Python Data Structures implementation

list, dict: how does CPython actually implement them?

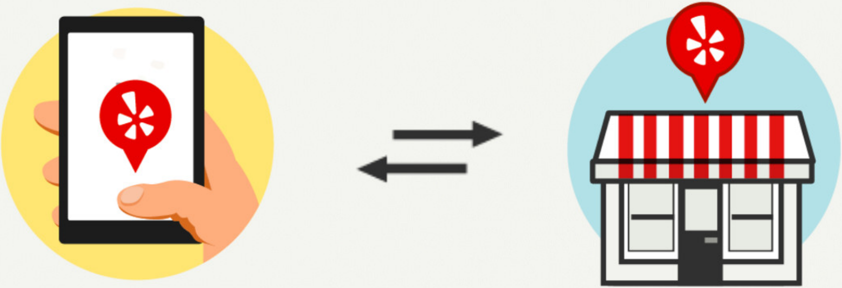
Flavien Raynaud

05 February 2017

FOSDEM 2017

Yelp's Mission

Connecting people with great local businesses.



list & dict

- To get started

```
>>> repeat('l.append(42)', 'l = [ ]')
2.1469707062351516e-07
>>> repeat('l.append(42)', 'l = [1]')
2.2675518121104688e-07
>>> repeat('l.append(42)', 'l = [i for i in range(1000)]')
2.475244084052974e-07
```

```
>>> repeat('d["year"] = 2017', 'd = {}')
1.5317109500756487e-07
>>> repeat('d["year"] = 2017', 'd = {"one": 1}')
1.5375611219496933e-07
>>> repeat('d["year"] = 2017', 'd = {str(i): i for i in range(1000)}')
2.46819700623746e-07
```

list & dict

- Focus on CPython 3.6
- A lot of hidden (and really cool) ideas
- A lot of lines of code (and comments)
  - ~3500 for lists
  - ~4500 for dicts
- (Almost) everyone knows (at least roughly) how they work

list

- A sequence of values (read: objects), 0-indexed
- $\mathcal{O}(1)$ amortized insert, $\mathcal{O}(1)$ random access, $\mathcal{O}(n)$ deletion

list

- A sequence of values (read: objects), 0-indexed
- $\mathcal{O}(1)$ amortized insert, $\mathcal{O}(1)$ random access, $\mathcal{O}(n)$ deletion
- Vector
  - Over-allocated array
  - Invariant: $0 \leq \mathit{len}(\mathit{list}) \leq \mathit{capacity}$

creating a new list

```
list()  # please avoid :-)
[]
[0, 1, 2, 3, 4]
list((0, 1, 2, 3, 4))
[i for i in range(5)]
```

- `[]`, `list()` → size = 0, capacity = 0
- `[0, 1, 2, 3, 4]` → size = 5, capacity = 5

appending to a list

```
categories = []
categories.append('food')
```

appending to a list

```
categories = []
categories.append('food')
```

What is really happening?

- `resize(size+1)`
- `set last value to 'food'`

resize

```
# resize the vector if necessary
resize(new_size):
 if capacity/2 <= new_size <= capacity:
  return

 capacity = (new_size / 8) + new_size
 capacity += (3 if new_size < 9 else 6)
 # realloc
```

resize

```
# resize the vector if necessary
resize(new_size):
 if capacity/2 <= new_size <= capacity:
  return

 capacity = (new_size / 8) + new_size
 capacity += (3 if new_size < 9 else 6)
 # realloc
```

- 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
- Growth rate: ~12.5%

append takeaway

```
>>> categories = [
  'food', 'tacos', 'bar', 'dentist', 'scuba diving'
]
>>> size_capacity(categories)
SizeCapacity(size=5, capacity=5)
```

append takeaway

```
>>> categories = [
  'food', 'tacos', 'bar', 'dentist', 'scuba diving'
]
>>> size_capacity(categories)
SizeCapacity(size=5, capacity=5)
```

```
>>> categories = []
>>> categories.append('food')
>>> categories.append('tacos')
>>> categories.append('bar')
>>> categories.append('dentist')
>>> categories.append('scuba diving')
>>> size_capacity(categories)
SizeCapacity(size=5, capacity=8)
```

removing from a list

```
categories = ['food', 'tacos', 'bar', 'dentist']
categories.pop()
categories.pop(i)
```

