

Timing and Profiling in IPython

Timing and profiling code is all sorts of useful, and it's also just good ol' fashioned fun (and sometimes surprising!). In this post, I'll introduce how to do the following through IPython magic functions:

- `%time` & `%timeit`: See how long a script takes to run (one time, or averaged over a bunch of runs).
- `%prun`: See how long it took each function in a script to run.
- `%lprun`: See how long it took each line in a function to run.
- `%mprun` & `%memit`: See how much memory a script uses (line-by-line, or averaged over a bunch of runs).

Installation & Setup

Please make sure you're running IPython 0.11 or greater. This post was authored against Python 2.7 and IPython 0.13.1.

```
$ pip install ipython
$ ipython --version
0.13.1
```

Most of the functionality we'll work with is included in the standard library, but if you're interested in line-by-line or memory profiling, go ahead and run through this setup. First, install the following:

```
$ pip install line-profiler
$ pip install psutil
$ pip install memory_profiler
```

Next, create an IPython profile and extensions directory where we'll configure a couple of missing magic functions:

```
$ ipython profile create
[ProfileCreate] Generating default config file: u'/Users/tsclausing/.ipython/profile_default/i
python_config.py'

$ mkdir ~/.ipython/extensions/
```

Create the following IPython extension files with the contents below to define the magic functions:

~/ipython/extensions/line_profiler_ext.py

```
import line_profiler

def load_ipython_extension(ip):
    ip.define_magic('lprun', line_profiler.magic_lprun)
```

~/ipython/extensions/memory_profiler_ext.py

```
import memory_profiler

def load_ipython_extension(ip):
    ip.define_magic('memit', memory_profiler.magic_memit)
    ip.define_magic('mprun', memory_profiler.magic_mprun)
```

Finally, register the extension modules you just created with the default IPython profile we made earlier:

Edit ~/ipython/profile_default/ipython_config.py , search for, **uncomment**, and modify these lists to include:

```
c.TerminalIPythonApp.extensions = [
    'line_profiler_ext',
    'memory_profiler_ext',
]
c.InteractiveShellApp.extensions = [
    'line_profiler_ext',
    'memory_profiler_ext',
]
```

And that's it! We're ready to time and profile to our hearts content. Start `ipython` and test for the following:

```
$ ipython
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.1 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]: %time?

In [2]: %timeit?

In [3]: %prun?

In [4]: %lprun?

In [5]: %mprun?

In [6]: %memit?
```

Time Profiling

Time profiling does exactly what it sounds like - it tells you how much time it took to execute a script, which may be a simple one-liner or a whole module.

%time

See how long it takes a script to run.

```
In [7]: %time {1 for i in xrange(10*1000000)}
CPU times: user 0.72 s, sys: 0.16 s, total: 0.88 s
Wall time: 0.75 s
```

%timeit

See how long a script takes to run averaged over multiple runs.

```
In [8]: %timeit 10*1000000  
10000000 loops, best of 3: 38.2 ns per loop
```

`%timeit` will limit the number of runs depending on how long the script takes to execute. Keep in mind that the `timeit` module in the standard library *does not* do this by default, so timing long running scripts that way may leave you waiting forever.

The number of runs may be set with `-n 1000`, for example, which will limit `%timeit` to a thousand iterations, like this:

```
In [9]: %timeit -n 1000 10*1000000  
1000 loops, best of 3: 67 ns per loop
```

Also note that the run-time reported will vary more when limited to fewer loops.

%prun

See how long it took each function in a script to run.

```
In [10]: from time import sleep
```

```
In [11]: def foo(): sleep(1)
```

```
In [12]: def bar(): sleep(2)
```

```
In [13]: def baz(): foo(), bar()
```

```
In [14]: %prun baz()
```

```
7 function calls in 3.001 seconds
```

```
Ordered by: internal time
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2	3.001	1.500	3.001	1.500	{time.sleep}
1	0.000	0.000	3.001	3.001	<ipython-input-17-c32ce4852c7d>:1(baz)
1	0.000	0.000	2.000	2.000	<ipython-input-11-2689ca7390dc>:1(bar)
1	0.000	0.000	1.001	1.001	<ipython-input-10-e11af1cc2c91>:1(foo)
1	0.000	0.000	3.001	3.001	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

%lprun

See how long it took each line in a function to run.

Create and edit a new module named `foo.py` in the same directory where you started IPython. Paste the following code in the file and jump back to IPython.

```
def foo(n):
    phrase = 'repeat me'
    pmul = phrase * n
    pjoin = ''.join([phrase for x in xrange(n)])
    pinc = ''
    for x in xrange(n):
        pinc += phrase
    del pmul, pjoin, pinc
```

Import the function and profile it line by line with `%lprun`. Functions to profile this way must be passed by name with `-f`.

