# Core Python Containers

Under-the-hood

Raymond Hettinger

# Containers: Under-The-Hood

Q. What is a core python container?

A. lists, tuples, dicts, sets, deques

Q. Why look under the hood?

A. So you know what runs fast.

And, because it's cool.

# Where is the hood and what is under it?

http://svn.python.org/view/python/trunk

Include/listobject.h

http://tinyurl.com/2a65l3

Objects/listobject.c

http://tinyurl.com/2e5fjo

# List Implementation

- Fixed-length array of pointers

- When the array grows or shrinks,

  calls realloc()

  and, if necessary, copies all of

  the items to the new space

# How expensive is an allocator call?

YMMV.

- Some memory allocators are better than others.
- Good ones have cheap calls, bad ones don't.
- Good ones layout memory strategically to minimize data copies.
- Other allocators fragment like crazy.

# Python assumes the worst

To minimize reallocs() and memcpy(),
we adopt an overallocation strategy

IOW, we leave a little room to grow.

# Overallocation Example

```
>>> s = []
>>> for c in string.letters:
...         s.append(c)
[]                              # 0 items takes 0 space
[A . . .]                       # 1 item takes 4 spaces
[A B . .]                       # Second append() is free!
[A B C .]                       # So is the third.
[A B C D]                       # And the fourth.
[A B C D E . . .]               # Fifth item costs a realloc()
[A B C D E F . .]               # Sixth is free!
```

- If we're lucky, the allocator can just extend.
- If unlucky, the data gets copied to a new, larger array.

# Overallocation Details

Growth pattern is 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...

For larger values, never more than 12 1/2% overallocated.

Result is amortized O(1) cost of an append.  Nice!

Lots of short 1 or 2 item lists uses a lot of space.

# Q. What if the array shrinks?

A. Reallocs when size goes below half of the allocated space.

So, `list.pop()` is *very* cheap.

Very few calls to the memory allocator.

Even then, there tends to be no data copy.

# Q. What if array grows or shrinks in the middle?

A. Realloc still depends on total length.

BUT!

All the trailing elements have to be copied ☹

```
list.insert(n, item)    # O(n) operation
list.pop(n)             # O(n) operation
del list[n]             # O(n) operation
```

# Insertion Example

>>> s = [A B C D E F G H]

>>> s.insert(3, X)

```
[A B C . D E F G H]     # shift trailing elements
[A B C X D E F G H]     # add new element
```

Inserting at the third position, entails moving five other pointers (to D E F G H).

# Q. Is inserting and deleting element-0 expensive?

A. Yes!

Q. Well, what to do?

A. Use collections.deque() which is optimized for appends and pops at BOTH ends. Though, it is slower for indexed accesses like s[n]=1, etc.

# Summary

- Lists implemented as fixed length arrays
- Cost of resizing varies across builds
- Python over-allocates to save re-sizes.
- Larger lists never more than 12 1/2% overallocated.
- list.append() and list.pop() are O(1)
- list.insert(n,x) and list.pop(n) are O(n)
- deque() is fast at both ends but not in middle.

# Q. Anything else about lists?

A. lists of known length get pre-sized exactly.

- `s = range(n)` allocates EXACTLY n spaces.
- No wasted space.
- No resizing as it gets filled.

- Some functions like map() and list() pre-size exactly.
- `[None] * n` will pre-size.
- So will most slicing operations.

- That's nice.

# Set Implementation

- Fixed-length hash table.
- Entries have two elements:
  1. object
  2. its hash value
- Smallest size is 8.
- When 2/3 full, grows by factor of four.

# How fast is set.add()?

O(1)

Most of the time, there is no resize or data movement.

Once in a while, the size quadruples and the entries are re-inserted.

# Anything else about sets?

Yes. Sets remember the hash value for each object.

- Many potentially expensive equality tests can be saved.
- We make a cheap check for match on identity.
- We make a cheap check for hash mismatch.
- Only then, will an equality test happen.

```
def match(x, elem):
    if x is elem:                return True
    if x.hash != elem.hash:      return False
    return x == elem
```

# Other uses for stored hash values?

Yes.

Set-to-set operations and set-to-dict operations already know ALL of the relevant hash values so they NEVER need to call __hash__().

```
s.copy()          # no calls to __hash__
s & t             # no calls to __hash__
d.fromkeys(s)     # no calls to __hash__
```

These are very cheap!

Many times faster than creating the input dicts or sets.

About a fifth as fast a list copy!

# Moral of the story

Put data in sets or dicts just once.
Subsequent manipulations a very cheap.

Slow:

```
same = set(dataone) & set(datatwo)
both = set(dataone) | set(datatwo)
diff = set(dataone) - set(datatwo)
```

Faster:

```
d1, d2 = set(dataone), set(datatwo)
same = d1 & d2
both = d1 | d2
diff = d1 - d2
```

Q. Does this mean that sets guarantee to never unnecessarily call __hash__()?

A. Yes

# What you know about sets

- Sets are hash tables with sizes 8, 32, 128, 512, etc.
- Hash tables don't get more than 2/3 full.
- Sets searches average no more than 1.5 probes
- Each entry has an object and its hash value
- Insertion and deletion are O(1) operations
- Fast identity and hash checks save needless __eq__() calls.
- Set-to-set operations are about as fast a list copy.
- Set-to-dict operations are cheap too.
- Building sets is much more expensive than using them.
- So, build them once and them manipulate them cheaply.
- Set operations never call __hash__() needlessly.

# What about dictionaries?

Yawn.

Dicts are the same as sets but the hash tables store (hash, key, value)

Q. So, the performance and algorithms are the same as sets?

A. Yes.

# Anything else about dicts?

Yes, they are the most finely tuned data structure in the language.

Use them fearlessly and often.

Tim Peters: "Code written with Python dictionaries is a gazillion times faster than C"

Raymond: "If you need a mapping but try something else, it will be dog slow no matter what language you use."

# Deque Implementaion

- Many implementations possible
- This one chosen for two reasons
  - Minimize calls to malloc/free
  - Exploit cache locality during iteration

    a cache hit cost ½ cycle

    a cache miss costs as much as a floating point divide
- Doubly linked blocks of 62 elements
- Block size chosen to fit well into a cache line
- Data written once and never moves

# Deque Performance

- Up to 10 blocks kept on a freelist

  Helps avoid calls to malloc/free

  Supports use case of continuous growth on one side while shrinking on the other

- Indexing optimizations
  - End-points d[0] and d[-1] are immediate
  - Scan-time = n/62 + 1
  - Knows whether to scan from right or left

# New deque feature in 2.6: maxlen

- d = deque(open(filename), maxlen=10)
- Remembers the last ten lines of a file
- When maxlen is reached, every append/ appendleft automatically pops from the other end.
- When combined with freelisting, it is effectively a circular queue! No allocator accesses ☺

# Named Tuples

- Implemented as a tuple subclass
- Zero space overhead
- Regular tuple methods run at tuple speed
- Adds attribute access
  - implemented as property(itemgetter(0))
  - runs at C speed (no trips around the eval-loop)
- Constructors add some pure python overhead
- That overhead can be eliminated when error checking is not needed (i.e. verifying the correct number of elements).

# Open for Questions