Python 103... MMMM:
Understanding Python's Memory Model, Mutability, Methods

Wesley J. Chun, Principal
CyberWeb Consulting
`@wescpy :: wescpy@gmail.com`
`http://cyberwebconsulting.com`

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

About the Speaker

- Python Experience
  - 14+ consecutive years of full-time Python development
  - Came to Python in 1997 (C, Unix, networking backgrnd)
- Programming, teaching, and writing for over 2 decades
- Book Author (or Co-author):
  - *Core Python Programming* ([2009,] 2007, 2001)
  - *Python Fundamentals* LiveLessons DVD (2009)
  - *Python Web Development with Django* (2009)
- Active Community Volunteer
  - User groups: BayPIGgies and SF Python Meetup
  - Python Tutor mailing list
  - PyCon Conference Organizer

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

I teach.

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Google

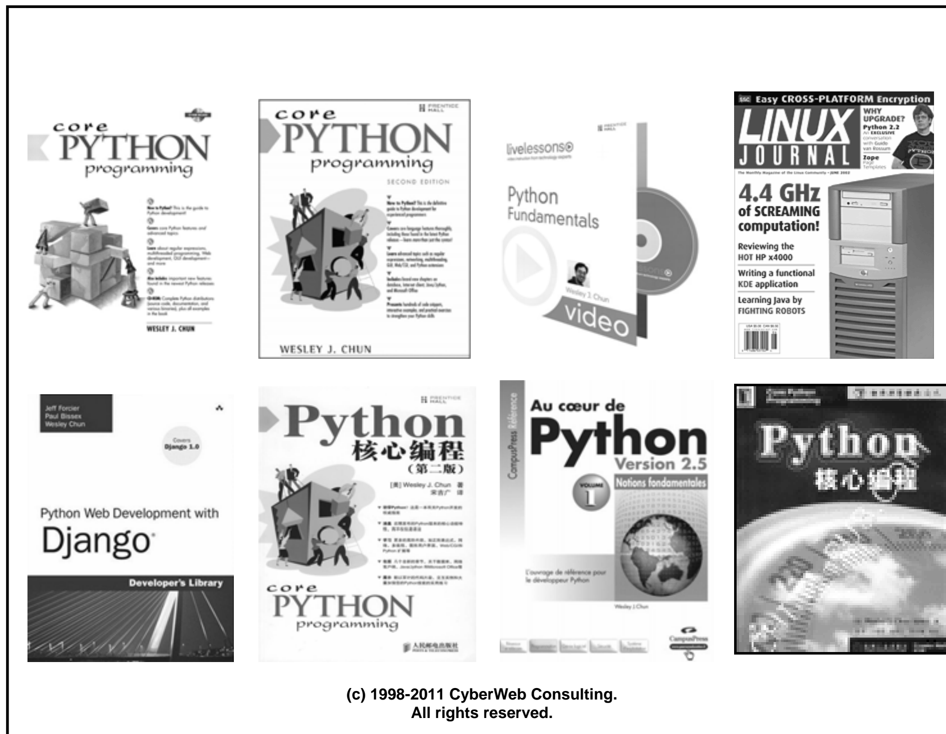
AVAYA

Foothill College
Upgrade. Advance.

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

I write.

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.



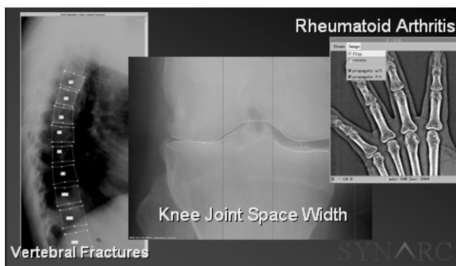
(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

I code.

