Pycon2017 CompactDict Talk Documentation

Release 1.0

Raymond Hettinger

Nov 14, 2017

CONTENTS

1 A confluence of a dozen great ideas 1

2 Raymond Hettinger 3

CHAPTER

ONE

A CONFLUENCE OF A DOZEN GREAT IDEAS

CHAPTER

TWO

RAYMOND HETTINGER

My Mission Train thousands of Python Programmers

Contact Info raymond dot hettinger at gmail dot com

Twitter Account @raymondh

Company: Mutable Minds, Inc.

Venue: US Pycon 2017 Portland, Oregon May 20, 2017

Contents:

2.1 Our Journey: The Beginning and the End

Python is built around dictionaries. The various namespaces include globals, locals, module dictionaries, class dictionaries, instance dictionaries.

Of these, instance dictionaries are among the most prolific.

2.1.1 Instance Dictionaries

Create a class to track user assignments with in a property category.

```
from __future__ import division, print_function
import sys

class UserProperty:
    def __init__(self, v0, v1, v2, v3, v4):
        self.guido = v0
        self.sarah = v1
        self.barry = v2
        self.rachel = v3
        self.tim = v4

    def __repr__(self):
        return 'UserProperty(%r, %r, %r, %r, %r)' \
               % (self.guido, self.sarah, self.barry, self.rachel, self.tim)

colors = UserProperty('blue', 'orange', 'green', 'yellow', 'red')
cities = UserProperty('austin', 'dallas', 'tuscon', 'reno', 'portland')
fruits = UserProperty('apple', 'banana', 'orange', 'pear', 'peach')

for user in [colors, cities, fruits]:
```

```
    print(vars(user))

print(list(map(sys.getsizeof, map(vars, [colors, cities, fruits]))))
```

2.1.2 Contents of the Instance Dictionaries

The three instance dictionaries:

```
{'guido': 'blue',
 'sarah': 'orange',
 'barry': 'green',
 'rachel': 'yellow',
 'tim': 'red'}

{'guido': 'austin',
 'sarah': 'dallas',
 'barry': 'tuscon',
 'rachel': 'reno',
 'tim': 'portland'}

{'guido': 'apple',
 'sarah': 'banana',
 'barry': 'orange',
 'rachel': 'pear',
 'tim': 'peach'}
```

2.1.3 Dictionary Size

Version	Dict Size	Dict Ordering	Notes
Python 2.7	280, 280, 280	['sarah', 'barry', 'rachel', 'tim', 'guido']	Scrambled
Python 3.5	196, 196, 196	dict_keys(['sarah', 'tim', 'rachel', 'barry', 'guido'])	Randomized
Python 3.6	112, 112, 112	dict_keys(['guido', 'sarah', 'barry', 'rachel', 'tim'])	Ordered

2.2 Evolution: A Dozen Good Ideas

In the beginning dinosaurs roamed the earth, there were databases:

```
Name       Color     City        Fruit
--------   --------  ---------   -------
'guido',   'blue',   'austin',   'apple'
'sarah',   'orange', 'dallas',   'banana'
'barry',   'green',  'tuscon',   'orange'
'rachel',  'yellow', 'reno',     'pear'
'tim',     'red',    'portland', 'peach'
```

Index into the database:

```
[None, 4, None, 1, None, None, 0, None, 2,
 None, 3, None, None, None, None, None]
```

2.2.1 Setup

Here is our sample data to store in our dictionaries.

```
from __future__ import division, print_function
from pprint import pprint

keys = 'guido sarah barry rachel tim'.split()
values1 = 'blue orange green yellow red'.split()
values2 = 'austin dallas tuscon reno portland'.split()
values3 = 'apple banana orange pear peach'.split()
hashes = list(map(abs, map(hash, keys)))
entries = list(zip(hashes, keys, values1))
comb_entries = list(zip(hashes, keys, values1, values2, values3))
```

2.2.2 How a Database Would Do It

The data is dense (no holes or over-allocations). And without an index, the search is linear.

```
def database_linear_search():
    pprint(list(zip(keys, values1, values2, values3)))
```

Structure:

```
[('guido', 'blue', 'austin', 'apple'),
 ('sarah', 'orange', 'dallas', 'banana'),
 ('barry', 'green', 'tuscon', 'orange'),
 ('rachel', 'yellow', 'reno', 'pear'),
 ('tim', 'red', 'portland', 'peach')]
```

2.2.3 How LISP Would Do It

Store lists of pairs.

```
def association_lists():
    pprint([
        list(zip(keys, values1)),
        list(zip(keys, values2)),
        list(zip(keys, values3)),
    ])
```

