CHAPTER EIGHTEEN

Python's Dictionary Implementation: Being All Things to All People

Andrew Kuchling

Dictionaries are a fundamental data type in the Python programming language. Like *awk's* associative arrays and Perl's hashes, dictionaries store a mapping of unique keys to values. Basic operations on a dictionary include:

- Adding a new key/value pair
- Retrieving the value corresponding to a particular key
- Removing existing pairs
- Looping over the keys, values, or key/value pairs

Here's a brief example of using a dictionary at the Python interpreter prompt. (To try out this example, you can just run the *python* command on Mac OS and most Linux distributions. If Python isn't already installed, you can download it from *http://www.python.org*.)

In the following interactive session, the >>> signs represent the Python interpreter's prompts, and `d` is the name of the dictionary I'm playing with:

```
>>> d = {1: 'January', 2: 'February',
...   'jan': 1, 'feb': 2, 'mar': 3}
```

```
{'jan': 1, 1: 'January', 2: 'February', 'mar': 3, 'feb': 2}
>>> d['jan'], d[1]
(1, 'January')
>>> d[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 12
>>> del d[2]
>>> for k, v in d.items(): print k,v   # Looping over all pairs.
jan 1
1 January
mar 3
feb 2
...
```

Two things to note about Python's dictionary type are:

- A single dictionary can contain keys and values of several different data types. It's legal to store the keys `1`, `3+4j` (a complex number), and `"abc"` (a string) in the same dictionary. Values retain their type; they aren't all converted to strings.
- Keys are not ordered. Methods such as `.values()` that return the entire contents of a dictionary will return the data in some arbitrary arrangement, not ordered by value or by insertion time.

It's important that retrieval of keys be a very fast operation, so dictionary-like types are usually implemented as hash tables. For the C implementation of Python (henceforth referred to as CPython), dictionaries are even more pivotal because they underpin several other language features. For example, classes and class instances use a dictionary to store their attributes:

```
>>> obj = MyClass()           # Create a class instance
>>> obj.name = 'object'       # Add a .name attribute
>>> obj.id = 14               # Add a .id attribute
>>> obj.__dict__              # Retrieve the underlying dictionary
{'name': 'object', 'id': 14}
>>> obj.__dict__['id'] = 12   # Store a new value in the dictionary
>>> obj.id                    # Attribute is changed accordingly
12
```

Module contents are also represented as a dictionary, most notably the `__builtin__` module that contains built-in identifiers such as `int` and `open`. Any expression that uses such built-ins will therefore result in a few dictionary lookups. Another use of dictionaries is to pass keyword arguments to a function, so a dictionary could potentially be created and destroyed on every function call. This internal use of the dictionary type means that any running Python program has many dictionaries active at the same time, even if the user's program code doesn't explicitly use a dictionary. It's therefore important that dictionaries can be created and destroyed quickly and not use an overly large amount of memory.

The implementation of dictionaries in Python teaches several lessons about performance-critical code. First, one has to trade off the advantages of an optimization against the

overhead it adds in space or calculation time. There were places where the Python developers found that a relatively naïve implementation was better in the long run than an extra optimization that seemed more appealing at first. In short, it often pays to keep things simple.

Second, real-life benchmarking is critical; only that way can you discover what's really worth doing.

Inside the Dictionary

Dictionaries are represented by a C structure, `PyDictObject`, defined in *Include/dictobject.h*. Here's a schematic of the structure representing a small dictionary mapping `"aa"`, `"bb"`, `"cc"`, ..., `"mm"` to the integers 1 to 13:

```
int ma_fill         13
int ma_used         13
int ma_mask         31

PyDictEntry ma_table[]:
[0]: aa, 1                  hash(aa) == -1549758592, -1549758592 & 31 = 0
[1]: ii, 9                  hash(ii) == -1500461680, -1500461680 & 31 = 16
[2]: null, null
[3]: null, null
[4]: null, null
[5]: jj, 10                 hash(jj) == 653184214, 653184214 & 31 = 22
[6]: bb, 2                  hash(bb) == 603887302, 603887302 & 31 = 6
[7]: null, null
[8]: cc, 3                  hash(cc) == -1537434360, -1537434360 & 31 = 8
[9]: null, null
[10]: dd, 4                 hash(dd) == 616211530, 616211530 & 31 = 10
[11]: null, null
[12]: null, null
[13]: null, null
[14]: null, null
[15]: null, null
[16]: gg, 7                 hash(gg) == -1512785904, -1512785904 & 31 = 16
[17]: ee, 5                 hash(ee) == -1525110136, -1525110136 & 31 = 8
[18]: hh, 8                 hash(hh) == 640859986, 640859986 & 31 = 18
[19]: null, null
[20]: null, null
[21]: kk, 11                hash(kk) == -1488137240, -1488137240 & 31 = 8
[22]: ff, 6                 hash(ff) == 628535766, 628535766 & 31 = 22
[23]: null, null
[24]: null, null
[25]: null, null
[26]: null, null
[27]: null, null
[28]: null, null
[29]: ll, 12                hash(ll) == 665508394, 665508394 & 31 = 10
[30]: mm, 13                hash(mm) == -1475813016, -1475813016 & 31 = 8
[31]: null, null
```

