Totally Awesome Computing

Python as a General-Purpose Object-Oriented Programming Language

Chuck Allison
Associate Professor, Utah Valley State College
President, Fresh Sources, Inc.
chuck@freshsources.com
985 N 400 E
Springville, UT 84663

OOPSLA 2007, October 21-25, 2007, Montréal, Québec, Canada
ACM 07/0010

Table of Contents

Instructor Biography

Chuck Allison is an Associate Professor of Computer Science at Utah Valley State College in Orem, Utah, and president of Fresh Sources, Inc. a software training consulting company that specializes in C++, Python, and Design Patterns. He was a contributor to the 1998 C++ ISO standard and Senior Editor of the C/C++ Users Journal. He is founding and current editor of the C++ Source, the online journal for the C++ community, and a technical editor and columnist for Better Software Magazine. He is the author of two C++ books and over 90 articles.

Objectives of the Course

Attendees will:

- Understand the importance of Python in today's software development arena
- Understand the basic types and constructs of the language
- Become familiar with the programming paradigms supported by Python
- Be exposed to "Pythonic Programming" (idiomatic Python)
- Be prepared to begin software development in Python

Introduction

Python is well-known as a scripting language and is often used in server-side web programming. What many people don't know is that it is a full-powered object-oriented programming language, with some functional programming support thrown in for good measure. Python is a portable, modular, very-high-level language. Programs in Python are typically 3-10 times smaller in code size than their Java or C++ equivalents, and take a fraction of the time to develop. The Python library is among the most robust of any available—whatever you need, chances are there's a Python library for you. (As they say that the Python release comes with "batteries included".) Another lesser-known fact is that Python evolved from a language designed to teach programming to children, hence its easy learning curve and its elegant appearance. Python interfaces nicely with C++ and Java, so you can call upon their facilities in the rare case that the standard Python distribution doesn't already solve your problem. This is a soup-to-nuts tutorial for attendees already familiar with most any modern programming language.

Totally Awesome Computing

Python as a General-Purpose Object-Oriented Programming Language

OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada
ACM 07/0010

About Python

- Developed around 1990 by Guido van Rossum
  - Named after Monty Python
- A superb scripting language
- Also a general-purpose programming language
  - Fully object-oriented, multi-paradigm
- Can use with Java (Jython), C, C++ (Boost.Python), and .NET (IronPython)

Features

- Simple syntax
  - Very natural and easy to learn
  - A small number of rules applied consistently
- Incredibly powerful ("Batteries Included")
  - Useful built-in types and data structures
  - *Huge* library that supports…
    - Networking
    - XML and web applications
    - Mathematical computing
    - You name it!

Similarities to Other Languages

- Interpreted
  - Like Ruby, Perl, Lisp (and Java, C#, but no JIT)
- Garbage-collected (automatic memory mgt.)
  - Like those just mentioned and others
- Object-oriented
  - More than most languages (*everything* is an object)
- Supports Operator Overloading
  - Like C++, C#
- Supports Functional Programming (mostly)

Python on The Web

- Visit www.python.org
- Can download Python and many related items of interest (current version: 2.5.1)
- Documentation is there
  - Also Guido van Rossum's tutorial
  - And the library reference and module index
- Python's [Tutor] mail list: http://mail.python.org/mailman/listinfo/tutor
- Free online book: http://diveintopython.org/

Who Uses Python?

- Big Corporations:
  - NASA
  - NYSE
  - Industrial Light and Magic
  - Google
- And…
  - Yours Truly
  - Hopefully, you!

What is Python Used For?

- Education
- Web Programming
- Test Scripting
- Scientific Programming
- Game Development
- Much more…

Jedi Wisdom

Perl vs. Python (www.netfunny.com/rhf/jokes/99/Nov/perl.html)

YODA: Code! Yes. A programmer's strength flows from code maintainability. But beware of Perl. Terse syntax… more than one way to do it… default variables. The dark side of code maintainability are they. Easily they flow, quick to join you when code you write. If once you start down the dark path, forever will it dominate your destiny, consume you it will.

LUKE: Is Perl better than Python?

YODA: No… no… no. Quicker, easier, more seductive.

LUKE: But how will I know why Python is better than Perl?

YODA: You will know. When your code you try to read six months from now.

Today's Agenda

- The Nickel Tour
- Data Types: Sequences
- Data Types: Files and Dictionaries
- Functions and Functional Programming
- Classes and Objects

The Nickel Tour

The Nickel Tour – Topics

- Running Python
- Basic Data Types and Operations
- Useful Built-in Functions
- Basic String Handling
- Control Flow

Running Python

- Can run from the command line:
  - C:> python myfile.py
- Can run in the interpreter
  - >>> …
- Can run in an IDE
  - There are many
  - IDLE, PythonWin, Komodo, Stani's Python Editor

When Python Runs…

- The *interpreter* (python.exe) is the program that the O/S is actually running
- Your files (modules) are automatically loaded by and executed in the interpreter
- The first module loaded is the *main* module
  - __name__ = '__main__'
  - You load other modules via **import**

Example

```
# first.py
name = raw_input("Enter your first name: ")
age = input("Enter your age (we won't tell!): ")
print "So,", name, "you're",age
print type(name)
print type(age)
```


Demo of first.py

- Illustrate **import**
- print **__name__**
- Illustrate **from <module> import ???**

About Modules

- When a module is “loaded”, 2 things occur:
  - – The source code is compiled into an internal “byte code” understood by the interpreter
    - The byte code is contained in a module object and bound to a variable with the same name as the module’s file
  - – The byte code is also saved in a .pyc file for future loading (saves a little time)

import a Standard Module

```
# second.py
import math
print math
print dir(math)
print math.sqrt(2)
```

Output:
<module 'math' (built-in)>
['__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
1.41421356237

Variables and Bindings

- Python is a *dynamically typed* language
- Variables are not declared
  - – They are *bound* to values (objects) via assignment statements
  - – Must begin with a letter or underscore
  - – The variable ‘_’ yields the last *interactive* result
  - – Variables are “unbound” with **del**
- Variables can be rebound at will
  - – To an object of any type
  - – Previously bound objects that are no longer bound are eligible for garbage collection

