Measuring memory usage in Python: it's tricky!

by Itamar Turner-Trauring
Last updated 22 Jun 2021, originally created 21 Jun 2021

If you want your program to use less memory, you will need to measure memory usage. You'll want to measure the current usage, and then you'll need to ensure it's using less memory once you make some improvements.

It turns out, however, that measuring memory usage isn't as straightforward as you'd think. Even with a highly simplified model of how memory works, different measurements are useful in different situations.

In this article you'll learn:

- A simplified but informative model of how memory works.
- Two measures of memory–resident memory and allocated memory–and how to measure them in Python.
- The tradeoffs between the two.

A (not so) simple example

Consider the following code:

```
>>> import numpy as np
>>> arr = np.ones((1024, 1024, 1024, 3), dtype=np.uint8)
```

This creates an array of 3GB–gibibytes, specifically–filled with ones. Naively, once this code runs we expect the process to be using a little bit more than

3GB.

One way to measure memory is using "resident memory", which we'll define later in the article. We can get this information using the handy `psutil` library, checking the resident memory of the current process:

```
>>> import psutil
>>> psutil.Process().memory_info().rss / (1024 * 1024)
3083.734375
```

With this particular measurement, we're using 3083MB, or 3.08GB, and the difference from the array size is no doubt the memory used by the Python interpreter and the libraries we've imported. Success!

But resident memory is tricker than it seems. Let's say I go and open some websites in my browser, leaving the Python interpreter running the background. Then I switch back to the interpreter, and run the exact same command again:

```
>>> psutil.Process().memory_info().rss / (1024 * 1024)
2902.12109375
```

What's going on? Why did 200MB of memory disappear?

To find the answer we'll need to learn a bit more about how the operating system manages memory.

A simplified model of memory allocation

A running program *allocates* some memory, i.e. gets back an address in virtual memory from the operating system. Virtual memory is a process-specific address space, essentially numbers from `0` to 2^{64}`-1`, where the process can read or write bytes.

In a C program you might use APIs like `malloc()` or `mmap()` to do so; in

Python you just create objects, and the Python interpreter will call `malloc()` or `mmap()` when necessary. The process can then read or write to that particular address and consecutive bytes.

We can see this in action by using the Linux utility `ltrace` to trace calls to `malloc()`. Given the following program:

```
import numpy as np
arr = np.ones((170_000,), dtype=np.uint8)
```

We can run Python under `ltrace`:

```
$ ltrace -e malloc python ones.py
...
_multiarray_umath.cpython-39-x86_64-linux-gnu.so->malloc(170
000) = 0x5638862a45e0
...
```

Here's what happening:

1. Python create a NumPy array.
2. Under the hood NumPy calls `malloc()`.
3. The result of that `malloc()` is an address in memory: `0x5638862a45e0`.
4. The C code used to implement NumPy can then read and write to that address and the next consecutive 169,999 addresses, each address representing one byte in virtual memory.

Where are these 170,000 bytes stored?

- ➢ They can be stored in RAM; this is the default.
- ➢ They can be stored on the computer's hard drive or disk, in which case they're said to be stored in *swap*.
- ➢ Some of the bytes might be stored in RAM and some in swap.
- ➢ Finally, they can be stored nowhere at all.

For now, we'll ignore that confusing last case, and just focus on the first

three situations.

Resident memory is RAM usage

RAM is fast, and your hard drive is slow... but RAM is also expensive, so typically you'll have far more hard drive space than RAM. For example, the computer I'm writing this on has about 500GB of hard drive storage, but only 16GB RAM.

Ideally, all of your program's memory would be stored in fast RAM, but the various processes running on your computer might have allocated more memory than is available in RAM. If that happens, the operating system will move–or "swap"–some data from RAM to hard drive. In particular, it will try to swap data that isn't actively being used.

And now we're ready to define our first measure of memory usage: resident memory. Resident memory is how much of the process' allocated memory is resident–or stored–in RAM.

In our first example we started out with all 3GB the allocated array stored in RAM.

3GB array (=allocated memory)
3GB in RAM (=resident memory)