- `i == size - 1`
  - `resize(size - 1)`
- `i < size - 1`
  - `categories[i:]  = categories[i+1:]`
  - `memmove, resize(size - 1)`

removing from a list - i < size - 1

```
categories = ['food', 'tacos', 'bar', 'dentist']
categories.pop(1)  # no more tacos :-(
```

food	tacos	bar	dentist

size = 4, capacity = 4

removing from a list - i < size - 1

```
categories = ['food', 'tacos', 'bar', 'dentist']
categories.pop(1)  # no more tacos :-(
```

food	bar	dentist	dentist

$size = 4$, $capacity = 4$

removing from a list - i < size - 1

```python
categories = ['food', 'tacos', 'bar', 'dentist']
categories.pop(1)  # no more tacos :-(
```

food	bar	dentist	dentist

size = 4, capacity = 4

removing from a list - i < size - 1

```
categories = ['food', 'tacos', 'bar', 'dentist']
categories.pop(1)  # no more tacos :-(
```

food	bar	dentist	

size = 3, capacity = 4

list misc.

- `list` as a queue (`.append()`, `.pop(0)`) is bad → `deque`

list misc.

- `list` as a queue (`.append()`, `.pop(0)`) is bad → `deque`
- slicing is really powerful!

```
ints = [0, 1, 2, 3, 4]
ints[1:4] = [42] -> [0, 42, 4]
ints[1:1] = [42, 43] -> [0, 42, 43, 1, 2, 3, 4]
```

list misc.

- `list` as a queue (`.append()`, `.pop(0)`) is bad → `deque`
- slicing is really powerful!

```
ints = [0, 1, 2, 3, 4]
ints[1:4] = [42] -> [0, 42, 4]
ints[1:1] = [42, 43] -> [0, 42, 43, 1, 2, 3, 4]
```

- reference reuse scheme

```
>>> a, b, c = [0, 1], [2, 3], [4, 5]
>>> id(a), id(b), id(c)
(140512822066120, 140512822065864, 140512822065928)
>>> del b
>>> d = [6, 7]
>>> id(d)
140512822065864
```

dict

- `dict` = dictionary
- Store (*key*, *value*) pairs

creating a new dict

```
dict()  # please avoid :-)
{}
{str(i): i for i in range(5)}
dict([('1', 1), ('2', 2)])
{
    'name': 'flavr',
    'nationality': 'french',
    'language': 'python',
    'age': 42,
}
```

dict usecases

- `kwargs`
  - ~1 write, ~1 read, small length

dict usecases

- `kwargs`
  - ~1 write, ~1 read, small length
- `class methods` (`MyClass.__dict__`)
  - ~1 write, many reads, any length but all share 8-16 elements

dict usecases

- `kwargs`
  - ~1 write, ~1 read, small length
- `class methods` (`MyClass.__dict__`)
  - ~1 write, many reads, any length but all share 8-16 elements
- `attributes, global vars` (`obj.__dict__`, `globals()`)
  - many writes, many reads, any length but often < 10

dict usecases

- `kwargs`
  - ~1 write, ~1 read, small length
- `class methods` (`MyClass.__dict__`)
  - ~1 write, many reads, any length but all share 8-16 elements
- `attributes, global vars` (`obj.__dict__`, `globals()`)
  - many writes, many reads, any length but often < 10
- `builtins` (`__builtins__.__dict__`)
  - ~0 writes, many reads, length ~150

dict usecases

- `kwargs`
  - ~1 write, ~1 read, small length
- `class methods` (`MyClass.__dict__`)
  - ~1 write, many reads, any length but all share 8-16 elements
- `attributes, global vars` (`obj.__dict__`, `globals()`)
  - many writes, many reads, any length but often < 10
- `builtins` (`__builtins__.__dict__`)
  - ~0 writes, many reads, length ~150
- `uniquification` (remove duplicates, counters)
  - many writes, ~1 read, any length

dict usecases

- `kwargs`
  - ~1 write, ~1 read, small length
- `class methods` (`MyClass.__dict__`)
  - ~1 write, many reads, any length but all share 8-16 elements
- `attributes, global vars` (`obj.__dict__`, `globals()`)
  - many writes, many reads, any length but often < 10
- `builtins` (`__builtins__.__dict__`)
  - ~0 writes, many reads, length ~150
- `uniquification` (remove duplicates, counters)
  - many writes, ~1 read, any length
- `other use`
  - any writes, any reads, any length, any deletions

dict history

- many implementation changes (2.1, 3.3, 3.6)
- `dict` in CPython 3.6
  - inspired from Pypy
  - ordered (`.keys()`, `.values()`, `.items()`)
  - memory-efficient (re-use keys when possible)
    - PEP412 - Key-Sharing dictionary
    - Split table, **Combined table**