The ma_ prefix in the field names comes from the word *mapping*, Python's term for data types that provide key/value lookups. The fields in the structure are:

`ma_used`
: Number of slots occupied by keys (in this case, 13).

`ma_fill`
: Number of slots occupied by keys or by dummy entries (also 13).

`ma_mask`
: Bitmask representing the size of the hash table. The hash table contains `ma_mask+1` slots—in this case, 32. The number of slots in the table is always a power of 2, so this value is always of the form 2^n-1 for some n, and therefore consists of n set bits.

`ma_table`
: Pointer to an array of `PyDictEntry` structures. `PyDictEntry` contains pointers to:

  - The key object
  - The value object
  - A cached copy of the key's hash code

  The hash value is cached for the sake of speed. When searching for a key, the exact hash values can be quickly compared before performing a slower, full equality comparison of the keys. Resizing a dictionary also requires the hash value for each key, so caching the value saves having to rehash all the keys when resizing.

We don't keep track directly of the number of slots in the table, but derive it instead as needed from `ma_mask`. When looking up the entry for a key, `slot = hash & mask` is used to figure out the initial slot for a particular hash value. For instance, the hash function for the first entry generated a hash of –1549758592, and –1549758592 mod 31 is 0, so the entry is stored in slot 0.

Because the mask is needed so often, we store it instead of the number of slots. It's easy to calculate the number of slots by adding 1, and we never need to do so in the most speed-critical sections of code.

`ma_fill` and `ma_used` are updated as objects are added and deleted. `ma_used` is the number of keys present in the dictionary; adding a new key increases it by 1, and deleting a key decreases it by 1. To delete a key, we make the appropriate slot point to a dummy key; `ma_fill` therefore remains the same when a key is deleted, but may increase by 1 when a new key is added. (`ma_fill` is never decremented, but will be given a new value when a dictionary is resized.)

Special Accommodations

When trying to be all things to all people—a time- and memory-efficient data type for Python users, an internal data structure used as part of the interpreter's implementation, and a readable and maintainable code base for Python's developers—it's necessary to

complicate a pure, theoretically elegant implementation with special-case code for particular cases…but not too much.

A Special-Case Optimization for Small Hashes

The `PyDictObject` also contains space for an eight-slot hash table. Small dictionaries with five elements or fewer can be stored in this table, saving the time cost of an extra `malloc()` call. This also improves cache locality; for example, `PyDictObject` structures occupy 124 bytes of space when using x86 GCC and therefore can fit into two 64-byte cache lines. The dictionaries used for keyword arguments most commonly have one to three keys, so this optimization helps improve function-call performance.

When Special-Casing Is Worth the Overhead

As previously explained, a single dictionary can contain keys of several different data types. In most Python programs, the dictionaries underlying class instances and modules have only strings as keys. It's natural to wonder whether a specialized dictionary object that only accepted strings as keys might provide benefits. Perhaps a special-case data type would be useful and make the interpreter run faster?

The Java implementation: another special-case optimization

In fact, there *is* a string-specialized dictionary type in Jython (*http://www.jython.org*), an implementation of Python in Java. Jython has an `org.python.org.PyStringMap` class used only for dictionaries in which all keys are strings; it is used for the `__dict__` dictionary underpinning class instances and modules. Jython code that creates a dictionary for user code employs a different class, `org.python.core.PyDictionary`, a heavyweight object that uses a `java.util.Hashtable` to store its contents and does extra indirection to allow `PyDictionary` to be subclassed.

Python's language definition doesn't allow users to replace the internal `__dict__` dictionaries by a different data type, making the overhead of supporting subclassing unnecessary. For Jython, having a specialized string-only dictionary type makes sense.