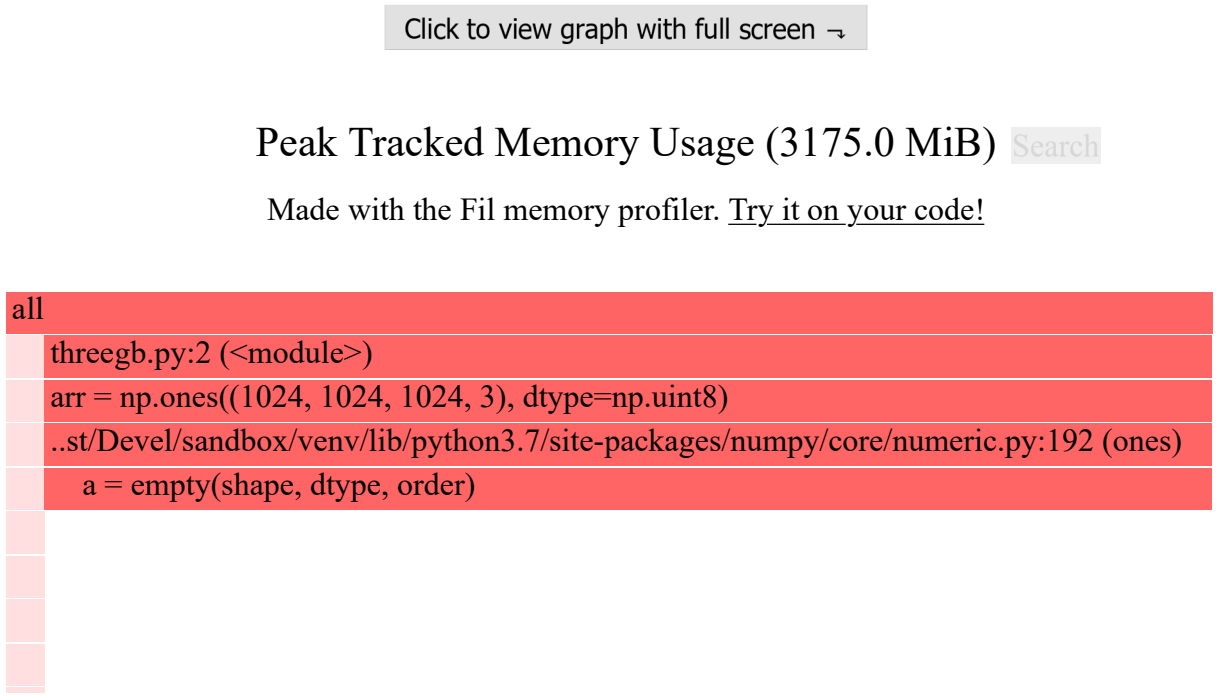
Then when I opened a bunch of browser tabs, loading those websites used quite a lot of RAM, and so the operating system decided to swap some data from RAM to disk. And part of the memory that got swapped was from our array. As a result, the resident memory for our Python process went down: the data was all still accessible, but some of it had been moved to disk.

3GB array (=allocated memory)	
2.8GB in RAM (=resident memory)	0.2GB on disk

An alternative: allocated memory

It would be useful to measure allocated memory, to always get 3GB back regardless of whether the operating system put the data in RAM or swapped it to disk. This would give us consistent results, and also tell us how much memory the program actually asked for.

In Python (if you're on Linux or macOS), you can measure allocated memory using the Fil memory profiler, which specifically measures *peak* allocated memory. Here's the output of Fil for our example allocation of 3GB:



The tradeoffs between resident memory and allocated memory

As a method of measuring memory, resident memory has some problems:

- Other processes can distort the results by using up limited RAM and encouraging swapping, as they did in our 3GB example above.
- Because resident memory is capped at available physical RAM, you never actually learn how much memory the program asks for, once you hit that ceiling. If you have 16GB of RAM you won't be able to

distinguish between a program that needs 17GB memory and a program that needs 30GB of memory: they will both report the same resident memory.

Allocated memory, on the other hand, is not affected by other processes, and tells you what the program actually requested.

Of course, resident memory does have some advantages over allocated memory:

- That swapped memory may well never be used: imagine creating an array, forgetting to delete the reference, and then never actually using it again for the rest of the program. If it gets swapped, that's fine, and arguably we shouldn't be measuring it.
- More broadly, because resident memory measures actually used RAM from the operating system perspective, it can capture edge cases that aren't visible to the allocated memory tracking.

Let's look at an example of one such edge case.

Phantom memory!

In all our examples so far we've been allocating arrays that are full of ones. If we're measuring allocated memory, what the array is filled with makes no difference: we can switch to creating arrays full of zeroes, and still get the exact same result.

But on Linux, resident memory tells us an interesting story:

```
>>> import numpy as np
>>> import psutil
>>> arr = np.zeros((1024, 1024, 1024, 3), dtype=np.uint8)
>>> psutil.Process().memory_info().rss / (1024 * 1024)
28.5546875
```

Once again, we've allocated a 3GB array, this time full of zeroes. We measure resident memory and–the array isn't counted, resident memory is just 29M.

Where did the array go?

It turns out that Linux doesn't bother storing all those zeroes in RAM. Instead, it will add chunks of zeroes to RAM only when the data is actually accessed–until then no RAM is used. Until that happens, allocated memory will report 3GB, but resident memory will notice that those 3GB aren't actually using any resources.

Allocated memory is a good starting point

It's worth keeping in mind that this is still a quite simplified model of memory usage. It doesn't cover the file cache, or memory fragmentation in the allocator, or other available metrics–some useful, some less so–you can get from Linux.

That being said, for many applications allocated memory is probably sufficient as a first-pass measure to help you optimize your program's memory use. Allocated memory tells you what your application actually asked for, and that's usually what you're going to have to optimize.

Learn even more techniques for reducing memory usage—read the rest of the Larger-then-memory datasets guide for Python.

Next: Fil: a new Python memory profiler for data scientists and scientists
Previous: The mmap() copy-on-write trick: reducing memory usage of array

- Home
- Products
- Training services
- About me
- Feed
- Privacy policy
- Terms & Conditions