Examining runtime

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

datacamp

Why should we time our code?

- Allows us to pick the **optimal** coding approach
- Faster code == more efficient code!

How can we time our code?

- Calculate runtime with IPython magic command `%timeit`
- **Magic commands:** enhancements on top of normal Python syntax
  - Prefixed by the "%" character
  - Link to docs (**here**)
  - See all available magic commands with `%lsmagic`

Using %timeit

Code to be timed

```
import numpy as np

rand_nums = np.random.rand(1000)
```

4. Using %timeit
Consider this example: we want to inspect the runtime for selecting 1,000 random numbers between zero and one using NumPy's random-dot-rand function. Using %timeit just requires adding the magic command before the line of code we want to analyze. That's it! One simple command to gather runtimes. Magic indeed!

Timing with `%timeit`

```
%timeit rand_nums = np.random.rand(1000)
```

```
8.61 µs ± 69.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

%timeit output

Code to be timed

```
rand_nums = np.random.rand(1000)
```

Timing with `%timeit`

```
%timeit rand_nums = np.random.rand(1000)
```

```
8.61 µs ± 69.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

%timeit output

Code to be timed

```
rand_nums = np.random.rand(1000)
```

Timing with `%timeit`

```
%timeit rand_nums = np.random.rand(1000)
```

```
8.61 µs ± 69.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

7. %timeit output
We also see that multiple runs and loops were generated. % timeit runs through the provided code multiple times to estimate the code's execution time. This provides a more accurate representation of the actual runtime rather than relying on just one iteration to calculate the runtime. The mean and standard deviation displayed in the output is a summary of the runtime considering each of the multiple runs.

5. %timeit output
One advantage to using %timeit is the fact that it provides an average of timing statistics.

%timeit output

Code to be timed

```
rand_nums = np.random.rand(1000)
```

Timing with `%timeit`

```
%timeit rand_nums = np.random.rand(1000)
```

```
8.61 µs ± 69.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Specifying number of runs/loops

Setting the number of runs (`-r`) and/or loops (`-n`)

```
# Set number of runs to 2 (-r2)
# Set number of loops to 10 (-n10)

%timeit -r2 -n10 rand_nums = np.random.rand(1000)
```

```
16.9 µs ± 5.14 µs per loop (mean ± std. dev. of 2 runs, 10 loops each)
```

8. Specifying number of runs/loops
The number of runs represents how many iterations you'd like to use to estimate the runtime. The number of loops represents how many times you'd like the code to be executed per run. We can specify the number of runs, using the -r flag, and the number of loops, using the -n flag. Here, we use -r2, to set the number of runs to two and -n10, to set the number of loops to ten. In this example, %timeit would execute our random number selection 20 times in order to estimate runtime (2 runs each with 10 executions).

Using %timeit in line magic mode

Line magic (`%timeit`)

```
# Single line of code


%timeit nums = [x for x in range(10)]
```

```
914 ns ± 7.33 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

9. Using %timeit in line magic mode
Another cool feature of %timeit is its ability to run on single or multiple lines of code. When using %timeit in line magic mode, or with a single line of code, one percentage sign is used.

Using %timeit in cell magic mode

Cell magic (`%%timeit`)

```
# Multiple lines of code

%%timeit
nums = []
for x in range(10):
    nums.append(x)
```

Note: Whenever using magic functions, you must keep it as the first line in your code.

```
1.17 µs ± 3.26 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

10. Using %timeit in cell magic mode
Similarly, we can run %timeit in cell magic mode (or provide multiple lines of code) by using two percentage signs.

Saving output

Saving the output to a variable (`-o`)

```
times = %timeit -o rand_nums = np.random.rand(1000)
```

```
8.69 µs ± 91.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

11. Saving output
We can save the output of %timeit into a variable using the -o flag.

```
times.timings
```

```
[8.697893059998023e-06,
 8.651204760008113e-06,
 8.634270530001232e-06,
 8.66847825998775e-06,
 8.619398139999247e-06,
 8.902550710008654e-06,
 8.633500570012985e-06]
```

```
times.best
```

```
8.619398139999247e-06
```

```
times.worst
```

```
8.902550710008654e-06
```

Comparing times

Python data structures can be created using formal name

```
formal_list = list()
formal_dict = dict()
formal_tuple = tuple()
```

Python data structures can be created using literal syntax

```
literal_list = []
literal_dict = {}
literal_tuple = ()
```

```
f_time = %timeit -o formal_dict = dict()
```

```
145 ns ± 1.5 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

```
l_time = %timeit -o literal_dict = {}
```

```
93.3 ns ± 1.88 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

```
diff = (f_time.average - l_time.average) * (10**9)
print('l_time better than f_time by {} ns'.format(diff))
```