The C implementation: selecting the storage function dynamically

CPython does *not* have a specialized dictionary type, as Jython does. Instead, it employs a different trick: an individual dictionary uses a string-only function until a search for non-string data is requested, and then a more general function is used. The implementation is simple. `PyDictObject` contains one field, `ma_lookup`, that's a pointer to the function used to look up keys:

```
struct PyDictObject {
    ...
    PyDictEntry *(*ma_lookup)(PyDictObject *mp, PyObject *key, long hash);
};
```

`PyObject` is the C structure that represents any Python data object, containing basic fields such as a reference count and a pointer to a type object. Specific types such as `PyIntObject` and `PyStringObject` extend the structure with additional fields as necessary. The dictionary implementation calls `(dict->ma_lookup)(dict, key, hash)` to find a key; `key` is a pointer to the `PyObject` representing the key, and `hash` is the hash value derived for the key.

`ma_lookup` is initially set to `lookdict_string`, a function that assumes that both the keys in the dictionary and the key being searched for are strings represented as Python's standard `PyStringObject` type. `lookdict_string` can therefore take a few shortcuts. One shortcut is that string-to-string comparisons never raise exceptions, so some unnecessary error checking can be skipped. Another is that there's no need to check for rich comparisons on the object; arbitrary Python data types can provide their own separate versions of `<`, `>`, `<=`, `>=`, `==`, and `!=`, but the standard string type has no such special cases.

If a nonstring key is encountered, either because it's used as a dictionary key or the program makes an attempt to search for it, the `ma_lookup` field is changed to point to the more general `lookdict` function. `lookdict_string` checks the type of its input and changes `ma_lookup` if necessary, then calls the chosen function to obtain a correct answer. (CPython trivia: this means that a dictionary with only string keys will become slightly slower if you issue `d.get(1)`, even though the search can't possibly succeed. All subsequent code in the program that refers to the dictionary will also go through the more general function and incur a slight slowdown.) Subclasses of `PyStringObject` have to be treated as nonstrings because the subclass might define a new equality test.

Collisions

For any hash table implementation, an important decision is what to do when two keys hash to the same slot. One approach is *chaining* (see *http://en.wikipedia.org/wiki/Hash_table#Chaining*): each slot is the head of a linked list containing all the items that hash to that slot. Python doesn't take this approach because creating linked lists would require allocating memory for each list item, and memory allocations are relatively slow operations. Following all the linked-list pointers would also probably reduce cache locality.

The alternative approach is *open addressing* (see *http://en.wikipedia.org/wiki/Hash_table#Open_addressing*): if the first slot `i` that is tried doesn't contain the key, other slots are tried in a fixed pattern. The simplest pattern is called *linear probing*: if slot `i` is full, try `i+1`, `i+2`, `i+3`, and so on, wrapping around to slot 0 when the end of the table is reached. Linear probing would be wasteful in Python because many programs use consecutive integers as keys, resulting in blocks of filled slots. Linear probing would frequently scan these blocks, resulting in poor performance. Instead, Python uses a more complicated pattern:

```
/* Starting slot */
slot = hash;

/* Initial perturbation value */
perturb = hash;
```

```
while (<slot is full> && <item in slot doesn't equal the key>) {
    slot = (5*slot) + 1 + perturb;
    perturb >>= 5;
}
```

In the C code, `5*slot` is written using bit shifts and addition as `(slot << 2) + slot`. The perturbation factor `perturb` starts out as the full hash code; its bits are then progressively shifted downward 5 bits at a time. This shift ensures that every bit in the hash code will affect the probed slot index fairly quickly. Eventually the perturbation factor becomes zero, and the pattern becomes simply `slot = (5*slot) + 1`. This eventually generates every integer between 0 and `ma_mask`, so the search is guaranteed to eventually find either the key (on a search operation) or an empty slot (on an insert operation).

The shift value of 5 bits was chosen by experiment; 5 bits minimized collisions slightly better than 4 or 6 bits, though the difference wasn't significant. Earlier versions of this code used more complicated operations such as multiplication or division, but though these versions had excellent collision statistics, the calculation ran slightly more slowly. (The extensive comments in *Objects/dictobject.c* discuss the history of this optimization in more detail.)

Resizing

The size of a dictionary's hash table needs to be adjusted as keys are added. The code aims to keep the table two-thirds full; if a dictionary is holding n keys, the table must have at least $n/(2/3)$ slots. This ratio is a trade-off: filling the table more densely results in more collisions when searching for a key, but uses less memory and therefore fits into cache better. Experiments have been tried where the 2/3 ratio is adjusted depending on the size of the dictionary, but they've shown poor results; every insert operation has to check whether the dictionary needed to be resized, and the complexity that the check adds to the insert operation slows things down.

