Precious Memory

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

Most of the Python code that I write isn't part of an exotic framework or huge application. Instead, it's usually related to a mundane data analysis task involving a CSV file. It isn't glamorous, but Python is an effective tool at getting the job done without too much fuss. When working on such problems, I prefer to not worry too much about low-level details (I just want the final answer). However, if you use Python for manipulating a lot of data, you may find that your scripts use a large amount of memory. In this article, I'm going to peek under the covers of how memory gets used in a Python program and explore options for using it more efficiently. I'll also look at some techniques for exploring and measuring the memory use of your programs. Disclosure: Python 3 is assumed for all of the examples, but the underlying principles apply equally to Python 2.

Reading a Large CSV File

My Chicago office is located along a major bus route, the trusty #22 that will take me down the road to Wrigley Field if I want to avoid work during the summer. It tends to be a pretty busy route, but just how busy? Chicago, being a data-friendly city, has historical bus ridership data posted online [1]. You can download it as a CSV file. If you do, you'll get a 13.8 MB file with 676,476 lines of data that give you the ridership of every bus route in the city on every day of the year going back to the year 2001. It looks like this:

```
route,date,daytype,rides
3,01/01/2001,U,7354
4,01/01/2001,U,9288
6,01/01/2001,U,6048
8,01/01/2001,U,6309
9,01/01/2001,U,11207
...
```

By modern standards, a 13.8 MB CSV file isn't so large. Thus, I'm inclined to grab it using Python's `csv` module. Problem solved:

```
>>> import csv
>>> with open('cta.csv') as f:
...     rows = list(csv.DictReader(f))
...
>>> len(rows)
676476
>>> rows[0]
{'date': '01/01/2001', 'route': '3', 'rides': '7354', 'daytype': 'U'}
>>>
```

Now let's tabulate the ridership totals across all of the bus routes using the `collections` module:

```
>>> from collections import Counter
>>> ride_counts = Counter()
>>> for row in rows:
...     ride_counts[row['route']] += int(row['rides'])
...
>>> ride_counts['22']
104039097
>>>
```

While we're at it, why don't we find out the five most common bus routes.

```
>>> ride_counts.most_common(5)
[('79', 153736884), ('9', 138645554), ('49', 113908939),
 ('4', 111154851), ('66', 110746972)]
>>>
```

Great. Before you quit, however, go look at the memory use of the Python interpreter in your system process viewer—you'll find that it's using nearly 300 MB of RAM (maybe more). Yikes! For a 13.8 MB input file, that sure seems like a lot—almost as much as some of the minimally useful apps on my phone. The horror.

Measuring Memory Use

Measuring the memory use of a Python program in a portable way was not an entirely easy task until somewhat recently. Yes, you could always go view the Python process in the system task viewer, but there were no standard library modules to help you out. This changed somewhat in Python 2.6 with the addition of the `sys.getsizeof()` function. It lets you determine the size in bytes of individual objects. For example:

```
>>> import sys
>>> a = 42
>>> sys.getsizeof(a)
28
>>> b = 'hello world'
>>> sys.getsizeof(b)
60
>>>
```

Unfortunately, the usefulness of `sys.getsizeof()` is a bit limited. For containers such as lists and dicts, it only reports the size of the container itself, not the cumulative sizes of the items contained inside. It's subtle, but you can see this yourself if you look carefully at this example where the combined size of two items in a list is smaller than the reported size of the list itself:

```
>>> a = 'hello'
>>> b = 'world'
>>> items = [a, b]
>>> sys.getsizeof(a)
54
>>> sys.getsizeof(b)
54
>>> sys.getsizeof(items)  # Notice size is less than combined
                          # a, b size
80
>>>
```

Containers also present complications in determining an accurate use. For example, the same object might appear more than once such as in a list of `[a, a, b, b]`. Also, Python tends to aggressively share immutable values under the covers. So it's not a simple case where you can just add up the byte totals for all of the items in a container and get an accurate figure. Instead you'd need to gather information on all unique objects using their object IDs like this:

```
>>> items = [a, a, b, b]
>>> unique_items = { id(item): sys.getsizeof(item) for item
      in items }
>>> total_size = sys.getsizeof(items) + sum(unique_items.
      values())
204
>>>
```

