before, the notebook sends the result to the pager, but it looks something like this:

```
Timer unit: 1e-06 s

Total time: 0.009382 s
File: <ipython-input-19-fa2be176cc3e>
Function: sum_of_lists at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def sum_of_lists(N):
     2         1            2      2.0      0.0      total = 0
     3         6            8      1.3      0.1      for i in range(5):
     4         5         9001   1800.2     95.9          L = [j ^ (j >> i) ...
     5         5          371     74.2      4.0          total += sum(L)
     6         1            0      0.0      0.0      return total
```

The information at the top gives us the key to reading the results: the time is reported in microseconds and we can see where the program is spending the most time. At this point, we may be able to use this information to modify aspects of the script and make it perform better for our desired use case.

For more information on `%lprun`, as well as its available options, use the IPython help functionality (i.e., type **`%lprun?`** at the IPython prompt).