

[Huy Nguyen](#)

- [posts](#)
- [about](#)
-  [github](#)
-  [twitter](#)
-  [rss feed](#)

03 Sep 2013

A guide to analyzing Python performance

While it's not always the case that every Python program you write will require a rigorous performance analysis, it is reassuring to know that there are a wide variety of tools in Python's ecosystem that one can turn to when the time arises.

Analyzing a program's performance boils down to answering 4 basic questions:

1. How fast is it running?
2. Where are the speed bottlenecks?
3. How much memory is it using?
4. Where is memory leaking?

Below, we'll dive into the details of answering these questions using some awesome tools.

Coarse grain timing with time

Let's begin by using a quick and dirty method of timing our code: the good old unix utility `time`.

```
$ time python yourprogram.py

real    0m1.028s
user    0m0.001s
sys     0m0.003s
```

The meaning between the three output measurements are detailed in this [stackoverflow article](#), but in short

- `real` - refers to the actual elapsed time
- `user` - refers to the amount of cpu time spent outside of kernel
- `sys` - refers to the amount of cpu time spent inside kernel specific functions

You can get a sense of how many cpu cycles your program used up regardless of other programs running on the system by adding together the `sys` and `usertimes`.

If the sum of `sys` and `usertimes` is much less than `real` time, then you can guess that most your program's performance issues are most likely related to IO waits.

Fine grain timing with a timing context manager

Our next technique involves direct instrumentation of the code to get access to finer grain timing information. Here's a small snippet I've found invaluable for making ad-hoc timing measurements:

```
timer.py

import time

class Timer(object):
    def __init__(self, verbose=False):
        self.verbose = verbose

    def __enter__(self):
        self.start = time.time()
        return self

    def __exit__(self, *args):
        self.end = time.time()
        self.secs = self.end - self.start
        self.msecs = self.secs * 1000 # millisecs
        if self.verbose:
            print 'elapsed time: %f ms' % self.msecs
```

In order to use it, wrap blocks of code that you want to time with Python's `with` keyword and this `Timer` context manager. It will take care of starting the timer when your code block begins execution and stopping the timer when your code block ends.

Here's an example use of the snippet:

```
from timer import Timer
from redis import Redis
rdb = Redis()

with Timer() as t:
    rdb.lpush("foo", "bar")
    print "=> elapsed lpush: %s s" % t.secs

with Timer() as t:
    rdb.lpop("foo")
    print "=> elapsed lpop: %s s" % t.secs
```

I'll often log the outputs of these timers to a file in order to see how my program's performance evolves over time.

Line-by-line timing and execution frequency with a profiler

Robert Kern has a nice project called [line_profiler](#) which I often use to see how fast and how often each line of code is running in my scripts.

To use it, you'll need to install the python package via pip:

```
$ pip install line_profiler
```

Once installed you'll have access to a new module called "line_profiler" as well as an executable script "kernprof.py".

To use this tool, first modify your source code by decorating the function you want to measure with the `@profile` decorator. Don't worry, you don't have to import anything in order to use this decorator. The `kernprof.py` script automatically injects it into your script's runtime during execution.

primes.py

```
@profile
def primes(n):
    if n==2:
        return [2]
    elif n<2:
        return []
    s=range(3,n+1,2)
    mroot = n ** 0.5
    half=(n+1)/2-1
    i=0
    m=3
    while m <= mroot:
        if s[i]:
            j=(m*m-3)/2
            s[j]=0
            while j<half:
                s[j]=0
                j+=m
            i=i+1
            m=2*i+3
    return [2]+[x for x in s if x]
primes(100)
```

Once you've gotten your code setup with the `@profile` decorator, use `kernprof.py` to run your script.

```
$ kernprof.py -l -v fib.py
```