If you had deeply nested data structures, you'd have to take further steps to recursively traverse the entire data structure. Needless to say, it gets ugly. Just to illustrate, here's how you would measure the memory usage of the list holding all of that bus data.

```
>>> unique_objects = { id(rows): rows }
>>> unique_objects.update((id(row), row) for row in rows)
>>> unique_objects.update((id(val), val) for row in rows for
      val in row.values())
>>> sum(sys.getsizeof(val) for val in unique_objects.values())
308977196
>>>
```

Starting in Python 3.4, you can obtain global memory statistics using the `tracemalloc` module [2]. This module allows you to selectively monitor the memory use of Python and have it record memory allocations. It's not so useful for small measurements, but you can use it in a script:

```
import tracemalloc
import csv

def read_data(filename):
    with open(filename) as f:
        return list(csv.DictReader(f))

tracemalloc.start()
rows = read_data('cta.csv')
print(len(rows), 'Rows')
print('Current: %d, Peak %d' % tracemalloc.get_traced_memory())
```

If I run this on my machine with Python 3.5, I get the following output:

```
676476 Rows
Current: 308979047, Peak 309009543
```

The reported memory use is ever so slightly higher than what was calculated directly with `sys.getsizeof()`, but basically the two figures agree.

Exploring Common Data Structure Choices

Given the large memory footprint associated with reading this file, you might consider other choices for representing a simple record, such as a list, tuple, or class instance. Here are several different functions that read the data in different forms:

```
import csv

def read_data_as_dicts(filename):
    with open(filename) as f:
        return list(csv.DictReader(f))

def read_data_as_lists(filename):
    with open(filename) as f:
        rows = csv.reader(f)
        headers = next(rows)
        return list(rows)

def read_data_as_tuples(filename):
    with open(filename) as f:
        rows = csv.reader(f)
        headers = next(rows)
        return [tuple(row) for row in rows]

class RideData(object):
    def __init__(self, route, date, daytype, rides):
        self.route = route
        self.date = date
        self.daytype = daytype
        self.rides = rides

def read_data_as_instances(filename):
    with open(filename) as f:
        rows = csv.reader(f)
        headers = next(rows)
        return [ RideData(*row) for row in rows ]
```

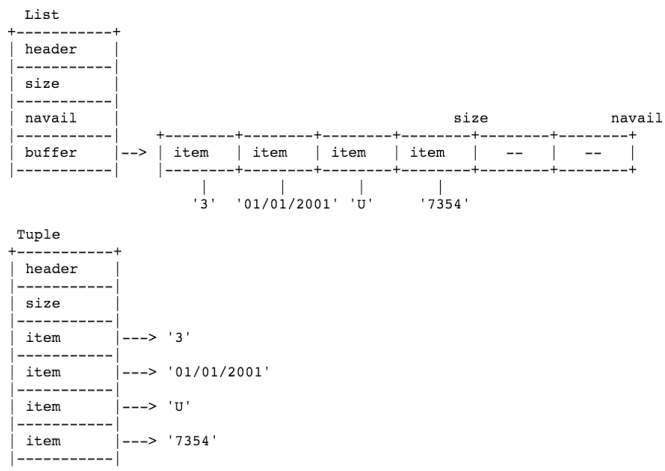
If you run and measure these different functions using `tracemalloc`, you will get memory use as follows:

Record Type	Memory Use (MB)
Dict	294.7
List	170.8
Tuple	160.1
Instance	268.5

In these results, you find that tuples provide the most efficient storage. This shouldn't be a surprise, but there are still some subtle aspects to the results. For example, what explains the 10 MB gap between tuples and lists? On the surface it doesn't seem like there would be much difference between the two structures given that they're both "list like." We can investigate with `sys.getsizeof()`:

```
>>> a = ('3', '01/01/2001', 'U', '7354')
>>> b = ['3', '01/01/2001', 'U', '7354']
>>> import sys
>>> sys.getsizeof(a)
80
>>> sys.getsizeof(b)
96
>>>
```

Here, we find that there is a 16-byte difference in storage between a list and tuple. Added up across the 676,476 rows of data, that amounts to about 10 MB of storage. The 16-byte difference is due to the fact that lists are a little more complicated than they might first seem. For one, since lists are mutable, their size can change as elements are added or removed. To manage this, lists internally contain a memory pointer to a resizable memory buffer where items are stored. Tuples, being immutable, don't have to handle resizing. Thus, the items in a tuple can be stored directly at the end of the underlying tuple structure. Lists also overallocate their internal storage so as to make repeated `append()` operations faster (this is to minimize a potentially expensive memory reallocation each time a new element is added). For example, a list containing only five items might actually have room to store eight items without asking for more space. To manage this, lists maintain an extra counter of how much total space is available in addition to a counter that records the actual number of elements used. Here is a diagram that illustrates the difference in the memory layout of a tuple versus a list:



The header portion contains some bookkeeping information, including the object's type and the reference count used in memory management. This is the same for all objects. The 16-byte difference in tuple/list storage is explained by the presence of an extra memory pointer (buffer) and counter (navail) on lists. Depending on the amount of unused space, lists might even be a bit larger.

Another surprising result is the efficiency of instances over dictionaries—especially if you happen to know that instances are actually built using dictionaries. For example:

```
>>> r = RideData('3', '01/01/2001', 'U', '7354')
>>> r.__dict__
{ 'route': '3', 'date': '01/01/2001', 'daytype': 'U', 'rides': '7354'
}
>>>
```

Thus, what explains the 26 MB advantage of using instances over dictionaries? As it turns out, this is also another memory optimization. When creating a lot of instances, Python makes an assumption that the dictionaries for all of the instances will probably contain the exact same set of keys. It makes sense—all objects are initialized in `__init__()` and are likely to have an identical underlying structure. Python exploits this and creates what's known as a key-sharing dictionary as described in PEP 412 [3]. In a nutshell, the keys for the instance data are split off from the normal dictionary and stored in a shared structure. It makes for a slightly smaller dictionary structure. You can investigate:

```
>>> c = { 'route': '3', 'date': '01/01/2001', 'daytype': 'U',
'rides': '7354' }
>>> sys.getsizeof(c)
288
>>> d = RideData('3', '01/01/2001', 'U', '7354')
>>> d
<__main__.RideData object at 0x101ad4f60>
>>> d.__dict__
{ 'route': '3', 'date': '01/01/2001', 'daytype': 'U', 'rides': '7354'
}
>>> sys.getsizeof(d.__dict__)  # Size of instance dict
192
>>>
```

Here, you see that the instance dictionary is quite a bit smaller than a normal dictionary. However, you can't forget that instances also contain some state, including the class and reference count:

```
>>> sys.getsizeof(d)   # Size of the instance structure
56
>>>
```

So, in this example, you'll find that an instance requires 56 bytes of storage plus the storage required for the instance dictionary. Added together, you find that an instance requires 248 bytes vs. 288 bytes for a normal dictionary. Multiplied by the 676,476 records, you get a savings of about 26 MB.

Named Tuples

Tuples are efficient, but one downside is that they often lead to code where you do a lot of ugly indexing. For example:

```
>>> rows = read_data_as_tuples('cta.csv')
>>> from collections import Counter
>>> ride_counts = Counter()
>>> for row in rows:
...     ride_counts[row[0]] += int(row[3])
...
>>>
```

You can clean this up using the `namedtuple()` function to define a class. For example:

```
from collections import namedtuple
RideTuple = namedtuple('RideTuple', ['route','date','daytype',
'rides'])
```

The `namedtuple()` function performs a neat trick using properties that produces a class roughly equivalent to this:

```
class RideTuple(tuple):
    __slots__ = ()      # Explained in next section
    @property
    def route(self):
        return self[0]
    @property
    def date(self):
        return self[1]
    @property
    def daytype(self):
        return self[2]
    @property
    def rides(self):
        return self[3]
```

In this class, properties have been added to pull attributes from a specific tuple index. This gives you nice access to those values via the dot (.) operator. For example:

```
>>> r = RideTuple('3','01/01/2001','U','7354')
>>> r.route
'3'
>>> r.date
'01/01/2001'
>>> r[0]
'3'
```

```
>>> r[1]
'01/01/2001'
>>>
```

Named tuples also offer a cautionary tale of measuring Python's memory use—namely, that you can't always trust it to tell you the truth! For example, suppose you measure the memory of a single named tuple versus a tuple:

```
>>> a = ('3', '01/01/2001', 'U', 7354')
>>> b = RideTuple('3','01/01/2001','U','7354')
>>> sys.getsizeof(a)
80
>>> sys.getsizeof(b)
80
>>>
```

Here, you will find that the memory is identical. That looks good. However, if you run two versions of code under `tracemalloc`, you'll find that they have different behavior.