dict

- $\mathcal{O}(1)$ average insert, $\mathcal{O}(1)$ average lookup, $\mathcal{O}(1)$ average deletion
- Fast access? Arrays
- `dict key` $\leftrightarrow$ `array index`

dict

- $\mathcal{O}(1)$ average insert, $\mathcal{O}(1)$ average lookup, $\mathcal{O}(1)$ average deletion
- Fast access? Arrays
- `dict key` $\leftrightarrow$ `array index`
- Hashing

hashing

Hash function: function used to map data from arbitrary size to data of (almost always) fixed size.

- CPython: $\{32{,}64\}$-bit integers

hashing

Hash function: function used to map data from arbitrary size to data of (almost always) fixed size.

- CPython: {32,64}-bit integers

```
>>> bits(hash(42))
'1000000000000000000000000000000000000000000000000000000000101010'
>>> bits(hash(1.61))
'1000000000000000000000000000000010111000111101100100001101110000'
>>> bits(hash("never gonna give..."))
'1000101010010000001111100110101110100001110011011101001100000001'
```

hashing

Hash function: function used to map data from arbitrary size to data of (almost always) fixed size.

- CPython: {32,64}-bit integers

```
>>> bits(hash(42))
'1000000000000000000000000000000000000000000000000000000000101010'
>>> bits(hash(1.61))
'1000000000000000000000000000000010111000111101100100001101110000'
>>> bits(hash("never gonna give..."))
'1000101010010000001111100110101110100001110011011101001100000001'
```

```
>>> bits(hash(-1))
'0111111111111111111111111111111111111111111111111111111111111110'
>>> bits(hash(-2))
'0111111111111111111111111111111111111111111111111111111111111110'
```

hashing

- Similar values often have dissimilar hashes

```
>>> bits(hash("hello"))
'0011010000111000011101000100111110100011110011111010100110100000'
>>> bits(hash("hallo"))
'1110100100100011001111111101011101100111100001100011110011101011'
```

hashing

- Similar values often have dissimilar hashes

```
>>> bits(hash("hello"))
'0011010000111000011101000100111110100011110011111010100110100000'
>>> bits(hash("hallo"))
'1110100100100011001111111101011101100111100001100011110011101011'
```

- Same value $\Rightarrow$ same hash

```
>>> bits(hash("hello"))
'0011010000111000011101000100111110100011110011111010100110100000'
>>> bits(hash("hello"))
'0011010000111000011101000100111110100011110011111010100110100000'
>>> bits(hash("hello"))
'0011010000111000011101000100111110100011110011111010100110100000'
```

dict

- `dict key` $\rightarrow$ `key hash` $\rightarrow$ `array index`

dict

- `dict key` $\rightarrow$ `key hash` $\rightarrow$ `array index`
- Can we actually represent a dict using arrays?

dict

- `dict key` → `key hash` → `array index`
- Can we actually represent a dict using arrays?
  - Yes.
  - `dict` = `2 arrays` (indices, entries)

dict as arrays

```python
# categories: dict(key=name, value=#businesses)
categories = {}
```

dict as arrays

```
# categories: dict(key=name, value=#businesses)
categories = {}
```

		entry index
0	000	
1	001	
2	010	
3	011	
4	100	
5	101	
6	110	
7	111	

	hash	key	value

- initial size = 8

dict & hash

- index of key `x`
  - last bits of `hash(x)`

```
>>> categories['food'] = 4000
>>> bits(hash('food'))
'0111100011101101110010011011100110110110110000001011001001001000'
>>> bits(hash('food'))[-3:]
'000'
```

dict & hash

- index of key `x`
  - last bits of `hash(x)`

```
>>> categories['food'] = 4000
>>> bits(hash('food'))
'0111100011101101110010011011100110110110110000001011001001001000'
>>> bits(hash('food'))[-3:]
'000'
```

		entry index
0	000	0
1	001	
2	010	
3	011	
4	100	
5	101	
6	110	
7	111	

	hash	key	value
0	01...000	'food'	4000

inserting in a dict

```
>>> categories['tacos'] = 31
>>> bits(hash('tacos'))[-3:]
'001'
>>> categories['bar'] = 127
>>> bits(hash('bar'))[-3:]
'101'
```

inserting in a dict

```
>>> categories['tacos'] = 31
>>> bits(hash('tacos'))[-3:]
'001'
>>> categories['bar'] = 127
>>> bits(hash('bar'))[-3:]
'101'
```