Structure:

```
[
 [('guido', 'blue'),
  ('sarah', 'orange'),
  ('barry', 'green'),
  ('rachel', 'yellow'),
  ('tim', 'red')],

 [('guido', 'austin'),
  ('sarah', 'dallas'),
  ('barry', 'tuscon'),
  ('rachel', 'reno'),
  ('tim', 'portland')],
```

```
 [('guido', 'apple'),
  ('sarah', 'banana'),
  ('barry', 'orange'),
  ('rachel', 'pear'),
  ('tim', 'peach')]
]
```

2.2.4 Separate Chaining

Use multiple buckets to reduce the linear search by a constant factor.

```
def separate_chaining(n):
    buckets = [[] for i in range(n)]
    for pair in entries:
        key, value = pair
        i = hash(key) % n
        buckets[i].append(pair)
    pprint(buckets)
```

Structure for `separate_chaining(2)`:

```
[[('guido', 'blue'), ('tim', 'red')],
 [('sarah', 'orange'), ('barry', 'green'), ('rachel', 'yellow')]]
```

Now, increase the number of buckets to minimize the load per bucket.

Structure for `separate_chaining(4)`:

```
[[('sarah', 'orange'), ('barry', 'green')],
 [('guido', 'blue')],
 [('tim', 'red')],
 ['rachel', 'yellow')]]
```

Further increase the number of buckets so that some buckets are empty and most of the others only have one entry per bucket.

Structure for `separate_chaining(8)`:

```
[[],
 [('barry', 'green')],
 [('sarah', 'orange')],
 [],
 [],
 [('guido', 'blue'), ('rachel', 'yellow')],
 [],
 [('tim', 'red')]]
```

2.2.5 Dynamic Resizing

With 8 buckets and 2000 entries, we would get linear searches of chains of length 250. The dictionary slows down as it gets bigger.

The solution is to periodically resize the dictionary so that it never more than two-thirds full:

```
def resize(self, n):
    items = self.items()                    # Save list of key/value pairs
    self.buckets = [[] for i in range(n)]   # Make a new, bigger table
    for key, value in items:                # Re-insert the saved pairs
        self[key] = value
```

2.2.6 Caching the Hash Value

Naïve resizing is expensive because the hash values would need to be recomputed for every key.

The solution is the store the full hash value in table so it can be used during resizing:

```
[[],
 [(873286367057653889, 'barry', 'green')],
 [(1395608851306079410, 'sarah', 'orange')],
 [],
 [],
 [(2612508993932319405, 'guido', 'blue'),
  (8886176438393677637, 'rachel', 'yellow')],
 [],
 [(38617469636359399, 'tim', 'red')]]
```

The resize code then becomes:

```
def faster_resize(self, n):
    new_buckets = [[] for i in range(n)]        # Make a new, bigger table
    for hashvalue, key, value in self.buckets:  # Re-insert the saved pairs
        bucket = new_bucket[hashvalue % n]      # Re-use cached hash value
        bucket.append((hashvalue, key, value))
```

You rarely see this innovation in textbooks because it isn't essential to the hash algorithm and because it eats additional space.

However, it makes resizes very cheap, about one-fifth as fast as a list copy!

It also makes the next innovation possible.

2.2.7 Faster Matching

When searching a bucket, we need to know whether the target key is found.We could test whether `key == target_key`, but that can be slow because any object can define a complex or interesting `__eq__()` method.

The solution is to have two fast early-outs that can bypass equality testing in some circumstances.

1) If two variables point to the same object, they are deemed equal. We say "identity implies equality".

2) For hash tables to work at all, they follow the invariant, "if two objects are equal, then their hash values as equal as well". We use the contra-positive, "if two objects have unequal hashes, then the objects must be unequal as well".

```
def fast_match(key, target_key):
    if key is target_key:             return True   # Fast
    if key.hash != target_key.hash:   return False  # Fast
    return key == target_key                        # Slow
```

This innovation is normally not seen in textbooks because it isn't essential to the core algorithm. In practice though, it dramatically speeds up equality testing.

The chance of an unnecessary equality test (where the hashes match and the keys are unequal) is 1 in 2**64.

2.2.8 Open Addressing

The problem with separate chaining is that a great deal of space is wasted by having pointers to many separate lists.

The solution is to make the table more dense and to cope with collisions using linear probing.

```
def open_addressing_linear(n):
    table = [None] * n
    for h, key, value in entries:
        i = h % n
        while table[i] is not None:
            i = (i + 1) % n
        table[i] = (key, value)
    pprint(table)
```

Structure for `open_addressing_linear(8)`:

```
[('tim', 'red'),              #     'tim' collided with 'sarah'
 None,
 None,
 ('guido', 'blue'),
 ('rachel', 'yellow'),
 None,
 ('barry', 'green'),
 ('sarah', 'orange')]
```

2.2.9 Deleted Entries

Open-addressing works great but makes it more challenging to delete keys.