Basic Data Types

(Mutable types in **boldface**)

- Numeric
  - int, long, float, complex, bool
- Sequences:
  - str, unicode, tuple, **list**, xrange
- “Hashed” Data Structures (fast lookup):
  - **dict**, **set**, frozenset
- Files
  - **file**
- None
  - Like `void` in C (sort of)

Numeric Operators

- The usual arithmetic ones:
  - +, -, *, /, <u>//</u>, <u>**</u>, %
- Bitwise operators:
  - |, ^, &, ~, >>, <<
- Comparisons (work for many types):
  - <, >, <=, >=, ==, !=

Precedence of Operators

- (…), […], {…}
- [n], [m:n:s]
- obj.attr
- f(…)
- +, -, ~ (unary)
- **
- *, /, //, %
- +, -
- <<, >>
- &
- ^
- |
- <, <=, >, >=, ==, !=, is, is not, in, not in
- not
- and
- or
- lambda

Useful Built-in Functions

- len
- min
- max
- sum
- Work on sequences and dictionaries
  - Not files

Useful Numeric Functions

- abs( )
  - abs(-42) == 42
- int( )
  - Truncates a real to an integer
  - int(2.5) == 2
- round( )
  - round (2.567) == 3.0
  - round(2.567, 2) == 2.5699999999999998
- pow(x, y) == x ** y
- float(x) (converts x to a real)

Applying Sequence Operations to Strings

```
>>> s = 'hello'
>>> s*4
'hellohellohellohello'
>>> t = "there"
>>> st = s + ', ' + t
>>> st
'hello, there'
>>> len(s)
5
```

Applying Sequence Operations to Strings (continued)

```
>>> s < t
True
>>> 'e' in s
True
>>> s[1]
'e'
>>> for c in s: print c
...
h
e
l
l
o
```


Slices

- The way to extract *substrings* by position
- Uses the syntax **s[start:endp1]**
- For example **"hello"[1:3]** == **"el"**
- The expression **s[p:]** extracts from position **p** to the end of the string
- The expression **s[:p]** extracts from the beginning up to *but not including* position **p**

"Negative" Slices

- Negative numbers in slices refer to positions based from the *end* of the string
  - The expression **s[-1]** extracts the last character
  - The expression **s[-2:]** extracts the last two characters
  - The expression **s[-3:-1]** extracts the two characters before the last (**s[-3]** and **s[-2]**)
  - (All are returned as strings: there is no character data type per se)

Envisioning Slices

- "Help"

```
+---+---+---+---+
| H | e | l | p |
+---+---+---+---+
  0   1   2   3   4 (outside)
 -4  -3  -2  -1
```

- The first (0-th) character is **s[-len(s)]**

Strides

- A third slice parameter defines a *stride*
  - The amount to skip between elements
- x[m:n:s] yields x[m], x[m+s], …, < x[n]
- If s > 0, n <= m yields an empty sequence
- If s < 0, m <= n yields an empty sequence

Slice "Quiz"

- Given a = 'hello':
- a[::2]
- a[::-2]
- a[:2:-1]
- a[5:0:-1]
- a[0:10:-1]
- a[-2:10]
- a[10:-2]

String Methods

- Called as <string-var>.<method>(…)
  - For example, **s.count('a')**
- They return a **bool** or a new string
  - Remember, strings are *immutable*
- capitalize, center, count, endswith, find, index, isalpha, isdigit (etc., as in C), istitle, join, lfind, ljust, lstrip, lower, replace, rfind, rjust, rstrip, split, startswith, strip, swapcase, title, translate, upper, zfill

Reversing a Sequence

- s[::-1]
  - That's it!
  - Returns a new sequence, of course
- Slice syntax:
  - [start:end+1:stride]
  - [end:start-1:-stride]
  - If stride is negative, you need **start > end** to get a non-empty sequence

Python Statements

- Assignments
- Control flow
  - if, while, for, break, continue, return, yield
- Exceptions
  - assert, try, raise
- Definitions
  - def (functions), class
- Namespaces
  - import, global
- Miscellaneous
  - pass, del, print, exec

Executing Strings as *Statements*

- The **exec** statement
  - Executes a string as if it were code in the current scope
  - **exec “a = 2”**
    - Binds **a** to 2 in the current scope
- The string can be computed or input, too!

Evaluating Strings as *Expressions*

- The **eval(str)** function
  - Evaluates the string as a Python expression and returns its value to the current context
- The string can be computed or input!

eval/exec Example

```
# eval_exec.py
x = input("Enter an integer value for x: ")
expression = raw_input("Enter an expression with x: ")
print eval(expression)
assign_y = raw_input("Enter an assignment for y: ")
exec assign_y
print y
```

```
C:\FreshSources\Symantec\Python07>python eval_exec.py
Enter an integer value for x: 10
Enter an expression with x: x*x+2
102
Enter an assignment statement for y: y = 'hello'
hello
```

The **if** statement

- Syntax:
  - **if** <condition-1>:
    <suite> (= indented group of statements)
    elif <condition-2>:
    <suite>
    …
    else
    <suite>
- Indentation is critical!
  - Unless you have a 1-liner

Conditional Expressions

- Like C's ternary operator (?:)
- **`x = y if z else w`**
  - Same as **`x = z ? y : w`**

Logical Connectives

- **and**
  - Both conditions must be **True**
- **or**
  - At least one condition must be **True**
- These are *short-circuited*
- **not**
  - Negates the truth value of the expression

Compound Boolean Idioms

- **if a < b < c** is equivalent to:
  if a < b and b < c
- What is the result of **a or b**?
  – For example, **2 or 3**
- What is the result of **a and b**?
- **[ ] and { }**?
- **3 or [ ]**?
- **A or B or C or None**?