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

slide

Google



NearbyNow

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

About this talk & you

- Several parts
  - Supercharge classes with special METHODS
  - Understanding Objects, References, & MEMORY Model
  - Groking your MUTABLE containers
  - Refactoring for performance, MEMORY, speed
- About you
  - Have some Python experience
  - Still don’t understand “weird” behavior
  - Want to learn more internals

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Part I

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Special Methods

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

How to Create a Class

- Class definition is code suite that follows header line
- Classes also support documentation strings
- New-style classes introduced in 2.2:

http://www.python.org/doc/newstyle/

```
class AddrBookEntry(object):
    "My very first class for a blackbook app"

    version = 0.1                          Data attribute

    def __init__(self, nm, ph):
        self.name = nm
        self.phone = ph                    Methods

    def updatePhone(self, newph):
        self.phone = newph
```

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Function/Method Evaluation

- "Call by reference" or "call by value"? Neither. Both.
- Behavior is based on whether object is mutable
  - Recall that objects are always passed by reference
  - For mutable objects, aliases behave like pointers... a change in one changes all, like call by reference
  - BUT for immutable objects, acts like call by value

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

More About Class Instances

- How to create an instance: call the class as if it were a function

```
john = AddrBookEntry('John Doe', '408-555-1212')
jane = AddrBookEntry('Jane Doe', '650-555-1212')
```

- Recall that instances have attributes too
  - `john.name, john.phone, jane.name, jane.phone`
  - Can have dynamic instance attributes: `john.tattoo='Mom'`
- `AddrBookEntry.version` is a class attribute

```
>>> print john.version      # use ABE.version
0.1
```

- But class attributes can be overridden, hiding the class attribute

```
>>> john.version = 1.0      # hides ABE.version
```

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

How to Create a Subclass

- Subclasses can be customized with their own attributes
- Subclass inherits all base class attributes unless overridden
- Base class initializer must be called manually if desired
  - Unless you did not override it; then it's called by default

```
class EmplAddrBookEntry(AddrBookEntry):
     'Employer Address Book'

     def __init__(self, nm, ph, id, ss):
       AddrBookEntry.__init__(self, nm, ph)
       self.empid = id
       self.ssn = ss
```

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Python Type Emulation

- Classes can have the same features as standard types
- Special class methods are used for this purpose
- By default they are defined empty (e.g. `pass`)
- Classes customized when you override/overwrite them
- Plenty of special class methods for this purpose

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Special Class Methods

Method Name	Description
`__init__()`	"Initializer," called upon instance initialization
`__new__()`	Like `__init__()` but for subclassing immutable objs
`__del__()`	"Terminator," called when instance's refcount == 0
`__str__()`	Overload `str()` for printable string representation
`__repr__()`	Overload `repr()` for evaluatable string representation
`__eq__(), __ne__()`	Overload `==` and `!=` symbol operations
`__add__()`	Overload `+` symbol for left-hand side operations
`__radd__()`	Overload `+` symbol for right-hand side operations
`__iadd__()`	Overload `+=` symbol operation
`__*sub__()`	Same as above but for `-` and `-=` operations
`__len__()`	Overload `len()` for sequence or mapping cardinality
`__getitem__()`	Overload `[]` to get sequence element/mapping value
`__setitem__()`	Overload `[]` to set sequence element/mapping value
`__getslice__()`	Overload `[:]` to get sequence slice
`__contains__()`	Overload `in` operator for membership testing

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Overriding Special Methods

- **time60.py**

```
class Time60(object):
    def __init__(self, hr, min): # what about "10:30"?
        self.hr = hr
        self.min = min

    def __str__(self):
        return '%d:%d' % (self.hr, self.min)

    def __add__(self, other):
        return self.__class__( __________, _________ )

>>> from time60 import Time60
>>> m = Time60(10,30)
>>> print m
10:30
>>> n = Time60(8,15) # 13-20(f) support sexagesimal oper
>>> print m + n
18:45
```

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

How to (not) use the Stack

And other performance considerations…

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Stack Utilization During Execution

- Stack pushes start from bottom, grows upward
- As frames are popped, stack shrinks downward
- 1 Python stack frame == 1 C stack frame*
- Memory problems occur when stack meets heap
- Misc: avoid global vars*, recursion fully supported, etc.

global variables

`foo2()`
arguments
`foo1()`
arguments
`"main()"`

grows

shrinks

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

The Stack and Execution Speed

- Which of the two loops is "faster?"