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	
6	110	
7	111	

	hash	key	value
0	01...000	'food'	4000
1	10...001	'tacos'	31

inserting in a dict

```
>>> categories['tacos'] = 31
>>> bits(hash('tacos'))[-3:]
'001'
>>> categories['bar'] = 127
>>> bits(hash('bar'))[-3:]
'101'
```

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	
7	111	

	hash	key	value
0	01...000	'food'	4000
1	10...001	'tacos'	31
2	00...101	'bar'	127

inserting in a dict

```
>>> categories['dentist'] = 17
>>> bits(hash('dentist'))[-3:]
'001'
```

inserting in a dict

```
>>> categories['dentist'] = 17
>>> bits(hash('dentist'))[-3:]
'001'
```

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	
7	111	

	hash	key	value
0	01...000	'food'	4000
1	10...001	'tacos'	31
2	00...101	'bar'	127

collision resolution

- Hash collision resolution: Open Addressing
  - $index = (5 * index + 1)\ \%\ size$

collision resolution

- Hash collision resolution: Open Addressing
  - $\mathit{index} = (5 * \mathit{index} + 1)\ \%\ \mathit{size}$
  - traverses each integer in $\{0, ..., \mathit{size} - 1\}$
  - (actually a bit more sophisticated)

inserting in a dict

- $index = 001_2 = 1$

inserting in a dict

- $index = 001_2 = 1$
- $index = (5 \times 1 + 1) \ \% \ 8 = 6 = 110_2$

inserting in a dict

- $index = 001_2 = 1$
- $index = (5 \times 1 + 1) \ \% \ 8 = 6 = 110_2$

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1	10...001	'tacos'	31
2	00...101	'bar'	127
3	11...001	'dentist'	17

lookup in a dict

```
>>> categories['food']
4000
>>> bits(hash('food'))[-3:]
'000'
```

lookup in a dict

```
>>> categories['food']
4000
>>> bits(hash('food'))[-3:]
'000'
```

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1	10...001	'tacos'	31
2	00...101	'bar'	127
3	11...001	'dentist'	17

lookup in a dict

```
>>> categories['dentist']
17
>>> bits(hash('dentist'))[-3:]
'001'
```

lookup in a dict

```
>>> categories['dentist']
17
>>> bits(hash('dentist'))[-3:]
'001'
```

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1	10...001	'tacos'	31
2	00...101	'bar'	127
3	11...001	'dentist'	17

lookup in a dict

```
>>> categories['dentist']
17
>>> bits(hash('dentist'))[-3:]
'001'
```

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1	10...001	'tacos'	31
2	00...101	'bar'	127
3	11...001	'dentist'	17

lookup in a dict

```
>>> categories['music']
Traceback (most recent call last):
 ...
KeyError: 'music'
>>> bits(hash('music'))[-3:]
'000'
```

lookup in a dict

```
>>> categories['music']
Traceback (most recent call last):
  ...
KeyError: 'music'
>>> bits(hash('music'))[-3:]
'000'
```

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1	10...001	'tacos'	31
2	00...101	'bar'	127
3	11...001	'dentist'	17

lookup in a dict

```
>>> categories['music']
Traceback (most recent call last):
  ...
KeyError: 'music'
>>> bits(hash('music'))[-3:]
'000'
```

next_index('000') = '001'

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1	10...001	'tacos'	31
2	00...101	'bar'	127
3	11...001	'dentist'	17

lookup in a dict

```
>>> categories['music']
Traceback (most recent call last):
 ...
KeyError: 'music'
>>> bits(hash('music'))[-3:]
'000'
```

```
next_index('001') = '110'
```

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1	10...001	'tacos'	31
2	00...101	'bar'	127
3	11...001	'dentist'	17

lookup in a dict

```
>>> categories['music']
Traceback (most recent call last):
 ...
KeyError: 'music'
>>> bits(hash('music'))[-3:]
'000'
```

`next_index('110') = '111'`

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1	10...001	'tacos'	31
2	00...101	'bar'	127
3	11...001	'dentist'	17

deleting from a dict

```
>>> del categories['tacos']
>>> bits(hash('tacos'))[-3:]
'001'
```

deleting from a dict

```
>>> del categories['tacos']
>>> bits(hash('tacos'))[-3:]
'001'
```

		entry index
0	000	0
1	001	
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1			
2	00...101	'bar'	127
3	11...001	'dentist'	17

deleting from a dict

```
>>> del categories['tacos']
>>> bits(hash('tacos'))[-3:]
'001'
```

		entry index
0	000	0
1	001	
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1			
2	00...101	'bar'	127
3	11...001	'dentist'	17