```
In [15]: from foo import foo
```

```
In [16]: %lprun -f foo foo(100000)
```

```
Timer unit: 1e-06 s
```

```
File: foo.py
```

```
Function: foo at line 1
```

```
Total time: 0.301032 s
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
=====					
1					def foo(n):
2	1	3	3.0	0.0	phrase = 'repeat me'
3	1	185	185.0	0.1	pmul = phrase * n
4	100001	97590	1.0	32.4	pjoi = ''.join([phrase for x in xrange(n)
5	1	4	4.0	0.0	pinc = ''
6	100001	90133	0.9	29.9	for x in xrange(n):
7	100000	112935	1.1	37.5	pinc += phrase
8	1	182	182.0	0.1	del pmul, pjoi, pinc

Memory Profiling

%mprun

See how much memory a script uses line by line. Let's take a look at the same `foo()` function that we profiled with `%lprun` - except this time we're interested in incremental memory usage and not execution time.

```
In [17]: %mprun -f foo foo(100000)
```

```
Filename: foo.py
```

Line #	Mem usage	Increment	Line Contents
1	20.590 MB	0.000 MB	def foo(n):
2	20.590 MB	0.000 MB	phrase = 'repeat me'
3	21.445 MB	0.855 MB	pmul = phrase * n
4	25.020 MB	3.574 MB	pjoi = ''.join([phrase for x in xrange(n)])
5	25.020 MB	0.000 MB	pinc = ''
6	43.594 MB	18.574 MB	for x in xrange(n):
7	43.594 MB	0.000 MB	pinc += phrase
8	41.102 MB	-2.492 MB	del pmul, pjoi, pinc

%memit

See how much memory a script uses overall. `%memit` works a lot like `%timeit` except that the number of iterations is set with `-r` instead of `-n`.

```
In [18]: %memit -r 3 [x for x in xrange(1000000)]
maximum of 3: 75.320312 MB per loop
```

What do you know?

Please leave other tips & tools in the comments below. I remember a while back seeing a video from someone who built a profiling visualization in matplotlib, but I haven't been able to dig it up. If you find it, please post it here, too!

Some additional reading and sources:

- Run `%magicfunctionname?` for each magic function for specific help.
- Profiling Python Code (<http://scikit-learn.org/dev/developers/performance.html#profiling-python-code>)
- Chapter 3 of the Python Data Analysis book (<http://www.amazon.com/Python-Data-Analysis-Wes-McKinney/dp/1449319793>)
- Python MotW: Debugging and Profiling (<http://pymotw.com/2/profilers.html>)

by Scot Clauson on 06 Mar 2013

Related Posts

- 03 Apr 2013 » Fun Extending Dict (/2013/04/03/fun-extending-dict.html)
- 25 Mar 2013 » How I got started with Python (/2013/03/25/how-i-got-start-with-python.html)
- 18 Mar 2013 » Examining PyPI Package Statistics with lxml and pandas (/2013/03/18/pypi-package-stats.html)

5 Comments

Pynash

 Login ▾

Sort by Best ▾

Share  Favorite ★

Join the discussion...

**Fabian Pedregosa** · a year ago

With the latest `memory_profiler` (0.24) you can load the IPython magic functions using `"%load_ext memory_profiler"`, no need to edit the IPython config file :-)

1 ^ | v · Reply · Share ›

**olgabot** · 2 months ago

Thank you! One tip is that I had to add the flag `"--pre"` to each of the `"pip install"` commands because they were all in beta/pre-release stage.

^ | v · Reply · Share ›

**Augustine Dunn** · 10 months ago

regarding:

"Notice that by default `%timeit` runs your code millions of times before returning. Timing long running scripts this way may leave you waiting forever."

It is my understanding that ``timeit`` loops intelligently. Your example takes no time so it loops millions. However if the code took longer it will lower the loops it runs automatically unless told to loop ``n`` times.

^ | v · Reply · Share ›

**T. Scot Clausing** → Augustine Dunn · 10 months ago

Thanks for the heads up! I've submitted a PR for this change:

<https://github.com/pynashorg/p...>

^ | v · Reply · Share ›

**T. Scot Clausing** · a year ago

Hey Fabian! Thanks for the heads up - I was not aware of `%load_ext`. IPython docs here:

<http://ipython.org/ipython-doc...>

And thank you for your awesome work on memory_profiler (and SciKit, SymPy, SciPy ...)! It's a long way, but if you're ever in Nashville, TN (or at PyCon) - holler at me and I'll buy you a beer.

^ | v • Reply • Share ›

ALSO ON PYNASH

WHAT'S THIS?

How I got started with Python

2 comments • a year ago



Ryan Macy — I would also add Idiomatic Python <http://www.jeffknupp.com/writi...> as another great reference for new Pythonistas.

repr and dir for success

1 comment • a year ago



T. Scot Clausen — If you find yourself using `dir()` frequently, make sure you check out `see()`: <https://github.com/inky/see>

Quick Hit: Virtualenvwrapper Auto Directory Tips

2 comments • a year ago



Patrick Altman — Great tip. This is something I have been using along these same lines <https://github.com/kennethreit...> so that if I ...

diff in 50 lines of Python

1 comment • a year ago



Éric Araujo — Instead of using insecure `os.system` and OS-specific `cat`, `inspect.getsource` can do the work for you :)

 [Subscribe](#)

 [Add Disqus to your site](#)