The `-l` option tells `kernprof` to inject the `@profile` decorator into your script's builtins, and `-v` tells `kernprof` to display timing information once your script finishes. Here's one the output should look like for the above script:

```
Wrote profile results to primes.py.lprof
Timer unit: 1e-06 s
```

```
File: primes.py
Function: primes at line 2
Total time: 0.00019 s
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
2					@profile
3					def primes(n):
4	1	2	2.0	1.1	if n==2:
5					return [2]
6	1	1	1.0	0.5	elif n<2:
7					return []
8	1	4	4.0	2.1	s=range(3,n+1,2)
9	1	10	10.0	5.3	mroot = n ** 0.5
10	1	2	2.0	1.1	half=(n+1)/2-1
11	1	1	1.0	0.5	i=0
12	1	1	1.0	0.5	m=3
13	5	7	1.4	3.7	while m <= mroot:
14	4	4	1.0	2.1	if s[i]:
15	3	4	1.3	2.1	j=(m*m-3)/2
16	3	4	1.3	2.1	s[j]=0
17	31	31	1.0	16.3	while j<half:
18	28	28	1.0	14.7	s[j]=0
19	28	29	1.0	15.3	j+=m
20	4	4	1.0	2.1	i=i+1
21	4	4	1.0	2.1	m=2*i+3
22	50	54	1.1	28.4	return [2]+[x for x in s if x]

Look for lines with a high amount of hits or a high time interval. These are the areas where optimizations can yield the greatest improvements.

How much memory does it use?

Now that we have a good grasp on timing our code, let's move on to figuring out how much memory our programs are using. Fortunately for us, Fabian Pedregosa has implemented a nice [memory profiler](#) modeled after Robert Kern's `line_profiler`.

First install it via pip:

```
$ pip install -U memory_profiler
$ pip install psutil
```

(Installing the `psutil` package here is recommended because it greatly improves the performance of the `memory_profiler`).

Like `line_profiler`, `memory_profiler` requires that you decorate your function of interest with an `@profile` decorator like so:

```
@profile
def primes(n):
    ...
    ...
```

To see how much memory your function uses run the following:

```
$ python -m memory_profiler primes.py
```

You should see output that looks like this once your program exits:

Filename: primes.py

Line #	Mem usage	Increment	Line Contents
2			@profile
3	7.9219 MB	0.0000 MB	def primes(n):
4	7.9219 MB	0.0000 MB	if n==2:
5			return [2]
6	7.9219 MB	0.0000 MB	elif n<2:
7			return []
8	7.9219 MB	0.0000 MB	s=range(3,n+1,2)
9	7.9258 MB	0.0039 MB	mroot = n ** 0.5
10	7.9258 MB	0.0000 MB	half=(n+1)/2-1
11	7.9258 MB	0.0000 MB	i=0
12	7.9258 MB	0.0000 MB	m=3
13	7.9297 MB	0.0039 MB	while m <= mroot:
14	7.9297 MB	0.0000 MB	if s[i]:
15	7.9297 MB	0.0000 MB	j=(m*m-3)/2
16	7.9258 MB	-0.0039 MB	s[j]=0
17	7.9297 MB	0.0039 MB	while j<half:
18	7.9297 MB	0.0000 MB	s[j]=0
19	7.9297 MB	0.0000 MB	j+=m
20	7.9297 MB	0.0000 MB	i=i+1
21	7.9297 MB	0.0000 MB	m=2*i+3
22	7.9297 MB	0.0000 MB	return [2]+[x for x in s if x]

IPython shortcuts for line_profiler and memory_profiler

A little known feature of `line_profiler` and `memory_profiler` is that both programs have shortcut commands accessible from within IPython. All you have to do is type the following within an IPython session:

```
%load_ext memory_profiler
%load_ext line_profiler
```

Upon doing so you'll have access to the magic commands `%lprun` and `%mprun` which behave similarly to their command-line counterparts. The major difference here is that you won't need to decorate your to-be-profiled functions with the `@profile` decorator. Just go ahead and run the profiling directly within your IPython session like so:

```
In [1]: from primes import primes
In [2]: %mprun -f primes primes(1000)
In [3]: %lprun -f primes primes(1000)
```

This can save you a lot of time and effort since none of your source code needs to be modified in order to use these profiling commands.

Where's the memory leak?

The cPython interpreter uses reference counting as it's main method of keeping track of memory. This means that every object contains a counter, which is incremented when a reference to the object is stored somewhere, and decremented when a reference to it is deleted. When the counter reaches zero, the cPython interpreter knows that the object is no longer in use so it deletes the object and deallocates the occupied memory.

A memory leak can often occur in your program if references to objects are held even though the object is no longer in use.

The quickest way to find these "memory leaks" is to use an awesome tool called [objgraph](#) written by Marius Gedminas. This tool allows you to see the number of objects in memory and also locate all the different places in your code that hold references to these objects.

To get started, first install `objgraph`:

```
pip install objgraph
```

Once you have this tool installed, insert into your code a statement to invoke the debugger:

```
import pdb; pdb.set_trace()
```

Which objects are the most common?

At run time, you can inspect the top 20 most prevalent objects in your program by running:

```
(pdb) import objgraph
(pdb) objgraph.show_most_common_types()

MyBigFatObject      20000
tuple                16938
function             4310
dict                 2790
wrapper_descriptor   1181
builtin_function_or_method 934
weakref              764
list                 634
method_descriptor    507
getset_descriptor    451
type                 439
```

Which objects have been added or deleted?

We can also see which objects have been added or deleted between two points in time:

```
(pdb) import objgraph
(pdb) objgraph.show_growth()
.
.
.
(pdb) objgraph.show_growth() # this only shows objects that has been added or deleted since last show_growth() call

traceback           4      +2
KeyboardInterrupt    1      +1
frame                24      +1
list                 667     +1
tuple                16969   +1
```

What is referencing this leaky object?

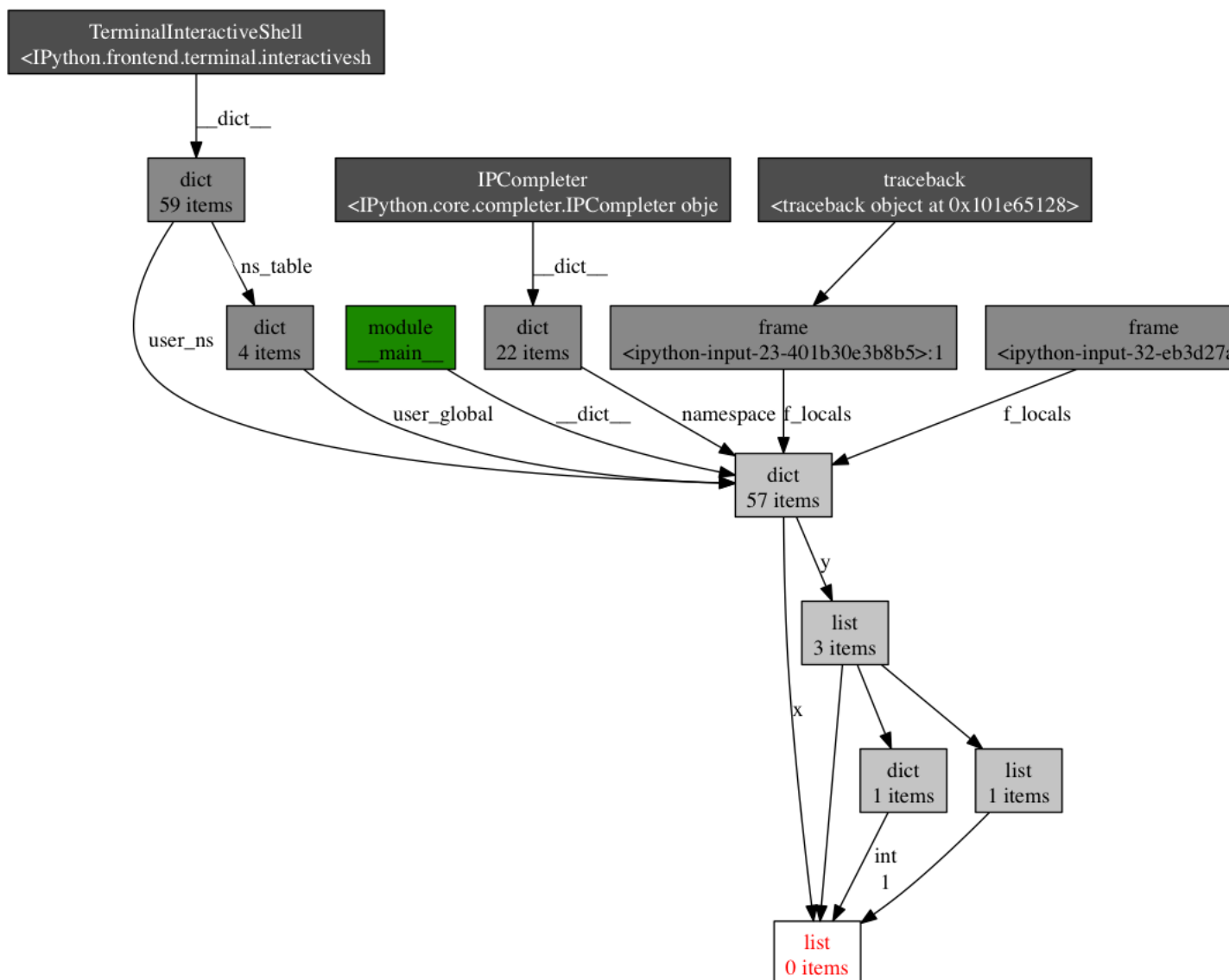
Continuing down this route, we can also see where references to any given object is being held. Let' s take as an example the simple program below:

```
x = [1]
y = [x, [x], {"a":x}]
import pdb; pdb.set_trace()
```

To see what is holding a reference to the variable `x`, run the `objgraph.show_backref()` function:

```
(pdb) import objgraph
(pdb) objgraph.show_backref([x], filename="/tmp/backrefs.png")
```

The output of that command should be a PNG image stored at `/tmp/backrefs.png` and it should look something like this:



The box at the bottom with red lettering is our object of interest. We can see that it's referenced by the symbol `x` once and by the list `y` three times. If `x` is the object causing a memory leak, we can use this method to see why it's not automatically being deallocated by tracking down all of its references.

So to review, [objgraph](#) allows us to:

- show the top N objects occupying our python program's memory
- show what objects have been deleted or added over a period of time
- show all references to a given object in our script

Effort vs precision

In this post, I've shown you how to use several tools to analyze a python program's performance. Armed with these tools and techniques you should have all the information required to track down most memory leaks as well as identify speed bottlenecks in a Python program.

As with many other topics, running a performance analysis means balancing the tradeoffs between effort and precision. When in doubt, implement the simplest solution that will suit your current needs.

References

- [stack overflow - time explained](#)
- [line profiler](#)
- [memory profiler](#)
- [objgraph](#)

I build computer vision software at [Flickr](#). You can get updates on new essays by subscribing to my [rss feed](#). Occasionally, I will send out interesting links on [twitter](#) so follow me if you like this kind stuff.

Related Posts

26 Mar 2014

[Similarity search 101 - Part 2 \(search with vp-trees\)](#)

19 Mar 2014

[Similarity search 101 - Part 1 \(overview\)](#)

14 Dec 2013

[Faster numpy dot product for multi-dimensional arrays](#)37 Comments **Huy Nguyen** Login Recommend 5  Share

Sort by Best



Join the discussion...

**ptone** • 5 years agoThis is a great article, you should consider contributing it to <http://python-guide.org/>

15 ^ | v • Reply • Share ›

**will** → ptone • 3 years ago

This is a great article

^ | v • Reply • Share ›

**wesrog** • 4 years ago

Small typo :)

(pdb) objgraph.show_backref([x], filename="/tmp/backrefs.png")
should be:

(pdb) objgraph.show_backrefs([x], filename="/tmp/backrefs.png")

Other than that, really great article. Thank you!

5 ^ | v • Reply • Share ›

**rafaelolg** • 5 years ago

Nice guide. Thanks.

3 ^ | v • Reply • Share ›

**Samuel** • 4 years ago

Thanks, it is very informative.

2 ^ | v • Reply • Share ›

**dulang** • 2 years ago

this is a great article ,真是好文章

1 ^ | v • Reply • Share ›

**Himanshu Mishra** • 4 years ago

Nice and very useful article. Thank you

1 ^ | v • Reply • Share ›

**gigimon4ik** • 5 years ago

Good article, but what about big software in python, daemons? How to find leaks

1 ^ | v • Reply • Share ›

**Carlos** • 5 years agoYou get finer resolution, at least in Windows, for your timer if you use `time.clock()` instead of `time.time()`.In *nix, `time.clock()` actually gives you CPU time spent by the process, and `time.time()` gives you better resolution, already.<http://stackoverflow.com/qu...>

1 ^ | v • Reply • Share ›

**Kevin Choi** • a year ago

Is there a way to make kernprof profile all methods? or profile of methods by %?

^ | v • Reply • Share ›

**Ron Barak** • 2 years ago

One erratum: once >>you're<< script

Also, a link to <https://docs.python.org/2/l...> (and its Python3 equivalent), which describes cProfile and profile - may make your post more complete.

^ | v • Reply • Share ›



Irq3000 • 2 years ago

A new pure python profiler that allows for line-by-line profiling: pprofile: [https://github.com/vpelletier...](https://github.com/vpelletier/pprofile)

Personally, this is the best python profiler I have ever used.

^ | v • Reply • Share ›



dracodoc • 2 years ago

Great article! It helped me find performance bottle neck quickly. Though I found moving time measurement point involves lots of indent operations which could be cumbersome.

I wrote a different version without using a class. I'm not sure if there is other reason for using a class, but right now my simple script works and easier to use.

For details, please see
[https://dracodoc.wordpress....](https://dracodoc.wordpress.com)

^ | v • Reply • Share ›



Ryan Schwiebert • 2 years ago

I couldn't get line_profiler to work. using python 2, it terminates with ImportError: ... usr/local/.../_line_profiler.so: undefined symbol: PyUnicodeUCS4_DecodeUTF8

Then thinking this might be for python3, I tried that, but similarly
 ImportError: ... undefined symbol: PyString_Type

^ | v • Reply • Share ›



rodrev • 2 years ago

Thank you Hui. Good article.

I used a timer script in \$PYTHONPATH that prints formatted elapsed time like so:

```
from time import time
from timer import timer
go = time()
runprogram()
end = time()
timer(go, end)
```

It was annoying to type for every .py script. I like your Timer wrapper better. I combined it with my timer, to print fast results like so:

```
>>> 1.968050e-02 seconds
or slower results (for large data):
>>> 0h 10m 32s
```

Thought I would share it here if it's ok with you. Call it timer(dot)py. Code is here: [https://gist.github.com/rod...](https://gist.github.com/rodrev)

use like so:

[see more](#)

^ | v • Reply • Share ›



Guest • 2 years ago

Thank you Hui. Good article.

I used a timer.py script in \$PYTHONPATH that prints formatted elapsed time like so:

```
from time import time
from timer import timer
go = time()
runprogram()
end = time()
timer(go, end)
```

It was annoying to type for every .py script. I like your Timer wrapper better. I combined it with my timer, to print fast results like so:

```
>>> 1.968050e-02 seconds
or slower results (for large data):
>>> 0h 10m 32s
```

Thought I would share it here if it's ok. Call it timer.py

use like so:

[see more](#)

^ | v • Reply • Share ›



rodrev → Guest • 2 years ago

sorry for the double post. I used my google account by mistake and then tried to delete the post but it just removed my name instead. You can remove this one now called 'Guest' if you like.

^ | v • Reply • Share ›



Mike iLL • 3 years ago

if you have a problem installing line-profiler, it may be because there is not a "stable" version as recognized by pip. as per <http://stackoverflow.com/qu...>, install using

```
sudo pip install --pre line_profiler
```

^ | v • Reply • Share ›



Mike iLL • 3 years ago

Awesome, man! This should be really useful with testing my echonest remix code.

^ | v • Reply • Share ›



Michal • 3 years ago

Amazing job !

^ | v • Reply • Share ›



Leo Liang • 3 years ago

thanks for the great post. saved my life:D

^ | v • Reply • Share ›



amateur • 3 years ago

where is redis module?

^ | v • Reply • Share ›



Harikrishna • 3 years ago

Informative, Is there any way to redirect output a text file instead of printing on the screen. Like a output text file, Thanks in advance

^ | v • Reply • Share ›



Sumana Harihareswara • 3 years ago

Thanks for the article!

with Timer as t:

needs to be

with Timer() as t:

^ | v • Reply • Share ›



TRAN • 3 years ago

Hi, it's very nice guide. I imagine if you have some suggestions tools for Python 3?

^ | v • Reply • Share ›



Jordan Rinke • 3 years ago

As many others have said, great article.

^ | v • Reply • Share ›



Cornelius Lilge • 3 years ago

Did not work for me @ python 2.7 / windows:

Downloading/unpacking line-profiler

Could not find a version that satisfies the requirement line-profiler (from versions: 1.0b1, 1.0b2, 1.0b3)

Cleaning up...

No distributions matching the version for line-profiler

^ | v • Reply • Share ›



tushar12123 → Cornelius Lilge • 3 years ago

This problem got resolved on my ubuntu machine when I type the following in my terminal :

```
$ pip install line_profiler==1.0b3
```

1 ^ | v • Reply • Share ›



Mike III Kilmer → Cornelius Lilge • 3 years ago

```
sudo pip install --pre line_profiler
```

^ | v • Reply • Share ›



Sijo Jose • 3 years ago

Very helpful...Keep it up...

^ | v • Reply • Share ›



Haridas N • 3 years ago




Great article to check the python runtime behaviors. I'm bookmarking it for later usage. Thank you.



^ | v • Reply • Share ›



Wendal Chen • 3 years ago

great article


^ | v • Reply • Share ›
will • 3 years ago
This is a great article.
^ | v • Reply • Share ›
Ha Pham • 4 years ago
Really useful article, thank you for sharing analyzing tips!
^ | v • Reply • Share ›
Tichis Silva • 4 years ago
Please look at this:


<http://pythonfasterway.uni.me/>
^ | v • Reply • Share ›
Robert Lugg ➔ **Tichis Silva** • 2 years ago
Great link Tichis!
^ | v • Reply • Share ›
doublemarket • 4 years ago
Hello, thank you for great post!
Since the post is very useful, I personally translate it into Japanese.
<http://yakst.com/ja/posts/42>

If you have a problem for that, please let me know.
Then I'll make my translation private or delete from my site.
^ | v • Reply • Share ›





ALSO ON HUY NGUYEN

Similarity search 101 - Part 2 (search with vp-trees)

7 comments • 3 years ago •


milan — Is it possible to represent this in a relational database?**Faster numpy dot product for multi-dimensional arrays**

3 comments • 3 years ago •


Dmitri — Hey, thanks for an interesting post! Yet I have one question, do you use paralleled or unparalleled ... [Subscribe](#)  [Add Disqus to your site](#) [Add Disqus](#) [Add](#)  [Privacy](#)