The problem is that removing a key leaves a "hole" so that linear probing is unable to find keys that had collisions.

The solution is to mark the slot with as `DUMMY` entry to serve as a placeholder. During the key-insertion phase, we try to re-use that dummy entry whenever possible.

```
def lookup(h, key):
    freeslot = None                     # No dummy encountered yet
    for h, key, value in entries:
        i = h % n
        while True:
            entry = table[i]
            if entry == FREE:
                return entry if freeslot is None else freeslot
            elif entry == DUMMY:
                if freeslot is None:
                    freeslot = i        # Remember where the dummy is
            elif fast_match(key, entry.key):
                return entry
            i = (i + 1) % n
```

This lookup scheme is known as "Knuth – Algorithm D" and dates back to the late 1960s.

2.2.10 Multiple Hashing

The problem with linear probing is that we can end up with catastrophic linear pile-up.

The solution is re-hash to other locations based on both the full hash value (all 64 bits) and on the number of probes.

To make sure the probe sequence eventually visits every possible slot, we use a simple linear-congruential random number generator that provably eventually visits each slot, `i = 5 * i + 1`.

To make sure we use all the bits of the hash, we gradually shift in 5 bits at a time.

I believe Tim Peters originated these two ideas.

```
def open_addressing_multihash(n):
    table = [None] * n
    for h, key, value in entries:
        perturb = h
        i = h % n
        while table[i] is not None:
            print('%r collided with %r' % (key, table[i][0]))
            i = (5 * i + perturb + 1) % n
            perturb >>= 5
        table[i] = (h, key, value)
    pprint(table)
```

Structure for `open_addressing_multihash(8)`:

```
'barry' collided with 'guido'
'rachel' collided with 'guido'
'rachel' collided with 'barry'
'rachel' collided with 'guido'
'tim' collided with 'rachel'

[(2612508993932319405, 'guido', 'blue'),
 (873286367057653889, 'barry', 'green'),
 None,
 (1395608851306079410, 'sarah', 'orange'),
 None,
 (8886176438393677637, 'rachel', 'yellow'),
 None,
 (38617469636359399, 'tim', 'red')]
```

2.2.11 Early-Out For Lookups

The problem is that the core of Python relies heavily on dict lookups but may times that same lookup must be repeated on the off chance that the dictionary has mutated.

The solution was created by Victor Stinner in "PEP 509 – Add a private version to dict". The idea is to update an internal dict version number every time a dictionary is updated. That lets us do a fast version check to avoid slower repeated lookups.

2.2.12 Compact Dict

The problem is that dict tables have a lot of empty space internally for every entry which includes a hash, key, and value.

I designed a solution and proposed it to python-dev in 2012. The idea is to store the hash/key/value table densely and make a separate, tiny sparse table of indices that vector into the dense table.

```
def compact_and_ordered(n):
    table = [None] * n
    for pos, entry in enumerate(entries):
        h = perturb = entry[0]
        i = h % n
        while table[i] is not None:
            i = (5 * i + perturb + 1) % n
            perturb >>= 5
        table[i] = pos
    pprint(entries)
    pprint(table)
```

Structure for `compact_and_ordered(8)`:

```
[(6364898718648353932, 'guido', 'blue'),
 (8146850377148353162, 'sarah', 'orange'),
 (3730114606205358136, 'barry', 'green'),
 (5787227010730992086, 'rachel', 'yellow'),
 (4052556540843850702, 'tim', 'red')]

[2, None, 1, None, 0, 4, 3, None]
```

Note that the index row can be stored in *only 8 bytes!*:

```
def make_index(n):
    'New sequence of indices using the smallest possible datatype'
    if n <= 2**7: return array.array('b', [FREE]) * n       # signed char
    if n <= 2**15: return array.array('h', [FREE]) * n      # signed short
    if n <= 2**31: return array.array('l', [FREE]) * n      # signed long
    return [FREE] * n                                       # python integers
```