Loops

- **while** <cond>:
  <body-1>
  else:
  <body-2> *# if loop completes*
- **for** <item> **in** <iterable>:
  <body-1>
  else:
  <body-2> *# ditto*
- Premature termination with **break**
- Skip to next iteration with **continue**

Sequences

Sequences – Topics

- Tuples
- Lists
- Slice Assignment

Python Sequences

- Strings
  - **str** and **unicode**
- Extended Range (**xrange**)
- Generic container sequences
  - **tuple**
  - **list**

Tuples

- An immutable collection of an arbitrary number of elements:
  - x = (1,'two')
- The empty tuple is ( )
- A tuple of length 1 requires a trailing comma:
  - y = (3,)
  - Because (3) is just 3

Tuple Example

```
>>> folks = ('bob',)
>>> folks += ('carol',)    # += creates a new tuple!
>>> folks('bob', 'carol')
>>> folks += ('ted', 'alice')
>>> folks('bob', 'carol', 'ted', 'alice')
>>> 2*folks
('bob', 'carol', 'ted', 'alice', 'bob', 'carol', 'ted',
'alice')
>>> theTuple = ((1,2), ((3,4,5),6), 7)
>>> theTuple[1]
((3, 4, 5), 6)
>>> theTuple[1][0]
(3, 4, 5)
>>> theTuple[1][0][:2]
(3, 4)
>>> tuple([1,2,3])
(1, 2, 3)
```


Tuple Assignment

```
>>> people = [('john','m', 42), ('jane', 'f', 38)]
>>> for name,gender,age in people:
...     print name, age
...
john 42
jane 38
>>> list(enumerate(['a','fine','mess']))
[(0, 'a'), (1, 'fine'), (2, 'mess')]
>>> for i,x in enumerate(['a', 'fine', 'mess']):
...     print i,x
...
0 a
1 fine
2 mess
```


Swap Idiom

x,y = y,x

Lists

- **list** is the only *mutable* sequence
- There are 9 list *methods*
  - All but two modify the list (**count** and **index**)
  - Only **count** and **pop** return a value

List Methods

• count(x)	• *Number of x's*
• index(x)	• *Where first x is*
• append(x)	• *Appends a new item*
• extend(L)	• *Same as +=*
• insert(i,x)	• *Inserts x at position i*
• remove(x)	• *Removes first x*
• pop(i = -1)	• *Removes at position*
• reverse()	• *Obvious!*
• sort(f = cmp)	• *Sorts in place*

Using List Methods

```
L=[1,2,2,3,3,3]
for n in L: print L.count(n),
1 2 2 3 3 3
L.index(2)
1
L.append(5)
L
[1, 2, 2, 3, 3, 3, 5]
L.extend([5,5,5,5])
L
[1, 2, 2, 3, 3, 3, 5, 5, 5, 5, 5]
for i in range(4): L.insert(6+i, 4)
L
[1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5]
```

Using List Methods
Continued

```
L.reverse()
L
[5, 5, 5, 5, 5, 4, 4, 4, 4, 3, 3, 3, 2, 2, 1]
L.sort()
L
[1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5]
L.remove(3)
L
[1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5]
L.pop()
5
L.pop(4)
3
L
[1, 2, 2, 3, 4, 4, 4, 4, 5, 5, 5, 5]
```


Slice Assignment with Lists

- You can replace a list slice (sublist) with elements from another sequence
  - Any sequence may appear on the right
- List may grow or shrink
- You *remove* a slice with [ ] on the right
  - You can also remove a slice with **del**
- You *insert* a sublist with an empty slice on the left

Using Slice Assignment

```
x = [1,2,3,4]
x[1:3] = [22,33,44]
x
[1, 22, 33, 44, 4]
x[1:4] = [2,3]
x
[1, 2, 3, 4]
x[1:1] = (100, 200) # Note tuple
x
[1, 100, 200, 2, 3, 4]
x[1:2] = []
x
[1, 200, 2, 3, 4]
del x[1:2]
x
[1, 2, 3, 4]
```

Slice Assignment with Strides

Non-contiguous Replacement

```
>>> x = [1,2,3,4]
>>> x[::2] = ['x','y']  # Same as x[0:4:2] = …
>>> x
['x', 2, 'y', 4]
>>> x[::2] = ['x','y', 'z']
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: attempt to assign sequence of size 3 to
extended slice of size 2
```


The Meaning of +=

- It's conceptually the same as + followed by replacement (via assignment)
  - But it's generally more efficient
- You can append *any type* of sequence to a list
  - But *not* to tuples or strings (types must match)
  - It appends each element of the sequence
  - It creates a new object with the other sequences
- See next slide

Using +=

```
x = []
x += (1,2,3)
x
[1, 2, 3]
x += "abc"
x
[1, 2, 3, 'a', 'b', 'c']
x += [4,(5,6),'seven']
x
[1, 2, 3, 'a', 'b', 'c', 4, (5, 6), 'seven']
>>> y = (1,2)
>>> id(y)
10362272
>>> y += (3,4)
>>> y
(1, 2, 3, 4)
>>> id(y)
10077120
```

List Comprehensions

- A powerful list-creation facility
- Builds a list from an expression

```
>>> x = [x*x for x in range(1,11)]
>>> x
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> [i for i in range(20) if i%2 == 0]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```


Nested List Comprehensions

```
>>> set1
'abc'
>>> set2
(1, 2, 3)
>>> cartesian = [(x,y) for x in set1 for y in set2]
>>> for pair in cartesian: print pair
...
('a', 1)
('a', 2)
('a', 3)
('b', 1)
('b', 2)
('b', 3)
('c', 1)
('c', 2)
('c', 3)
```


Nested List Comprehensions

with predicates

```
>>> set1
'abc'
>>> set2
(1, 2, 3)
>>> cartesian = [(x,y) for x in set1 for y in set2\
if x != 'b' if y < 3]
>>> for pair in cartesian: print pair
...
('a', 1)
('a', 2)
('c', 1)
('c', 2)
```

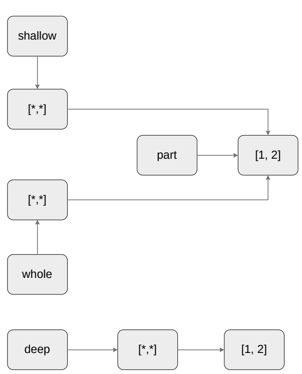
Copying Sequences

- Assigning one sequence to another makes no copy
  - A new variable referring to the original list is created and bound to the original list
  - The list's reference count is incremented
- If you want a copy, there are two types:
  - Use the [:] syntax   # shallow copy
  - copy.copy( )   # shallow copy
  - copy.deepcopy( )   # deep copy

Copying Lists

Example

