Building a Better Dictionary

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009) and *Python Cookbook* (3rd Edition, O'Reilly & Associates, 2013). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply/index.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

One of the software projects that I maintain is the PLY parser generator (http://www.dabeaz.com/ply). In a nutshell, PLY is a Python implementation of the classic lex and yacc tools used for writing parsers, compilers, and other related programs. It's also not the kind of program that tends to change often—to be sure, I'm not aware of any sort of space-race concerning the implementation of LALR(1) parser generators (although perhaps there's some startup company Lalrly.com just waiting to strike parsing gold).

As a stable piece of software, PLY only receives occasional bug reports, which are mostly in the form of minor feature requests; however, I recently received a report that PLY was randomly failing its unit tests on Python 3.3. Specifically, if you ran its unit test suite twice in succession, different sets of unit tests would fail each time. For a program involving no randomness or threads, this development was puzzling to say the least.

This problem of randomly failing unit tests was ultimately tracked down to a recent security-related change in Python's dictionary implementation. I'll describe this change a bit later, but this incident got me thinking about the bigger picture of Python dictionaries. If anything, it's safe to say that the dictionary is part of the bedrock that underlies the entire Python interpreter. Major parts of the Python language, such as modules and objects, use dictionaries extensively. Moreover, they are widely used as data structures in user applications. Last, but not least, the implementation of dictionaries is one of the most studied and tuned parts of the interpreter.

Given their importance, you might think that the dictionary implementation would be something that's set in stone. To be sure, Python's core developers are reluctant to make changes to something so important; however, in the past couple of years, the implementation of dictionaries has been evolving in interesting and unusual ways. In this article, I hope to peel back the covers a little bit and discuss how dictionaries work along with some notable recent changes.

Dictionaries as Data Structures

Most Python programmers are familiar with using a dictionary as a simple data structure. For example:

```
s = {
    'name': 'ACME',
    'shares': 100,
    'price': 123.45
}
```

A dictionary is simply a mapping of keys to values. To perform calculations, you simply access the key names:

```
>>> s['shares'] * s['price']
12345.0
>>> s['shares'] = 75
>>> s['name']
'ACME'
>>>
```

Dictionaries are unordered. Thus, if you look at the ordering of the keys, they're usually not in the same order as originally specified when the dictionary was created. For example:

```
>>> s
{'price': 123.45, 'name': 'ACME', 'shares': 75}
>>> s.keys()
['price', 'name', 'shares']
>>>
```

Although the lack of ordering sometimes surprises newcomers, it's not something that causes concern in most programs; it's just an artifact of the implementation.

From dictionaries to classes is only a small step. For example, suppose you have a class like this:

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

If you make an instance, it's actually just a thin wrapper around a dictionary. For example:

```
>>> s = Stock('ACME', 100, 123.45)
>>> s.shares * s.price
12345.0
>>> s.__dict__
{'price': 123.45, 'name': 'ACME', 'shares': 100}
>>>
```

Naturally, most of this is old news to anyone who's been programming in Python for a while.

Dictionary Implementation

Under the covers, dictionaries are implemented as hash tables. Each entry in a dictionary is represented by a structure (hashval, key, value) where hashval is an integer hashing code, key is a pointer to the key value, and value is a pointer to the value. The special hash value used in this triple is not something you normally think about, but it's easily obtained using the built-in hash() function (note: to get examples that exactly match what's shown, use Python 2 compiled for a 64-bit platform):

```
>>> hash('name')
-4166578487145698715
>>> hash('shares')
-5046406209814648658
>>>
```

When an empty dictionary is first created, a small eight-element array of dictionary entry structures is allocated. Entries are inserted into this array at positions determined by bit-masking the above integer hash codes. For example:

```
>>> hash('name') & 7
5
>>> hash('shares') & 7
6
>>> hash('price') & 7
2
>>>
```

The numerical order of the above positions determine the order in which keys will appear when you look at a dictionary. For example:

```
>>> s.keys()
['price', 'name', 'shares']
>>>
```

If you add a new key to a dictionary, its insertion position is determined in the same way. For example:

```
>>> hash('time') & 7
7
>>> s['time'] = '9:45am'
>>> s
{'price': 123.45, 'name': 'ACME', 'shares': 75, 'time': '9:45am'}
>>>
```