2.2.13 Key-Sharing Dict

The problem is with previous design is if you have several dictionaries with the same keys, then there is unnecessary repetition of the keys, hash values, and indices:

```
[(6364898718648353932, 'guido', 'blue'),
 (8146850377148353162, 'sarah', 'orange'),
 (3730114606205358136, 'barry', 'green'),
 (5787227010730992086, 'rachel', 'yellow'),
 (4052556540843850702, 'tim', 'red')]

 [None, None, 3, 4, 0, 2, 1, None]

[(6364898718648353932, 'guido', 'austin'),
 (8146850377148353162, 'sarah', 'dallas'),
 (3730114606205358136, 'barry', 'tuscon'),
 (5787227010730992086, 'rachel', 'reno'),
 (4052556540843850702, 'tim', 'portland')]

 [None, None, 3, 4, 0, 2, 1, None]

[(6364898718648353932, 'guido', 'apple'),
 (8146850377148353162, 'sarah', 'banana'),
```

```
 (3730114606205358136, 'barry', 'orange'),
 (5787227010730992086, 'rachel', 'pear'),
 (4052556540843850702, 'tim', 'peach')]

 [None, None, 3, 4, 0, 2, 1, None]

 ...
```

The solution was proposed by **Mark Shannon** in "PEP 412 – Key-Sharing Dictionary".

```
def shared_and_compact(n):
    'Compact, ordered, and shared'
    table = [None] * n
    for pos, entry in enumerate(comb_entries):
        h = perturb = entry[0]
        i = h % n
        while table[i] is not None:
            i = (5 * i + perturb + 1) % n
            perturb >>= 5
        table[i] = pos
    pprint(comb_entries)
    pprint(table)
```

Structure for `shared_and_compact(8)`:

```
[(6677572791034679612, 'guido', 'blue', 'austin', 'apple'),
 (473904286818950 70, 'sarah', 'orange', 'dallas', 'banana'),
 (2331978697662116749, 'barry', 'green', 'tuscon', 'orange'),
 (8526267319998534994, 'rachel', 'yellow', 'reno', 'pear'),
 (8496597955646384579, 'tim', 'red', 'portland', 'peach')]

[None, None, 3, 4, 0, 2, 1, None]
```

2.2.14 The Future Has Density and Great Sparsity

We can make the dict more sparse without moving any of the hash/key/value entries. The additional sparsity only costs 8 bytes and removes *all* hash collisions.

Structure for `shared_and_compact(16)`:

```
[(8950500660299631846, 'guido', 'blue', 'austin', 'apple'),
 (7019358351072014995, 'sarah', 'orange', 'dallas', 'banana'),
 (1995312852666664056, 'barry', 'green', 'tuscon', 'orange'),
 (4597548128032042170, 'rachel', 'yellow', 'reno', 'pear'),
 (4703852761116776113, 'tim', 'red', 'portland', 'peach')]

[None, 4, None, 1, None, None, 0, None, 2,
 None, 3, None, None, None, None, None]
```

Now, we have come full circle.

Conclusion: With all our progress on dictionaries, we've reinvented what was done with databases long ago.

2.2.15 Odds and Ends

- Sets use a different strategy, a mix of multiple chaining and linear probing.

- Cuckoo hashing is still possible with the current design though it is likely not going to be a win.
- SipHash is used for strings to prevent deliberate collisions.
- Internally, dict and sets have additional logic to guard against mutation while iterating.

2.3 Original Recipe for the Compact Dict

This is the code that launched the initiative in PyPy and seeded the first drafts for the patch for CPython.

At the time it was presented, the mood was opposed to dicts being ordered, so this recipe intentionally fills in deleted values with the last entry in the list.

The code runs under both Python 2 and Python 3.