```
l_time better than f_time by 51.90819192857814 ns
```

14. Comparing times

If we wanted to compare the runtime between creating a dictionary using the formal name and creating a dictionary using the literal syntax, we could save the output of the individual %timeit commands as shown here. Then, we could compare the two outputs.

Comparing times

```
%timeit formal_dict = dict()
```

```
145 ns ± 1.5 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

```
%timeit literal_dict = {}
```

```
93.3 ns ± 1.88 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

15. Comparing times
Here, we can see that using the literal syntax to create a dictionary is faster than using the formal name without writing code to do the analysis for us.

Off to the races!

WRITING EFFICIENT PYTHON CODE

Code profiling for runtime

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

datacamp

Code profiling

- Detailed stats on frequency and duration of function calls
- Line-by-line analyses
- Package used: `line_profiler`

```
pip install line_profiler
```

2. Code profiling
Code profiling is a technique used to describe how long, and how often, various parts of a program are executed. The beauty of a code profiler is its ability to gather summary statistics on individual pieces of our code without using magic commands like %timeit. We'll focus on the line_profiler package to profile a function's runtime line-by-line. Since this package isn't a part of Python's Standard Library, we need to install it separately. This can easily be done with a pip install command. Let's explore using line_profiler with an example.

Code profiling: runtime

```
heroes = ['Batman', 'Superman', 'Wonder Woman']

hts = np.array([188.0, 191.0, 183.0])

wts = np.array([ 95.0, 101.0,  74.0])
```

4. Code profiling: runtime
We've also developed a function called convert_units that converts each hero's height from centimeters to inches and weight from kilograms to pounds.

```
def convert_units(heroes, heights, weights):

    new_hts = [ht * 0.39370  for ht in heights]
    new_wts = [wt * 2.20462  for wt in weights]

    hero_data = {}

    for i,hero in enumerate(heroes):
        hero_data[hero] = (new_hts[i], new_wts[i])

    return hero_data
```

```
convert_units(heroes, hts, wts)
```

```
{'Batman': (74.0156, 209.4389),
 'Superman': (75.1967, 222.6666),
 'Wonder Woman': (72.0471, 163.1419)}
```

Code profiling: runtime

```
%timeit convert_units(heroes, hts, wts)
```

```
3 µs ± 32 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

6. Code profiling: runtime
We would have to use %timeit on each individual line of our convert_units function. **But, that's a lot of manual work and not very efficient.**

```
%timeit new_hts = [ht * 0.39370  for ht in hts]
```

```
1.09 µs ± 11 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
%timeit new_wts = [wt * 2.20462  for wt in wts]
```

```
1.08 µs ± 6.42 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
%%timeit
hero_data = {}
for i,hero in enumerate(heroes):
    hero_data[hero] = (new_hts[i], new_wts[i])
```

```
634 ns ± 9.29 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

6. Code profiling: runtime

We would have to use %timeit on each individual line of our convert_units function.

But, that's a lot of manual work and not very efficient.

Code profiling: line_profiler

Using `line_profiler` package

```
%load_ext line_profiler
```

Magic command for line-by-line times

```
%lprun -f
```

7. Code profiling: line_profiler
Instead, we can profile our function with the line_profiler package. To use this package, we first need to load it into our session. We can do this using the command %load_ext followed by line_profiler. Now, we can use the magic command %lprun, from line_profiler, to gather runtimes for individual lines of code within the convert_units function. %lprun uses a special syntax. First, we use the -f flag to indicate we'd like to profile a function.

Code profiling: line_profiler

Using `line_profiler` package

```
%load_ext line_profiler
```

Magic command for line-by-line times

```
%lprun -f convert_units
```

8. Code profiling: line_profiler
Next, we specify the name of the function we'd like to profile. Note, the name of the function is passed without any parentheses.

Code profiling: line_profiler

Using `line_profiler` package

```
%load_ext line_profiler
```

Magic command for line-by-line times

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

8. Code profiling: line_profiler
Next, we specify the name of the function we'd like to profile. Note, the name of the function is passed without any parentheses.

9. Code profiling: line_profiler
Finally, we provide the exact function call we'd like to profile by including any arguments that are needed.

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

10. %lprun output

The output from %lprun provides a nice table that summarizes the profiling statistics.

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

11. %lprun output
First, a column specifying the line number followed by a column displaying the number of times that line was executed (called the Hits column).

12. %lprun output
Next, the Time column shows the total amount of time each line took to execute.

13. %lprun output
This column uses a specific timer unit that can be found in the first line of the output. Here, the timer unit is listed in microseconds using scientific notation.

14. %lprun output
We see that line three took 13 timer units, or, roughly 13 microseconds to run.

17. %lprun output
The % Time column shows the percentage of time spent on a line relative to the total amount of time spent in the function. This can be a nice way to see which lines of code are taking up the most time within a function.

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

11. %lprun output
First, a column specifying the line number followed by a column displaying the number of times that line was executed (called the Hits column).

12. %lprun output
Next, the Time column shows the total amount of time each line took to execute.

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

19. %lprun output caveats

You may notice that the Total time reported when using %lprun and the time reported from using %timeit do not match. Remember, %timeit uses multiple loops in order to calculate an average and standard deviation of time, so the time reported from each of these magic commands aren't expected to match exactly.