```
>>> part = [1,2]
>>> whole = [part, part]
>>> shallow = whole[:]   # not shallow = whole
>>> deep = copy.deepcopy(whole)
>>> del part[0]
>>> part
[2]
>>> whole
[[2], [2]]
>>> shallow
[[2], [2]]
>>> deep
[[1, 2], [1, 2]]
>>> del deep[0][0]
>>> deep
[[2], [2]]
```

“is” vs. “==“

- “is” means two variables refer to the *same object*
- “==“ means that two variables or expressions refer to the *same value*:

```
>>> x = y = [1,2,3]
>>> z = [1,2,3]
>>> print id(x), id(y), id(z)
11100928 11100928 10383032
>>> x is y
True
>>> x is z
False
>>> x is not z
True
>>> x == z
True
```


Files and Dictionaries

Files and Dictionaries – Topics

- File Methods and Attributes
- Files as Iterables
- Dictionary Methods
- Dictionary Idioms
- Sets

File Methods

- open(<fname>, <mode> = ‘r’)
- file(<fname>) *(same as open(<fname>,'r')*
- read( ) *(whole file as a string)*
- read(n) *(n bytes as a string)*
- readline( ) *(read next line as a string)*
- readlines() *(all lines as a list of strings)*
- write(s) *(write a string)*
- writelines(L) *(write a list of strings)*
- close( )
- flush( )

File Example

```
>>> f = open('test.dat','w')
>>> f.write('This is line 1\n')
>>> lines = ['line 2 start', ' line 2 end\n', 'line 3\n']
>>> f.writelines(lines)
>>> f.close()
>>> lines = file('test.dat').readlines()
>>> lines
['This is line 1\n', 'line 2 start line 2 end\n', 'line
3\n']
>>> for line in lines: print line,
        ...
This is line 1
line 2 start line 2 end
line 3
```


File Open Modes

- r
- w
- a
- r+
- w+
- a+
- Can add the following:
  - b
  - U # Universal newline mode = \n; input only

File Attributes

- f.closed
- f.mode
- f.name

```
>>> f = file('first.py')
>>> f.name
'first.py'
>>> f.mode
'r'
>>> f.closed
False
>>> f.close()
>>> f.closed
True
```


Iterables

- Anything that can be “visited” in element order (aka “traversed”, “iterated over”)
- All **sequences** are iterable
- So are **files**, **generators**, **sets**, and **dictionaries**
  - You can make your own classes iterable!
- The **next()** method moves to the next item
  - That’s what **for** does under the hood

File Iteration

```
>>> for line in file('first.py'): print line,
...
# first.py
name = raw_input("Please enter your first name: ")
age = input("Please enter your age (we won't tell!): ")
print "So,", name, "you're",age
print type(name)
print type(age)
>>> list(file('first.py'))
['# first.py\n', 'name = raw_input("Please enter your first
name: ")\n', 'age = input("Please enter your age (we won\'t
tell!): ")\n', 'print "So,", name, "you\'re",age\n', 'print
type(name)\n', 'print type(age)\n']
>>> line1,line2,line3,line4,line5,line6 = file('first.py')
>>> line2
'name = raw_input("Please enter your first name: ")\n'
```

Explicit Iteration with **next()**

```
# iterate.py
f = file('first.py')
while True:
    try:
        print f.next(),
    except StopIteration:
        break
```

```
C:\FreshSources\Symantec\Python07>python iterate.py
# first.py
name = raw_input("Please enter your first name: ")
age = input("Please enter your age (we won't tell!): ")
print "So,", name, "you're",age
print type(name)
print type(age)
```


Iterator Helper Functions

- reversed(<iterable>)
  - Returns a reverse iterator for its argument
- sorted(<iterable>)
  - Returns the indicated sequence as a list
- See next slide

Helper Example

```
>>> x
['a', '1', 'b', '2']
>>> r = reversed(x)
>>> r
<listreverseiterator object at 0x009F3D70>
>>> for i in r: print i
...
2
b
1
a
>>> sorted(x)
['1', '2', 'a', 'b']
>>> sorted(('z','a'))
['a', 'z']
>>> sorted({'z':1,'a':2})
['a', 'z']
```

Dictionaries

- Dictionaries are unordered collections of pairs
  - (<immutable-key>, <value>)
  - They are stored for fast retrieval
- Duplicate keys are not allowed
  - You can change its associated value
  - You can remove its paired entry altogether
- Uses {…} or **dict()** for creation
- Can use key as an index with [ ]

Dictionary Methods

- has_key(x) (can also use "in")
- keys( )
- values( )
- items( ) (returns list of all pairs)
- copy( ) (shallow copy)
- d1.update(d2) (merge 2 dictionaries)
- get(key, def = None)
- setdefault( ) (= get( ) + create)
- pop(key [, def])
- popitem( ) Removes random pair
- clear( ) Removes all pairs
- Use **del** to remove elements by key

Dictionary Example

```
>>> d = {1:'a', 2:'b'}
>>> d[3] = 'c'
>>> d
{1: 'a', 2: 'b', 3: 'c'}
>>> d.has_key(2)
True
>>> d.has_key(4)
False
>>> 2 in d
True
>>> d.keys()
[1, 2, 3]
>>> d.values()
['a', 'b', 'c']
>>> d.items()
[(1, 'a'), (2, 'b'), (3,
'c')]
```

```
>>> d2 = {4:'d', 5:'e'}
>>> d.update(d2)
>>> d
{1: 'a', 2: 'b', 3: 'c',
4: 'd', 5: 'e'}
>>> d.get(1)
'a'
>>> d.get(10)
>>> d.get(10, 'j')
'j'
>>> d.setdefault(10, 'j')
'j'
>>> d
{1: 'a', 2: 'b', 3: 'c',
4: 'd', 5: 'e', 10: 'j'}
```

Dictionary Iterators

- **iteritems**
- **iterkeys**
  - Also returned by the dictionary name via **__iter__( )**
- **itervalues**

OOPSLA 2007 Tutorial 79 Chuck Allison

Dictionary Iterator Example