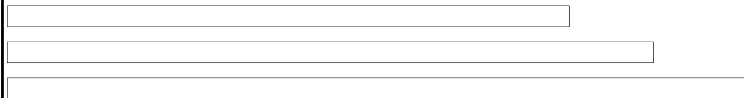
```
x = 'blah-blah ...heckuva long string...'

i = 0
while i < len(x):
    print x[i],
    i += 1
```

loop 1

```
i = 0
strlen = len(x)
while i < strlen:
    print x[i],
    i += 1
```

loop 2

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.



Objects & References

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Introduction to Python Objects

- **Objects are primary abstraction for data**
- **Attributes of all objects**
  - **Identity (similar to a memory address)**
  - **Type**
  - **Value**
- **All attributes are *read-only* except perhaps value**
- **Python has 30+ object types in all**
- `http://docs.python.org/ref/types.html`

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Objects

- **Standard Types**
  - Numbers (3-8)
  - Strings (2-3)
  - Lists
  - Tuples
  - Dictionaries
  - Sets (2)
- **Some Other Types**
  - `None`
  - Files
  - Functions/Methods
  - Modules
  - Types/Classes
  - Iterators/Generators

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Objects and References

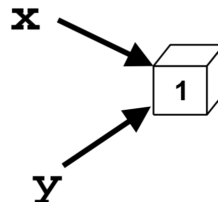
- Objects allocated on assignment
- All objects passed by reference
  - References are also called ***aliases***
  - ***Reference count*** used to track total number
  - Count in/decrements based upon usage
  - Objects garbage-collected when count goes to 0

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

More on References

- Variables don't "hold" data per se (not memory)
- Variables just point to objects (aliases)
- Additional aliases to an object can be created
- Objects reclaimed when "refcount" goes to 0
- Be aware of cyclic references

```
x = 1
y = x
```



(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Reference Count Increased

- Examples of refcount increment:
  - It (the object) is created (and assigned)
    ```
    foo = 'Python is cool!'
    ```
  - Additional aliases for it are created
    ```
    bar = foo
    ```
  - It is passed to a function (new local reference)
    ```
    spam(foo)
    ```
  - It becomes part of a container object
    ```
    lotsaFoos = [123, foo, 'xyz']
    ```

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Reference Count Decreased

- Examples of refcount decrement:
  - A local reference goes out-of-scope
    i.e., when `spam()` ends
  - Aliases for that object are explicitly destroyed
    ```
    del bar   # or del foo
    ```
  - An alias is reassigned a different object
    ```
    bar = 42
    ```
  - It is removed from a container object
    ```
    lotsaFoos.remove(foo)
    ```
  - The container itself is deallocated
    ```
    del lotsaFoos # or out-of-scope
    ```

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Categorizing the Standard Types

- **Why?**
  - **To make you learn them faster**
  - **To make you understand them better**
  - **To know how to view them internally**
  - **To encourage more proficient programming**

- **Three Models**
  - **Storage**
  - **Update**
  - **Access**

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Storage Model

- **How data is stored in an object**
- **Can it hold single or multiple objects?**

Model Category	Python Type
literal/scalar	numbers (all), strings
container	lists, tuples, dicts, sets

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Update Model

- Can an object's value be updated?
- Mutable == yes and immutable == no
- There is one of each set type
- `bytearray` type is mutable (3.x)

Model Category	Python Type
mutable	lists, dicts, sets
immutable	numbers, strings, tuples, frozensets

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Access Model

- How data is accessed in an object
  - Directly, via index, or by key
- Primary model for type differentiation

Model Category	Python Type
direct	numbers, sets
sequence	strings, lists, tuples
mapping	dicts