If two keys map to the same index, a new position is found by repeatedly perturbing the index to a new value until a free slot is found. Without explaining the rationale for the mathematical details, the following session illustrates what happens if you add a new entry `s['account'] = 1` to the above dictionary:

```
>>> hval = hash('account')
>>> index = hval & 7
>>> index          # Collision with "price"
2
>>> perturb = hval
>>> index = (index << 2) + index + perturb + 1
>>> index & 7       # Collision with "name"
5
>>> perturb >>= 5
>>> index = (index << 2) + index + perturb + 1
>>> index & 7       # Collision with "name"
5
```

```
>>> perturb >>= 5
>>> index = (index << 2) + index + perturb + 1
>>> index & 7        # Free slot: position 0
0L
>>>
```

Indeed, if you try it, you'll find that the new entry appears first in the resulting dictionary:

```
>>> s['account'] = 1
>>> s
{'account': 1, 'price': 123.45, 'name': 'ACME', 'shares': 75, 'time':
   '9:45am'}
>>>
```

As dictionaries fill up, that collisions will occur and performance will degrade becomes increasingly more likely (for instance, notice that four different table positions were checked in the above example). Because of this, the size of the array used to hold the contents of a dictionary is increased by a factor of four whenever a dictionary becomes more than two-thirds full. This is a rather subtle implementation detail, but you can notice it if you carefully observe what happens if you add a sixth entry to the above dictionary:

```
>>> s
{'account': 1, 'price': 123.45, 'name': 'ACME', 'shares': 75, 'time':
   '9:45am'}
>>> s['date'] = '05/26/2013'
>>> s
{'account': 1, 'name': 'ACME', 'price': 123.45, 'shares': 75,
   'time':'9:45am', 'date': '05/26/2013'}
>>>
```

Notice how 'name' and 'price' swapped places when the next item was inserted. This is due to an expansion of the dictionary size from 8 to 32 entries and a recomputation of the hash table positions. In the new dictionary, the new positions for 'name' and 'price' are as follows:

```
>>> hash('name') & 31
5
>>> hash('price') & 31
10
>>>
```

To be fair, these kinds of details are not something that most programmers ever need to concern themselves with other than to realize that dictionaries involve some extra overhead both in computation and memory.

Digression: Dictionary Alternatives

If you're using dictionaries to store a lot of small data structures, it's probably worth noting that there are much more efficient alternatives available. For example, even a small dictionary has a memory footprint larger than you might expect:

```
>>> s = { 'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> import sys
>>> sys.getsizeof(s)
280
>>>
```

Here you see that the dictionary is 280 bytes in size (actually, 296 bytes in Python 3.3). Keep in mind, that this size is just for the dictionary itself, not for the items stored inside. If this seems like a lot, you're right. The extra overhead can add up significantly if creating a large number of small data structures (e.g., imagine a program that's read a million line CSV file into a list of dictionaries representing each row).

Class instances are even more inefficient, adding an additional 64 bytes of overhead to the total size. In fact, a basic instance with no data at all requires 344 bytes of storage when one adds up all of the parts. For example:

```
>>> s = Stock('ACME', 100, 123.4)
>>> sys.getsizeof(s)
64
>>> sys.getsizeof(s.__dict__)
280
>>>
```

If you're working with data, there are some better choices. One such option is to create a named tuple:

```
>>> from collections import namedtuple
>>> Stock = namedtuple('Stock', ['name', 'shares', 'price'])
>>> s = Stock('ACME', 100, 123.45)
>>> s.name
'ACME'
>>> s.shares * s.price
12345.0
>>> sys.getsizeof(s)
80
>>>
```

A named tuple gives you the nice attribute access normally associated with a class and much more compact representation; however, as a tuple, the attributes are immutable. If you need mutability, consider defining a class with `__slots__` instead:

```
class Stock(object):
    __slots__ = ('name', 'shares', 'price')
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

This produces an even more compact representation:

```
>>> s = Stock('ACME', 100, 123.45)
>>> s.name
'ACME'
>>> s.shares = 75
>>> sys.getsizeof(s)
72
>>> hasattr(s, '__dict__')   # No underlying __dict__
False
>>>
```

The use of __slots__ on a class is actually the most compact representation of a data structure in Python without resorting to lower-level hacks such as binary encodings or C extensions. It's even smaller than using a tuple:

```
>>> s = ('ACME', 100, 123.45)
>>> sys.getsizeof(s)
80
>>>
```

Therefore, if you're working with a lot of data, and you're thinking about using dictionaries because of their programming convenience, consider some of these alternatives instead.

Randomized Key Ordering

In late 2011, a new kind of denial-of-service attack that exploited hash-table collisions was unveiled (see "Efficient Denial of Service Attacks on Web Application Platforms" at http://events.ccc.de/congress/2011/Fahrplan/events/4680.en.html). Without going into too many details, this attack involves sending carefully crafted requests to a Web server that push Python's hash-table collision handling algorithm into worst-case O(n**2) performance—the end result of which is that a clever hacker can make a server consume vast numbers of CPU cycles.

To combat this, Python now randomly salts the computation of hash values from run-to-run of the interpreter. This is something that is enabled by default in Python 3.3 or that can be enabled by the -R option to the interpreter in Python 2.7. For example:

```
bash % python -R
>>> s = {'name': 'ACME', 'shares':100, 'price':123.45 }
>>> s
{'shares': 100, 'name': 'ACME', 'price': 123.45}
>>>
```

```
bash-3.2$ python -R
>>> s = { 'name': 'ACME', 'shares':100, 'price':123.45 }
>>> s
{'name': 'ACME', 'shares': 100, 'price': 123.45}
>>>
```

The random salting makes it impractical for an attacker to construct requests that will work everywhere; however, the randomization can also cause funny things to happen in certain programs that use dictionaries.

In the case of PLY, randomness of dictionary order changed the numbering of states in a large automatically created state machine. This, in turn, caused a certain randomness in the ordering of output messages being checked by unit tests.

Although random ordering is harmless to the overall execution of the program, I had to fix a number of unit tests to take it into account. I also selectively introduced a few uses of OrderedDict instances (from the collections module) to force a predictable order on data structures of critical importance to the construction of state tables.

Split-Key Dictionaries

Python 3.3 introduces yet another improvement on dictionaries related to their use in class instances. In a class such as the Stock class presented earlier, observe that every instance is going to have exactly the same set of keys. Taking this observation into account, Python 3.3 dictionaries actually have two internal representations; a combined representation where keys and values are stored together and a split representation where the keys are only stored once and shared among many different dictionaries.

For instances, the more compact split representation is used. This is a bit hard to view directly, but here is a simple example that shows the impact on the memory footprint:

```
>>> s = Stock('ACME', 100, 123.45)
>>> sys.getsizeof(s)
64
>>> sys.getsizeof(s.__dict__)  # Note: Greatly reduced size
104
>>>
```

Indeed, if you try a further experiment in which you create one million identical instances, you'll find the total memory use to be about 169 MB. On the other hand, creating one million identical dictionaries requires almost 293 MB.

This change in implementation is interesting in that it now makes the use of a class a much better choice for storing data structures if you care about memory use. The only downside is that all benefits are lost if you perform any manipulation of

instances that add attributes outside of the __init__() method. For example:

```
>>> sys.getsizeof(s.__dict__)
104
>>> s.date = '5/27/2013'
>>> sys.getsizeof(s.__dict__)  # Flips to combined dictionary
296
>>>
```

Final Words

If there's any take-away from this article, it might be that parts of Python often assumed to be frozen in time are still a target of active development. Dictionaries are no exception. If you make the move to Python 3.3, you'll find that they are used in a much more efficient way than before (especially for instances).

This is by no means the last word. At this time, Raymond Hettinger, one of Python's core developers, has been experimenting with yet another dictionary representation which is even more memory efficient. Some details about this can be found at http://code.activestate.com/recipes/578375-proof-of-concept-for-a-more-space-efficient-faster/.

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Why Join USENIX?

We support members' professional and technical development through many ongoing activities, including:

- Open access to research presented at our events
- Workshops on hot topics
- Conferences presenting the latest in research and practice
- LISA: The USENIX Special Interest Group for Sysadmins
- *;login:*, the magazine of USENIX
- Student outreach

Your membership dollars go towards programs including:

- Open access policy: All conference papers and videos are immediately free to everyone upon publication
- Student program, including grants for conference attendance
- Good Works program

Helping our many communities share, develop, and adopt ground-breaking ideas in advanced technology

Join us at www.usenix.org