```
>>> d
{1: 'a', 2: 'b'}
>>> for key in d: print key   # or "in d.iterkeys()"
...
1
2
>>> for value in d.itervalues(): print value
...
a
b
>>> for pair in d.iteritems(): print pair
...
(1, 'a')
(2, 'b')
```

OOPSLA 2007 Tutorial 80 Chuck Allison

Dictionary Idioms

- To process keys and values together:
  for key in d: <use **key** and **d[key]**>
- To process keys and values sorted by key:
  keys = d.keys()
  keys.sort()
  for key in keys: <use **key** and **d[key]**>
- Or use **sorted(d.iteritems())**
  - Gives pairs in sorted key order
  - Python 2.4 and later only

OOPSLA 2007 Tutorial 81 Chuck Allison

Sets

- Based on mathematical sets
  - Unordered
  - No duplicate items allowed
  - Stored for fast search times
    - Like a dictionary with no values
- Two types:
  - **set** (mutable)
  - **frozenset** (immutable)
    - Must be initialized from an iterable

Operations for all Sets

• set()	*(creation)*
• copy()	*(shallow)*
• union()	|
• intersection()	&
• difference()	-
• symmetric_difference()	^
• issubset()	<=
• issuperset()	>=

Set Example

```
>>> set1 = frozenset([1,2,3])
>>> set2 = frozenset([3,4,5])
>>> set1  | set2
frozenset([1, 2, 3, 4, 5])
>>> set1 & set2
frozenset([3])
>>> set1 - set2
frozenset([1, 2])
>>> set2 - set1
frozenset([4, 5])
>>> set1 ^ set2
frozenset([1, 2, 4, 5])
>>> set1 <= set2
False
>>> set1 <= set1
True
>>> set1 > set1
False
```

Mutable Set Operations

- add(item)
- clear()
- update(iterable) *(adds items)* |=
- intersection_update(s2) &=
- difference_update(s2) -=
- symmetric_difference_update(s2) ^=
- discard(item) *(ignores if not there)*
- remove(item) *(exception if not there)*
- pop() *(removes/returns random)*

Exercises

- Determine the number of lines in a text file in a single Python statement
- Create a list with all the words (whitespace-delimited) from a file in one line of Python
- Define a dictionary, and then create another containing the data from the first where the keys and values are swapped. Make sure your values in the original are unique.

Functions

Functions – Topics

- Parameter Lists
- Function Attributes
- Generators and Generator Expressions
- Scope
- Decorators
- Dynamic and Anonymous Functions
- Functional programming

Point of Departure

```
#funparms.py
def h(x):
    return x + 2

def r(s):
    return s*2

# g calls f on x:
def g(f, x):
    return f(x)

print g(h,3)      # prints 5
print g(r,'two')  # prints twotwo
#print g(2,3)     # error: 2 is not callable
```


Passing Parameters

- Can have as many as you want
- Parameters can have *default values*
- Parameters can be accessed by *name* or by *position*
- The number of parameters can vary
- Each argument is *assigned* to its corresponding parameter
  - So the original binding at the call point is undisturbed (pass-by-value like Java)

Keyword Parameters

```
#keyparms.py
def displayStuff(name, address, city, zip, phone):
    print 'Name =', name
    print 'Address =', address
    print 'City =', city
    print 'Zip =', zip
    print 'Phone =', phone

displayStuff(address='some street',\
             city='somewhere', phone='123-4567',\
             name='john doe',  zip='98765')
displayStuff('john doe', 'some street',\
             phone='123-4567', city='somewhere',\
             zip='98765')
```


Variable-length Arg Lists

**parms Syntax*

```
#print_parms.py
def print_parms(*parms):
    print parms

def print_parms2(*parms):
    for x in parms:
        print x

def mymax(*parms):
    return max(parms)

print_parms(1,2,3)
print_parms2(1,2,3)
print mymax(1,2,3)
```

Output:

```
(1, 2, 3)
1
2
3
3
```


Going the Other Way

- You can *unpack* a tuple at the call site
  - Just use the asterisk *there*
  - It calls the function as if you had provided comma-separated arguments
  - They are unpacked in tuple order
- Example:

```
>>> pair = (2,3)
>>> pow(*pair)   # pow(2,3)
8
```

Variable-length Keyword Parameter Lists

- Allows a variable-length set of *keyword parameters*
- Passes a *dictionary* instead of a tuple
- Uses a double-asterisk (**`**parms`**)

```
>>> def print_parms(**parms):
...     print parms
...
>>> print_parms(foo='spam', bar='eggs')
{'foo': 'spam', 'bar': 'eggs'}
```


Using **parms

```
#keyparms2.py
def displayStuff(extra='', **stuff):
    print 'Name =', stuff['name']
    print 'Address =', stuff['address']
    print 'City =', stuff['city']
    print 'Zip =', stuff['zip']
    print 'Phone =', stuff['phone']
    if extra and stuff.has_key(extra):
        print stuff[extra]

displayStuff(name='john doe', address='some street',\
             city='somewhere', phone='123-4567',\
             zip='98765')
displayStuff('state',name='john doe',address='some street',\
             city='somewhere', phone='123-4567',\
             zip='98765',state='oblivion')
```


Going the Other Way

- You can pass a dictionary at the call site
- It is unpacked into arguments by their names
  - Like the “keyword” approach mentioned earlier
- See next slide

Summary of Calling Styles

```
#hello.py
def hello(name = 'world', greeting = 'hello'):
    print '%s, %s!' % (greeting, name)

hello()
hello(name = 'joe')
hello(name = 'joe', greeting = 'get lost')
stuff = ('jane','hello')
hello(*stuff)
stuff = {'name':'cruel world', 'greeting':'goodbye'}
hello(**stuff)
```


A Very Flexible Function

Accepts any number of args, positional or keyword