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Type Categorization Summary

Data Type	Storage Model	Update Model	Access Model
numbers	literal/scalar	immutable	direct
strings	literal/scalar	immutable	sequence
lists	container	mutable	sequence
tuples	container	immutable	sequence
dictionaries	container	mutable	mapping
sets	container	im/mutable	direct

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Objects & References Quiz

- **What is the output of the code below? WHY?**

Example 1

```
x = 42
y = x
x = x + 1
print x
print y
```

Example 2

```
x = [ 1, 2, 3 ]
y = x
x[0] = 4
print x
print y
```

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Quiz Answers

Example 1

```
>>> x = 42
>>> y = x
>>> x += 1
>>> print x
43
>>> print y
42
```

Example 2

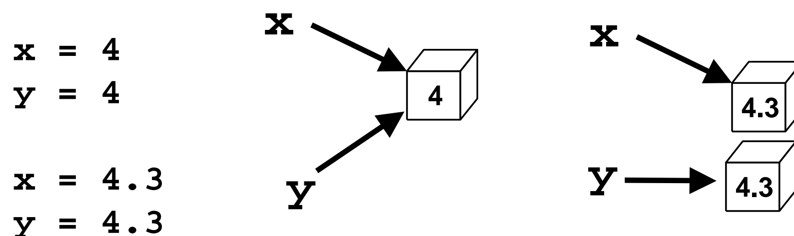
```
>>> x = [ 1, 2, 3 ]
>>> y = x
>>> x[0] = 4
>>> print x
[4, 2, 3]
>>> print y
[4, 2, 3]
```

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Interning of Objects

- Exception to the general rule
- Some strings and integers are "interned"
  - Integers in `range(-5, 257)` *[currently]*
  - Oft-used, single-character, and empty strings
- Primarily for performance reasons only

```
x = 4
y = 4

x = 4.3
y = 4.3
```



(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Objects and References Quiz 2

- Copy objects using their factory function(s)
  - Can use improper slice with sequences
- What is the output here (and WHY)?

```
x = ['foo', [1,2,3], 10.4]
y = list(x) # or x[:]
y[1][0] = 4
print x
print y
```

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Quiz 2 Answer

```
>>> x = ['foo', [1,2,3], 10.4]
>>> y = list(x) # or x[:]
>>> y[1][0] = 4
>>> print x
['foo', [4, 2, 3], 10.4]
>>> print y
['foo', [4, 2, 3], 10.4]
```

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Copying Objects

- Trickier with mutable objects
- Let's say you have a list `a` you wish to copy to `b`
  - Creating an alias not a copy
    `b = a   # a == b` and `a is b` *`[id(a) == id(b)]`*
  - Creating a shallow copy (all objects inside are aliases!)
    `b = a[:]  # a == b` but `a is not b`
  - Creating a deep copy (all objects inside are copies)
  - Use the `deepcopy()` function in the `copy` module
    `b = copy.deepcopy(a)`

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Part II

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Mutable Containers

- From Raymond Hettinger's PyCon 2008 talk
- Further illustrates value of understanding
- Create more efficient & correct code the first time!
- "Knowledge is Power"

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Lists

- Memory Allocation
- Growing & shrinking
  - Requires occasional call to `realloc()`
  - May or may not have to copy

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Some memory allocators better than others

- Good ones don't have much cost
- Bad ones are really bad here
- Good ones plan for growth/shrinkage
  - Strategically layout optimally
  - Slightly overallocate
  - Leave room to grow
  - Minimizes use of `realloc` () & `memcpy` ()
- Others slow and/or fragment

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Lists

- Fixed-length array of pointers
- Example appending one item at a time...
  - [] = no memory allocation
  - [X . . .] (1 item -> 4 units)
  - [X X . .] (2nd one free)
  - [X X X .] (so is the 3rd)
  - [X X X X] (and the 4th)
  - [X X X X X . . .] (5th `realloc` ()s)
  - [X X X X X X . .] (6th is free)
  - etc.
- Looks like a doubling w/each `realloc` ()

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

List Alloc Details

- Growth Pattern
  - 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, 106, 126,...
  - For larger values, don't go over 12.5% allocation
  - Performance: O(1) for `append ()`

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Growth Steps

```
>>> new_sz = 0
>>> while True:
    ...     print new_sz
    ...     new_sz = alloc_size(new_sz+1)
...
0
4
8
16
25
35
46
58
72
88
106
126
. . .
```

- What is `alloc_size ()`?