2.3.1 Recipe

```
from __future__ import division, print_function
import array
import collections
import itertools

# Placeholder constants
FREE = -1
DUMMY = -2

class Dict(collections.MutableMapping):
    'Space efficient dictionary with fast iteration and cheap resizes.'

    @staticmethod
    def _gen_probes(hashvalue, mask):
        'Same sequence of probes used in the current dictionary design'
        PERTURB_SHIFT = 5
        if hashvalue < 0:
            hashvalue = -hashvalue
        i = hashvalue & mask
        yield i
        perturb = hashvalue
        while True:
            i = (5 * i + perturb + 1) & 0xFFFFFFFFFFFFFFFF
            yield i & mask
            perturb >>= PERTURB_SHIFT

    def _lookup(self, key, hashvalue):
        'Same lookup logic as currently used in real dicts'
        assert self.filled < len(self.indices)   # At least one open slot
        freeslot = None
        for i in self._gen_probes(hashvalue, len(self.indices)-1):
            index = self.indices[i]
            if index == FREE:
                return (FREE, i) if freeslot is None else (DUMMY, freeslot)
            elif index == DUMMY:
                if freeslot is None:
                    freeslot = i
            elif (self.keylist[index] is key or
                  self.hashlist[index] == hashvalue
```

```
                and self.keylist[index] == key):
                  return (index, i)

    @staticmethod
    def _make_index(n):
        'New sequence of indices using the smallest possible datatype'
        if n <= 2**7: return array.array('b', [FREE]) * n       # signed char
        if n <= 2**15: return array.array('h', [FREE]) * n      # signed short
        if n <= 2**31: return array.array('l', [FREE]) * n      # signed long
        return [FREE] * n                                       # python integers

    def _resize(self, n):
        '''Reindex the existing hash/key/value entries.
           Entries do not get moved, they only get new indices.
           No calls are made to hash() or __eq__().

        '''
        n = 2 ** n.bit_length()                     # round-up to power-of-two
        self.indices = self._make_index(n)
        for index, hashvalue in enumerate(self.hashlist):
            for i in Dict._gen_probes(hashvalue, n-1):
                if self.indices[i] == FREE:
                    break
            self.indices[i] = index
        self.filled = self.used

    def clear(self):
        self.indices = self._make_index(8)
        self.hashlist = []
        self.keylist = []
        self.valuelist = []
        self.used = 0
        self.filled = 0                                         # used + dummies

    def __getitem__(self, key):
        hashvalue = hash(key)
        index, i = self._lookup(key, hashvalue)
        if index < 0:
            raise KeyError(key)
        return self.valuelist[index]

    def __setitem__(self, key, value):
        hashvalue = hash(key)
        index, i = self._lookup(key, hashvalue)
        if index < 0:
            self.indices[i] = self.used
            self.hashlist.append(hashvalue)
            self.keylist.append(key)
            self.valuelist.append(value)
            self.used += 1
            if index == FREE:
                self.filled += 1
                if self.filled * 3 > len(self.indices) * 2:
                    self._resize(4 * len(self))
        else:
            self.valuelist[index] = value

    def __delitem__(self, key):
```

```
        hashvalue = hash(key)
        index, i = self._lookup(key, hashvalue)
        if index < 0:
            raise KeyError(key)
        self.indices[i] = DUMMY
        self.used -= 1
        # If needed, swap with the lastmost entry to avoid leaving a "hole"
        if index != self.used:
            lasthash = self.hashlist[-1]
            lastkey = self.keylist[-1]
            lastvalue = self.valuelist[-1]
            lastindex, j = self._lookup(lastkey, lasthash)
            assert lastindex >= 0 and i != j
            self.indices[j] = index
            self.hashlist[index] = lasthash
            self.keylist[index] = lastkey
            self.valuelist[index] = lastvalue
        # Remove the lastmost entry
        self.hashlist.pop()
        self.keylist.pop()
        self.valuelist.pop()

    def __init__(self, *args, **kwds):
        if not hasattr(self, 'keylist'):
            self.clear()
        self.update(*args, **kwds)

    def __len__(self):
        return self.used

    def __iter__(self):
        return iter(self.keylist)

    def iterkeys(self):
        return iter(self.keylist)

    def keys(self):
        return list(self.keylist)

    def itervalues(self):
        return iter(self.valuelist)

    def values(self):
        return list(self.valuelist)

    def iteritems(self):
        return itertools.izip(self.keylist, self.valuelist)

    def items(self):
        return zip(self.keylist, self.valuelist)

    def __contains__(self, key):
        index, i = self._lookup(key, hash(key))
        return index >= 0

    def get(self, key, default=None):
        index, i = self._lookup(key, hash(key))
        return self.valuelist[index] if index >= 0 else default
```

```
    def popitem(self):
        if not self.keylist:
            raise KeyError('popitem(): dictionary is empty')
        key = self.keylist[-1]
        value = self.valuelist[-1]
        del self[key]
        return key, value

    def __repr__(self):
        return 'Dict(%r)' % list(self.items())

    def show_structure(self):
        'Diagnostic method.  Not part of the API.'
        print('=' * 50)
        print(self)
        print('Indices:', self.indices)
        for i, row in enumerate(zip(self.hashlist, self.keylist, self.valuelist)):
            print(i, row)
        print('-' * 50)


if __name__ == '__main__':

    d = Dict([('timmy', 'red'), ('barry', 'green'), ('guido', 'blue')])
    d.show_structure()
```

2.3.2 Output from the dictionary

```
==================================================
Dict([('timmy', 'red'), ('barry', 'green'), ('guido', 'blue')])
Indices: array('b', [-1, 0, 1, -1, -1, 2, -1, -1])
0 (7590622361340690025, 'timmy', 'red')
1 (-1892432705862437594, 'barry', 'green')
2 (-7052256505348629874, 'guido', 'blue')
--------------------------------------------------
```