```
# allargs.py

def f(*args, **kwargs):
    for arg in args:
        print arg
    for key in kwargs:
        print key, '=', kwargs[key]

f(1,2,t=3,f=4)

# Output:
1
2
t = 3
f = 4
```


Function Attributes

- Some come for free:
  - func_name, func_doc, func_dict, func_globals
- You can define your own
  - Not often used
  - Canonical example: tracking number of function calls
  - See next slide

Using a Function-Call Counter

```
>>> def f():
...     f.count += 1
...     print f.func_name,'call count:',f.count
...
>>> f.count = 0
>>> f()
f call count: 1
>>> f()
f call count: 2
>>> f.count = 100
>>> f()
f call count: 101
```


Exercises

- Write a function named **superpower** that will raise its arguments to powers in succession. For example, the call **superpower(2,2,2)** computes **2**2**2**, and **superpower(2,2,2,4)** computes **2**2**2**4**. Remember that this operator associates *right-to-left*.
- Write a function **trinum()** that returns the next "triangular number" each time it is called (details in the Notes section below)

Generators

- Defines an *iterable* object via function syntax
  - Returns a "generator"
  - Can call **next( )** explicitly, or can just iterate over it with **for**
- Use **yield** instead of **return**
- Each "call" to the generator starts where the last call left off

Generator Example

```
def countgen():
   """An infinite count generator"""
   count = 0
   while True:
      count += 1
      yield count
f = countgen()
f
<generator object at 0x00C3E878>
f.next()
1
f.next()
2
f.next()
3
del f
```


Generator Expressions

- Look like list comprehensions
  - Using *parentheses* instead of brackets
- They define generators on-the-fly
- Used for *simple* generators only
  - Whatever can fit in a single expression
  - **yield** is not used

Generator Expression Example

```
nums = (i*i for i in range(5))
nums.next()
0
nums.next()
1
nums.next()
4
nums.next()
9
nums.next()
16
nums = (i*i for i in range(5))
sum(nums)
30
sum(nums)
0
```

Scope

- The region of code where a binding is visible
- Each scope has a "namespace"
  - A dictionary that holds bindings of variables to values (name is __**dict**__)
- When a name appears in code, its binding is looked up
  - If the name is not in the current (local) namespace, the enclosing scopes are searched

Scope Creation

- A scope is created for every:
  - Module
    - Each module is "global" to the functions, classes, and objects it contains
  - Function
    - This is a "local" namespace
    - Function definitions can be nested
  - Class
    - Similar to a module; contains nested definitions
  - Object

Scope Example

```
#scope.py
a = 1
n = 1

def f(n):
    print 'In f, a =', a, 'and n =', n, vars()

f(10)
print vars()

""" Output:
In f, a = 1 and n = 10 {'n': 10}
{'a': 1, 'f': <function f at 0x00AEFE30>,
'__builtins__': <module '__builtin__' (built-in)>, 'n':
1, '__name__': '__main__', '__doc__': None}"""
```

The LEGB Rule

- First, the current ("**local**") scope's namespace is searched
  - A local name "hides" an identical non-local name
- If the name is not found, its **enclosing** scope's namespace is searched
  - This could be a function or the **global** ("top-level") scope
- Finally, the **built-in** namespace is searched
- The **vars( )** function returns local bindings

Modifying Global Variables

- (Usually not a good practice, by the way)
- Remember that an **assignment** introduces a **new binding**
- Some special feature is needed to modify a global variable
- See next slide

The **global** Statement

```
#global.py
a = 2

def f():
    global a
    print vars()    # {}
    a = 4

print a     # 2
f()
print a     # 4
```

Another Scope Example

Illustrates a Nested Function

```
#scope2.py
a = 1
n = 1

def f(n):
    print 'In f, a =', a, 'and n =', n, vars()
    def g(a):
        print 'In g, a =', a, 'and n =', n, vars()
    g(5)

f(10)
print vars()
```


Output

```
In f, a = 1 and n = 10 {'n': 10}
In g, a = 5 and n = 10 {'a': 5, 'n': 10}
{'a': 1, 'f': <function f at 0x00AEFFB0>,
'__builtins__': <module '__builtin__' (built-in)>,
'n': 1, '__name__': '__main__', '__doc__': None}
```


Nested Functions

- Note that **f**'s binding of **n** appeared in the namespace for **g**
  - That's because **g** used **f**'s **n**
  - If it didn't, **vars( )** would've just mentioned **a**
- This packaging of needed non-local bindings in nested functions is called a *closure*
- It *really matters* when a function is *returned as a value*

Functions as Objects

```
#closure.py
def addn(n):
    def g(x):
        return x + n
    return g

f = addn(5)
print f(1)  # 6
print f(2)  # 7
```


Decorators

- A function-based application of the Decorator Design Pattern
- Wraps an existing function inside another
  - To provide before/after functionality
- We'll see two standard decorators later
  - **@staticmethod** and **@classmethod**
- The wrapper function:
  - Takes the function to decorate (**f**) as a parm
  - Returns a new function that calls **f**

Using a Decorator

```
def trace(f):
    def wrapper(*args1, **args2):
        print f.__name__,'with',args1,args2
        return f(*args1, **args2)
    return wrapper

@trace             # Same as "foo = trace(foo)" below
def foo(parm):     #           |
    print parm     #           |
                   #           V
@trace
def bar(parm1, parm2):
    print parm1, parm2
```

Sample Execution

```
foo(1)
bar(2,3)
foo(parm=4)
bar(5,parm2=6)

# Output:
foo with (1,) {}
1
bar with (2, 3) {}
2 3
foo with () {'parm': 4}
4
bar with (5,) {'parm2': 6}
5 6
```


Another Decorator

```
def bracket(f):
    def wrapper(*args1, **args2):
        print "<<< Start of", f.func_name,">>>"
        result = f(*args1, **args2)
        print "<<< End of", f.func_name,">>>"
        return result
    return wrapper
```


Chaining Decorators

```
@bracket
@trace
def spam(parm):
    print parm

spam((1,2))
```

Output:

```
<<< Start of wrapper >>>
spam with ((1, 2),) {}
(1, 2)
<<< End of wrapper >>>
```

Chaining Decorators

Swap Order of Decorators