```
%timeit convert_units convert_units(heroes, hts, wts)
```

```
3 µs ± 32 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

Let's practice your new profiling powers!

WRITING EFFICIENT PYTHON CODE

Code profiling for memory usage

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

datacamp

Quick and dirty approach

```
import sys
```

```
nums_list = [*range(1000)]
sys.getsizeof(nums_list)
```

```
9112
```

```
import numpy as np

nums_np = np.array(range(1000))
sys.getsizeof(nums_np)
```

```
8096
```

Code profiling: memory

- Detailed stats on memory consumption
- Line-by-line analyses
- Package used: `memory_profiler`

```
pip install memory_profiler
```

- Using `memory_profiler` package

```
%load_ext memory_profiler


%mprun -f convert_units convert_units(heroes, hts, wts)
```

3. Code profiling: memory
Just like we've used code profiling to gather detailed stats on runtimes, we can also use code profiling to analyze the memory allocation for each line of code in our code base. We'll use the memory_profiler package that is very similar to the line_profiler package. It can be downloaded via pip and comes with a handy magic command (%mprun) that uses the same syntax as %lprun.

4. Code profiling: memory
One drawback to using %mprun is that any function profiled for memory consumption must be defined in a file and imported. **%mprun can only be used on functions defined in physical files, and not in the IPython session.** In this example, the convert_units function was placed in a file named hero_funcs-dot-py. Now, we simply import our function from the hero_funcs file and use the magic command %mprun to gather statistics on its memory footprint.

Code profiling: memory

- Functions must be imported when using `memory_profiler`
  - `hero_funcs.py`

```
from hero_funcs import convert_units
```

```
%load_ext memory_profiler


%mprun -f convert_units convert_units(heroes, hts, wts)
```

5. %mprun output
%mprun output is similar to %lprun output. Here, we see a line-by-line description of the memory consumption for the function in question.

10. %mprun output
Profiling a function with %mprun allows us to see what lines are taking up a large amount of memory and possibly develop a more efficient solution.

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

6. %mprun output
The first column represents the line number of the code that has been profiled.
7. %mprun output
The second column (Mem usage) is the memory used after that line has been executed.

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

8. %mprun output
Next, the Increment column shows the difference in memory of the current line with respect to the previous one. This shows us the impact of the current line on the total memory usage.

9. %mprun output
The last column (Line Contents) shows the source code that has been profiled.

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

10. %mprun output
Profiling a function with %mprun allows us to see what lines are taking up a large amount of memory and possibly develop a more efficient solution. Before moving on, I want to draw your attention to a few things.

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

11. %mprun output caveats
First, the memory is reported in mebibytes. Although one mebibyte is not exactly the same as one megabyte, for our purposes, we can assume they are close enough to mean the same thing.

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

12. %mprun output caveats

Second, the heroes list, hts array, and wts array used to generate this output are not the original datasets we've used in the past. Instead, a random sample of 35,000 heroes was used to create these datasets in order to clearly show the power of using the memory_profiler.

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

13. %mprun output caveats
If we had used the original datasets, the memory used would have been too small to report. Don't worry too much about this - the main take away is that small memory allocation may not show up when using %mprun and that is a perfectly fine result. The random sample of 35,000 heroes was used solely to show typical %mprun output when the memory was large enough to report.

%mprun output caveats

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

%mprun output caveats

Data used in this example is a random sample of 35,000 heroes.

(not original 480 superheroes dataset)

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

14. %mprun output caveats
Finally, the memory_profiler package inspects memory consumption by querying the operating system. This might be slightly different from the amount of memory that is actually used by the Python interpreter. Thus, results may differ between platforms and even between runs. Regardless, the important information can still be observed.

%mprun output caveats

Small memory allocations could result in `0.0 MiB` output.

(using original 480 superheroes dataset)

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1     98.7 MiB     98.7 MiB   def convert_units(heroes, heights, weights):
     2
     3     98.7 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4     98.7 MiB      0.0 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6     98.7 MiB      0.0 MiB       hero_data = {}
     7
     8     98.7 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9     98.7 MiB      0.0 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11     98.7 MiB      0.0 MiB       return hero_data
```

%mprun output caveats

- Inspects memory by querying the operating system
- Results may differ between platforms and runs
  - Can still observe how each line of code compares to others based on memory consumption

Let's practice!

WRITING EFFICIENT PYTHON CODE