- 'dentist' is not accessible anymore!

deleting from a dict

```
>>> del categories['tacos']
>>> bits(hash('tacos'))[-3:]
'001'
```

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1		<dummy>	
2	00...101	'bar'	127
3	11...001	'dentist'	17

deleting from a dict

```
>>> del categories['tacos']
>>> bits(hash('tacos'))[-3:]
'001'
```

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1		<dummy>	
2	00...101	'bar'	127
3	11...001	'dentist'	17

- 'dentist' is still accessible!

caveats

- 8 slots is rarely enough
- full `indices` array $\rightarrow$ slower lookups
- $<$ *dummy* $>$ keys $\rightarrow$ even slower

resizing a dict

- invariant: at least one empty slot
- `usable` $= \frac{2}{3}$ `size` ($= 5$ initially)

resizing a dict

- invariant: at least one empty slot
- `usable` $= \frac{2}{3}$ `size` (= 5 initially)

		entry index
0	000	0
1	001	1
2	010	
3	011	
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1		<dummy>	
2	00...101	'bar'	127
3	11...001	'dentist'	17

- `len = 3`, `size = 8`, `usable = 1`

resizing a dict

```
>>> categories['dinner'] = 1024
>>> del categories['dentist']
```

resizing a dict

```
>>> categories['dinner'] = 1024
>>> del categories['dentist']
```

		entry index
0	000	0
1	001	1
2	010	
3	011	4
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1		<dummy>	
2	00...101	'bar'	127
3		<dummy>	
4	00...011	'dinner'	1024

- len = 3, size = 8, usable = 0

resizing a dict

```
>>> categories['vegan'] = 1024
```

resizing a dict

```
>>> categories['vegan'] = 1024
```

- `resize(2 * len + size / 2)`

```
resize(min_size):
    # to truncate hashes
    new_size = NEXT_POWER_OF_TWO(min_size)
    create_new_dict(new_size)
    for (hash, key, value) in entries:
        insert_new(hash, key, value)
    delete_old_dict()
```

- `len = 3`, `size = 8` $\rightarrow$ `min_size = 10`, `new_size = 16`

resizing a dict

```
>>> categories['vegan'] = 1024
```

- `resize(2 * len + size / 2)`

```
resize(min_size):
    # to truncate hashes
    new_size = NEXT_POWER_OF_TWO(min_size)
    create_new_dict(new_size)
    for (hash, key, value) in entries:
        insert_new(hash, key, value)
    delete_old_dict()
```

- `len = 3`, `size = 8` → `min_size = 10`, `new_size = 16`
- larger arrays → more items can fit
- larger arrays → more free slots → faster lookups
- no more < *dummy* > keys

dict misc.

- reference reuse scheme
- split table
  - 3 arrays (indices, entries, values)
  - share (indices, entries), own values

References/Resources:

- `github.com/flavray/fosdem-python-data-structures`
- CPython 3.6
- PEP412 - Key-Sharing Dictionary
- Faster, more memory efficient and more ordered dictionaries on PyPy
- The Mighty Dictionary (2010) - Brandon Craig Rhodes

special cases

```
>>> ints = [0, 1, 2, 3, 4]
>>> size_capacity(ints)
SizeCapacity(size=5, capacity=5)
```

```
>>> ints = [i for i in range(5)]
# Comprehensions act as for-loops, .append
>>> size_capacity(ints)
SizeCapacity(size=5, capacity=8)
```

ordering

```
>>> categories.keys(), categories.values()
(['food', 'bar', 'dinner'], [4000, 127, 1024])
```

		entry index
0	000	0
1	001	1
2	010	
3	011	4
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1		<dummy>	
2	00...101	'bar'	127
3		<dummy>	
4	00...011	'dinner'	1024

ordering

```
>>> categories.keys(), categories.values()
(['food', 'bar', 'dinner'], [4000, 127, 1024])
```

		entry index
0	000	0
1	001	1
2	010	
3	011	4
4	100	
5	101	2
6	110	3
7	111	

	hash	key	value
0	01...000	'food'	4000
1		<dummy>	
2	00...101	'bar'	127
3		<dummy>	
4	00...011	'dinner'	1024

\o/