```
@trace
@bracket
def eggs(parm):
    print parm

eggs((1,2))
```

Output:

```
wrapper with ((1, 2),) {}
<<< Start of eggs >>>
(1, 2)
<<< End of eggs >>>
```


Anonymous Functions

(Look back 4 slides for context)

- Note that **g** needed **addn**'s binding of **n**
- Note also that the *name* **g** is not really needed outside of **addn**
  - In fact, it's not needed *at all*!
- You can create simple, unnamed functions on-the-fly for cases just like this
  - With a **lambda** expression
  - The name is historical
- See next slide

Using **lambda**

```
#closure2.py
def addn(n):
    return lambda x: x+n

f = addn(5)
print f(1)  # 6
print f(2)  # 7
```

Customizing **sort( )** with **lambda**

```
#sort.py
stuff = [1,2,3,4,5]
stuff.sort(lambda x,y: y - x)
print stuff     # [5, 4, 3, 2, 1]
```


Functional Programming

- A style of programming that focuses on functions
  - They are passed as parameters and returned as values
- Some *support functions* make it very powerful
  - **map**, **reduce**, **filter**
- Overlaps in applicability with *list comprehensions*

A Functional Programming Session

```
>>> map(lambda x: -x, [1,2,3])
[-1, -2, -3]
>>> [-x for x in [1,2,3]]
[-1, -2, -3]
>>> map(lambda x,y: x+y, [1,2,3],[4,5,6])
[5, 7, 9]
>>> map(operator.add, [1,2,3],[4,5,6])
[5, 7, 9]
>>> reduce(operator.add, map(lambda x: -x, [1,2,3]))
-6
>>> [reduce(operator.add, x) for x in [(1,2), (3,4)]]
[3, 7]
>>> filter(lambda x: x > 2, [1,2,3])
[3]
>>> [x for x in [1,2,3] if x > 2]
[3]
>>> filter(None,[0,False,True,[],[1]])
[True, [1]]
```

A Closer Look at **reduce**

```
import operator
def times(nums):
    return reduce(operator.mul, nums, 1)
nums = [3.0,2.0,1.0]
print times(nums)

def sumsquares(nums):
    return reduce(lambda sofar,x: sofar + x*x, nums, 0)
print sumsquares(nums)
```

Output:
6.0
14.0

reduce with Logical Functions

```
import operator
def all(list_of_bools):
    return reduce(operator.and_, list_of_bools, True)

def any (list_of_bools):
    return reduce(operator.or_, list_of_bools, False)
```


Sample Output

```
bools = [True, False]
bools2 = [True, True]
bools3 = [False, False]
print all(bools)          False
print all(bools2)         True
print all(bools3)         False
print all([])             True
print any(bools)          True
print any(bools2)         True
print any(bools3)         False
print any([])             False
```

Function Composition

- Functional programmers like to apply functions in sequence
  - And sometimes to store the composition of multiple functions as a function itself:
  - funs = compose(f,g,h)
  - y = funs(x)   # f(g(h(x)))

Implementing **compose**

```
# A reasonable solution
def compose2(*funs):
    def doit(x):
        result = x
        for f in reversed(funs):
            result = f(result)
        return result
    return doit
```


A More FP Solution

```
def compose(*funs):
    return lambda x: reduce(lambda z,f: f(z), \
           list(reversed(funs)), x)
```

Classes and Objects

Classes and Objects – Topics

- Classes
- Static methods
- Accessibility
- Inheritance and Polymorphism
- Class Methods
- Bound and Unbound methods
- Metaclasses

Defining Classes

- The **class** statement
- A *class object* is created in the current namespace
  - Usually a module
  - Could also be nested in a function or a class

A Person Class

```
class Person(object):
    def __init__(self,last,first,month,day,year):
        self.last = last
        self.first = first
        self.month = month
        self.day = day
        self.year = year
    def name(self):
        return self.first + ' ' + self.last
    def birth(self):
        return "%d-%d-%d" % (self.month,self.day,self.year)
    def __str__(self):
        return "{%s,%s}" % (self.name(),self.birth())

p = Person('Doe','John',10,20,1930)
print p.name()        # John Doe
print p.birth()       # 10-20-1930
print p               # {John Doe,10-20-1930}
```


Instance Methods

- Defined inside a class with **def**
- Instance methods apply to instances of a class (**obj.method()**)
- The object that invoked the method is implicitly passed as the first parameter
  - Called **self** by convention
  - **p.name()** calls **Person.name(p)**
  - **p** becomes **self** in the function

Inspecting Person

```
>>> import person
John Doe
10-20-1930
{John Doe,10-20-1930}
>>> person.p
<person.Person object at 0x009E2F50>
>>> type(person.p)
<class 'person.Person'>
>>> person.Person
<class 'person.Person'>
>>> type(person.Person)
<type 'type'>
>>> dir(person)
['Person', '__builtins__', '__doc__', '__file__', '__name__', 'p']
>>> dir(person.Person)
['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__', '__hash_
_', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__setattr__', '__str__', '__weakref__', 'birth', 'name']
```

Inspecting p

```
>>> dir(person.p)
['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__', '__hash_
_', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__setattr__', '__str__', '__weakref__', 'birth', 'day', 'first', 'last', 'm
onth', 'name', 'year']
>>> vars(person.p)
{'year': 1930, 'month': 10, 'last': 'Doe', 'day': 20, 'first': 'John'}
>>> person.p.__dict__
{'year': 1930, 'month': 10, 'last': 'Doe', 'day': 20, 'first': 'John'}
```


Binding Attributes

- You can add attributes to modules, functions, classes, and objects *anytime*:
  - Dog.genus = 'canus' # class attribute
  - dog.scent = 'musty' # instance attribute
- If you bind a list (or tuple) of variable name strings to a class attribute named __**slots**__, then only those attributes are allowed in objects of that class

Adding Attributes to an Instance

```
>>> person.p.spam='eggs'
>>> vars(person.p)
{'last': 'Doe', 'spam': 'eggs', 'month': 10, 'year': 1930, 'day': 20, 'first': 'John'}
```

Restricting Person's Attributes