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Growing

- See `Objects/listobject.c`

```
alloc_size = lambda sz: (sz >> 3) +
  (3 if sz < 9 else 6) + sz
```

- Note: Lots of 1-2 (or 5) item lists waste space

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Shrinking

- `realloc ()` when size goes < 1/2 alloc'd space:

```
if allocd >= sz and sz >= (allocd >>
  1): return
```

- Means that removing is fairly inexpensive as...
  - `realloc ()` calls further minimized
- Performance: O(1) (both `append ()` & `pop ()`)

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

If at beginning (or middle)...

- Growing/shrinking still based on cardinality
- But we're talking about `realloc` () calls here
- Examples

```
[X X X X X X].insert(0, X)
del list[0]
```

  - etc.
- Bummer: O(n) (both `insert` (0) & `pop` (0))
- Requires copying/shifting everything

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Alternative to `[0]` Operations

- Use `collections.deque` class
  - Optimized for this purpose
  - Very fast for `append` () & `pop` () at the ends
- Non-circular doubly-linked list
- Slower for middle access via index though

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

List perf summary

- Lists: fixed-length arrays
  - Python slightly overallocates to avoid `realloc()`
- Grow/shrink at the end fast
  - `list.append()` & `list.pop()`: O(1)
- Grow/shrink in the beginning or middle
  - `list.insert(`n,x`)` & `list.pop(`n`)`: O(n)
- `collections.deque` fast at both ends but not middle

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

As cool as lists & listcomps are...

- May pay for memory you don't need
- Think about alternatives
  - Generator Expressions
  - Iterators
  - Generators
  - Sets

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Sets

- Based on fixed-length hash tables
- When sets get 2/3 full, they're grown by 4x
  - 0, 4, 32, 128, 512, ...
- Faster than DIY set class using dicts
- Insert/remove: O(1)
- Dictionaries
  - Operate like sets but store more data
  - Same perf and algorithms used

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Dicts & Sets Luxuries

- Building dicts & sets expensive
  - Faster to use them than make them
  - Like Italian sports cars
  - Very expensive to make but very fast
- Dicts most finely-tuned data struct in Python
- Python dict way faster...
  - Than avg map-like C object implementation

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Postmortem/Review of your solutions

- Find places to improve speed
- Find places to improve memory usage
- Refactor as necessary
- Bear in mind Python types & how they work

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Summary

- Special methods let you create std-type-like objects
- Be mindful of superfluous use of the stack
- Understand references & Python's memory model
- Keep in mind mutability may be causing your "bug"
- Python lists overallocate to avoid `realloc`()
  - Grow/shrink at the end fast: O(1)
  - Grow/shrink in the beginning or middle: O(n)
- collections.deque() fast at both ends but not middle
- Consider alternatives to lists & listcomps
- Dicts (+sets) most finely-tuned data structs in Python
- Building dicts & sets more expensive than using them

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.

Recent+Upcoming Events

- Oct 18-20: Intro+Inter. Python course, San Francisco
  - http://cyberwebconsulting.com
- Jul 25-29 O'Reilly Open Source (OSCON), Portland
  - http://oscon.com
- Jul 11-13 ACM CSTA CS&IT Conference, NYC
  - http://www.csitsymposium.org
- Jun 20-25 EuroPython, Florence
  - http://europython.eu
- May 8-10: Google I/O, San Francisco
  - http://google.com/io

**(c) 1998-2011 CyberWeb Consulting.
All rights reserved.**

FINIS

(c) 1998-2011 CyberWeb Consulting.
All rights reserved.