Record Type	Memory Use (MB)
Tuple	160.1
Named tuple	165.3

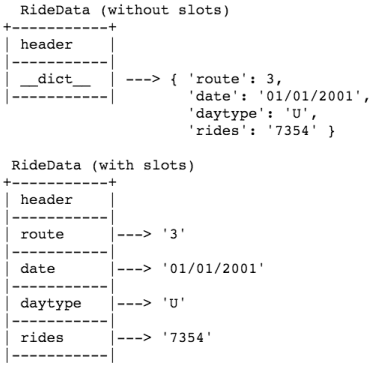
For reasons unknown, named tuples allocate an extra machine word (8 bytes on a 64-bit machine) for each instance. Added up over the 676,476 rows of our data set, that amounts to an extra 5 MB. If there's any takeaway, the results of `sys.getsizeof()` are not always to be trusted. If you must know, objects self-report their size using a special method `__sizeof__()` which could be implemented incorrectly. If you really care about accuracy, it's a good idea to measure memory use a few different ways.

Slots

A somewhat lesser known technique for saving memory is to define a class with a `__slots__` specifier like this:

```
class RideData(object):
    __slots__ = ('route', 'date', 'daytype', 'rides')
    def __init__(self, route, date, daytype, rides):
        self.route = route
        self.date = date
        self.daytype = daytype
        self.rides = rides
```

Normally, instances are represented by a dictionary. However, if you use slots, you're giving a hint about how many attributes will be stored. Python uses this to eliminate the instance dictionary and rearrange the storage of attributes into something that looks a lot like a tuple. Here is a diagram showing how instances are stored with and without slots:



Remarkably, a class that uses slots is even slightly more efficient than one using a tuple. For example, if you run a test under `tracemalloc`, you'll get these results:

Record Type	Memory Use (MB)
Tuple	160.1
Instance with slots	155.0

The savings is due to the fact that unlike a tuple, instances don't support indexing of attributes (e.g., `r[n]`). Thus, it is not necessary for a size to be stored on a per-instance basis. The attributes are merely loaded and stored from a hardwired position known in advance. The exact mechanism is almost exactly the same as the attribute properties defined on a named tuple.

Using the Appropriate Datatypes

In our example, we were being lazy and storing the numeric ride data as a string (e.g., `'7354'`) instead of as an integer (7354). However, strings are not the most efficient representation. Let's explore:

```
>>> a = '7354'
>>> b = 7354
>>> c = 7354.0
>>> sys.getsizeof(a)
53
>>> sys.getsizeof(b)
28
>>> sys.getsizeof(c)
24
>>>
```

As you can see, storing the number as an integer saves 25 bytes. However, storing the value as a floating point number saves a bit more. Integers require more space because they are allowed to grow to arbitrary magnitude. To handle this, they must not only store the integer value, but some additional sizing information. Floats don't need this.

By changing just one column of the data to a float, we save about 18 MB of memory. So being smart about what you store makes a difference.

Value Sharing

Under the covers, Python memory management is based on memory pointers. For example, suppose you make a list and "copy" it to another variable:

```
>>> a = [1,2,3]
>>> b = a
>>>
```

This didn't actually make a copy of the list. Instead, the names "a" and "b" both refer to the same object. If you change the list, it's reflected in both variables.

```
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> b
[1, 2, 3, 4]
>>>
```

The `id()` function will give you the object identity, a unique integer value. You can use this to see that `a` and `b` in the above example are the same object.

```
>>> id(a)
56623488
>>> id(b)
56623488
>>>
```

Now, how to use this? When reading certain kinds of data sets, you might encounter a lot of repetition. To illustrate, let's grab the bus data again.

```
>>> f = open('cta.csv')
>>> rows = list(csv.DictReader(f))
>>> unique_routes = set(row['route'] for row in rows)
>>> len(unique_routes)
182
>>> route_ids = set(id(row['route']) for row in rows)
>>> len(route_ids)
634285
>>>
```

What you're seeing here is that the data contains only 182 unique values for the "route" field, yet those values are stored in 634,285 unique objects. It's a bit odd that there aren't 676,476 unique values corresponding to the length of the entire data set. As it turns out, Python caches objects representing all single-letter ASCII strings. Thus routes 1–9 get special treatment. You can verify this:

```
>>> route_ids = set(id(row['route']) for row in rows if
len(row['route'])==1)
>>> len(route_ids)
9
>>>
```