Determining the New Table Size

When a dictionary needs to be resized, how should the new size be determined? For small- or medium-size dictionaries with 50,000 keys or fewer, the new size is `ma_used * 4`. Most Python programs that work with large dictionaries build up the dictionary in an initial phase of processing, and then look up individual keys or loop over the entire contents. Quadrupling the dictionary size like this keeps the dictionary sparse (the fill ratio starts out at 1/4) and reduces the number of resize operations performed during the build phase. Large dictionaries with more than 50,000 keys use `ma_used * 2` to avoid consuming too much memory for empty slots.

On deleting a key from a dictionary, the slot occupied by the key is changed to point to a dummy key, and the `ma_used` count is updated, but the number of full slots in the table isn't checked. This means dictionaries are never resized on deletion. If you build a large dictionary and then delete many keys from it, the dictionary's hash table may be larger than if you'd constructed the smaller dictionary directly. This usage pattern is quite

infrequent, though. Keys are almost never deleted from the many small dictionaries used for objects and for passing function arguments. Many Python programs will build a dictionary, work with it for a while, and then discard the whole dictionary. Therefore, very few Python programs will encounter high memory usage because of the no-resize-on-deletion policy.

A Memory Trade-Off That's Worth It: The Free List

Many dictionary instances are used by Python itself to hold the keyword arguments in function calls. These are therefore created very frequently and have a very short lifetime, being destroyed when the function returns. An effective optimization when facing a high creation rate and short lifetime is to recycle unused data structures, reducing the number of `malloc()` and `free()` calls.

Python therefore maintains a `free_dicts` array of dictionary structures no longer in use. In Python 2.5, this array is 80 elements long. When a new `PyDictObject` is required, a pointer is taken from `free_dicts` and the structure is reused. Dictionaries are added to the array when deletion is requested; if `free_dicts` is full, the structure is simply freed.

Iterations and Dynamic Changes

A common use case is looping through the contents of a dictionary. The `keys()`, `values()`, and `items()` methods return lists containing all of the keys, values, or key/value pairs in the dictionary. To conserve memory, the user can call the `iterkeys()`, `itervalues()`, and `iteritems()` methods instead; they return an iterator object that returns elements one by one. But when these iterators are used, Python has to forbid any statement that adds or deletes an entry in the dictionary during the loop.

This restriction turns out to be fairly easy to enforce. The iterator records the number of items in the dictionary when an `iter*()` method is first called. If the size changes, the iterator raises a `RuntimeError` exception with the message `dictionary changed size during iteration`.

One special case that modifies a dictionary while looping over it is code that assigns a new value for the same key:

```
for k, v in d.iteritems():
    d[k] = d[k] + 1
```

It's convenient to avoid raising a `RuntimeError` exception during such operations. Therefore, the C function that handles dictionary insertion, `PyDict_SetItem()`, guarantees not to resize the dictionary if it inserts a key that's already present. The `lookdict()` and `lookdict_string` search functions support this feature by the way they report failure (not finding the searched-for key): on failure, they return a pointer to the empty slot where the searched-for key would have been stored. This makes it easy for `PyDict_SetItem` to store the new value in the returned slot, which is either an empty slot or a slot known to be occupied by

the same key. When the new value is recorded in a slot already occupied by the same key, as in `d[k] = d[k] + 1`, the dictionary's size isn't checked for a possible resize operation, and the `RuntimeError` is avoided. Code such as the previous example therefore runs without an exception.

Conclusion

Despite the many features and options presented by Python dictionaries, and their widespread use internally, the CPython implementation is still mostly straightforward. The optimizations that have been done are largely algorithmic, and their effects on collision rates and on benchmarks have been tested experimentally where possible. To learn more about the dictionary implementation, the source code is your best guide. First, read the *Objects/dictnotes.txt* file at *http://svn.python.org/view/python/trunk/Objects/dictnotes.txt?view=markup* for a discussion of the common use cases for dictionaries and of various possible optimizations. (Not all the approaches described in the file are used in the current code.) Next, read the *Objects/dictobject.c* source file at *http://svn.python.org/view/python/trunk/Objects/dictobject.c?view=markup*.

You can get a good understanding of the issues by reading the comments and taking an occasional clarifying glance at the code.

Acknowledgments

Thanks to Raymond Hettinger for his comments on this chapter. Any errors are my own.