Perhaps you can take a similar caching strategy for reusing the rest of the values. Here is a simple function that caches strings:

```
def cache(value, _values = {}):
    if value not in _values:
        _values[value] = value
    return _values[value]
```

Next, you can apply the cache function to selected values during instance creation. For example:

```
class RideData(object):
    __slots__ = ['route','date','daytype','rides']
    def __init__(self, route, date, daytype, rides):
        self.route = cache(route)
        self.date = cache(date)
        self.daytype = daytype
        self.rides = float(rides)
```

Making this change, the storage required for our example data is reduced down to about 68 MB—not too bad considering it started out at over 300 MB.

Changing Your Orientation

So far, we have worked to represent the data as a list of records—varying the representation of each record. However, another approach is to turn everything sideways and represent the data as a collection of columns. For example, suppose you read the data using this function:

```
def read_data_as_columns(filename):
    route = []
    date = []
    daytype = []
    rides = []
    with open(filename) as f:
        for row in csv.DictReader(f):
            route.append(cache(row['route']))
            date.append(cache(row['date']))
            daytype.append(row['daytype'])
            rides.append(float(row['rides']))

    return {
        'route': route,
        'date': date,
        'daytype': daytype,
        'rides': rides
        }
```

Making this change reduces the memory use to about 38 MB. However, it also shatters your head as working with the resulting data is wacky. Instead of getting a single list of records, you get four lists representing each column. For example:

```
>>> columns = read_data_as_columns('cta.csv')
>>> len(columns)
4
>>> columns['route'][0]
'3'
>>> columns['date'][0]
'01/01/2001'
>>>
```

Yes, you can work with the data like this, but doing so might require a bit of ingenuity and increase your job security. You would probably be better off using a third party library such as Pandas, which also stores its data in a column form [4]. This brings us to the last important point about memory. Third party libraries often rely on C extensions and code outside of Python that can't be measured accurately using the tools described here. For example, you can try this experiment with Pandas:

```
>>> import pandas
>>> import tracemalloc
>>> tracemalloc.start()
>>> data = pandas.read_csv('cta.csv')
>>> tracemalloc.get_traced_memory()
(433375, 471219)
>>> import sys
>>> sys.getsizeof(data)
135868754
>>>
```

Pandas is efficient, but it's not so efficient that it's storing all of the data in only 430 KB. Nor is the reported size of the `data` variable 135 MB. A look in the task viewer shows Python actually using about 56 MB of memory. Bottom line: if you're using certain kinds of Python extensions, the memory profiling tools described here might not work.

If You Liked It, You Should Have Put a Generator on It

In the end, maybe it's best to ask yourself if you actually need to read all of the data at once. Perhaps a generator function can do the trick:

```
import csv
from collections import Counter
def read_data(filename):
    with open(filename) as f:
        rows = csv.DictReader(f)
        for row in rows:
            yield { **row, 'rides':int(row['rides']) }

ride_counts = Counter()
for row in read_data('cta.csv'):
    ride_counts[row['route']] += row['rides']
```

If you run this version under `tracemalloc`, you'll find that it tabulates all of the data and uses only 36K of memory. Yes, generators are your friend.

Final Thoughts

This article has looked at a variety of issues surrounding Python memory use. There are probably a few important takeaways. First, there are some built-in tools such as `sys.getsizeof()` and the `tracemalloc` that you can use to investigate the memory use of your program. They're not always reliable, but when used in combination, you can often get a pretty good idea of what's happening. Second, there are a variety of ways in which you can represent data to reduce the memory footprint. For example, using `__slots__` in a class definition. Small details, such as your choice of low-level data representation and value sharing with caching, can also make a big impact. Last but not least, different data organizations (e.g., rows vs. columns) can be important.

References

[1] CTA-Ridership Data: https://data.cityofchicago.org/Transportation/CTA-Ridership-Bus-Routes-Daily-Totals-by-Route/jyb9-n7fm.

[2] `tracemalloc` module: https://docs.python.org/3/library/tracemalloc.

[3] PEP 0412 -- Key-Sharing Dictionary: https://www.python.org/dev/peps/pep-0412/.

[4] Pandas: pandas.pydata.org.