```
class Person(object):
    __slots__ = ['last','first','month','day','year']
    def __init__(self,last,first,month,day,year):
        # same as before…
```

```
>>> person.p.spam='eggs'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'Person' object has no attribute 'spam'
```


Class Data

- A class can have data attributes
- Called "static" members in C++
- They are not attached to an instance
  - They are attached to the one and only class object
- See next slide
  - Counts the number of objects created…

Class Data Example

```
class Counted(object):
    count = 0
    def __init__(self):
        Counted.count += 1

print Counted.count
c1 = Counted()
print Counted.count
c2 = Counted()
print Counted.count
```

Output:

0
1
2

Accessing Class Data

- It is not good practice to access class data directly
- Usually, methods are provided to access the data indirectly…

Static Methods

```
class Counted(object):
    count = 0
    def __init__(self):
        Counted.count += 1
    @staticmethod
    def getCount():     # Note no "self"
        return Counted.count

print Counted.getCount()
c1 = Counted()
print Counted.getCount()
c2 = Counted()
print Counted.getCount()
```


Accessibilty/"Hiding" Attributes

- Done by a naming convention
- Easy to circumvent
  - Python trusts you not to snoop!
- Prepend the attribute name with 2 underscores
  - But *not* 2 trailing underscores!

```
class Counted(object):
    __count = 0
    def __init__(self):
        Counted.__count += 1
    @staticmethod
    def getCount():     # Note no "self"
        return Counted.__count

print vars(Counted)
print Counted.__count
```

Output:

```
{'_Counted__count': 2, '__module__': '__main__', '__dict__': <attribute '__dict__' of 'Counted' objects>, 'getCount': <staticmethod object at 0x009E3BF0>, '__weakref__': <attribute '__weakref__' of 'Counted' objects>, '__doc__': None, '__init__': <function __init__ at 0x009E85F0>}
Traceback (most recent call last):
  File "static3.py", line 15, in ?
    print Counted.__count
AttributeError: type object 'Counted' has no attribute '__count'
```


Inheritance

```
class Employee(Person):
    __slots__ = ['title','salary']
    def __init__(self,last,first,month,day,year,title,salary):
        Person.__init__(self,last,first,month,day,year)
        self.title = title
        self.salary = salary
    def __str__(self):
        return "{%s,%s,%s,%f}" % \
          (self.name(),self.birth(),self.title,self.salary)

e = Employee('Doe','John',10,20,1930,'Gopher',12345.67)
print e        # {John Doe,10-20-1930,Gopher,12345.670000}
```


Name Lookup Algorithm

- When Python sees **obj.attr**:
  - It first looks in the namespace of **obj** for the attribute name
  - If the name is not found, *and* if **obj** is an instance of a class:
    - Python looks in all superclasses, bottom-to-top, left-to-right
      - The process repeats recursively up the inheritance graph
    - So an object’s class and superclasses are an “enclosing scope” for *qualified names*

Unqualified Names

- Names without a dot-prefix
- These only match *local* or *global* entities
- To refer to something in an object, class, or function, you *must* use the dot syntax
  – Even inside of class methods

Class Methods

- Do not exist in C++, Java, C#
- Like static methods, you usually call them qualified with the class name
- Whenever a class method is called, the *class object* is passed as a hidden first parameter
  – Analagous to **self**
  – **cls** is the conventional name

Class Methods

```
class Klass(object) :
    @classmethod
    def cmethod(cls, x):
        print cls.__name__, "got", x

Klass.cmethod(1)
k = Klass()
k.cmethod(2)

# Output
Klass got 1
Klass got 2
```

An Application of Class Methods

- Counting objects
  - The logic of counting is type-independent
  - How can we automatically make a class “countable”?
- Need some form of inheritance, but we want a *separate counter* for *each class*
  - We dynamically add a counter to each class through the class object parameter of a class method
- See next slide

OOPSLA 2007 Tutorial 154 Chuck Allison

Classy Counting

```
class Shape(object):
    __count = 0          # A shared initializer

    @classmethod
    def __incr(cls):
        cls.__count += 1 # Create/update class attribute

    @classmethod
    def showCount(cls):
        print 'Class %s has count: %s' % \
          (cls.__name__, cls.__count)

    def __init__(self):  # A constructor
        self.__incr()
```

OOPSLA 2007 Tutorial 155 Chuck Allison

Classy Counting

```
class Point(Shape): pass
class Line(Shape): pass
p1 = Point()
p2 = Point()
p3 = Point()
Point.showCount()
Line.showCount()
x = Line()
Line.showCount()

# Output:
Class Point has count: 3
Class Line has count: 0
Class Line has count: 1
```

OOPSLA 2007 Tutorial 156 Chuck Allison

Revisting +=

- The expression **cls.__count += 1** is the same as:

cls.__count = cls.__count + 1

New class ***Shape.__count (= 0)***

- (When **cls.__count** doesn't exist)

Multiple Inheritance

- No Big Deal!
- The name lookup algorithm finds what you're looking for
  - But the order you list base classes makes a difference

An Animal Kingdom

```
class Animal(object):
    def __init__(self, name):
        self.name = name
    def whoAmI(self):
        return self.name

class Dog(Animal):
    def __init__(self, name):
        Animal.__init__(self, name)
    def speak(self):
        print "Bark!"

class Antelope(Animal):
    def __init__(self, name):
        Animal.__init__(self, name)
    def speak(self):
        print "<silent>"
```

Adding a Combined Type

```
class Basselope(Dog, Antelope):
    def __init__(self, name):
        Animal.__init__(self, name)
```

```
bl = Basselope("Rosebud")
print bl.whoAmI(),':',
bl.speak()

# Output:
Rosebud : Bark!
```


Adding Methods "After the Fact"

```
def eat(self, food):
    print self.whoAmI(),'eating',food

Dog.eat = eat   # Add new method to Dog!

muffy = Dog('Muffy')
muffy.eat('trash')
Dog.eat(muffy, 'bones')

# Output:
Muffy eating trash
Muffy eating bones
```


Methods Are Objects

- A method can be bound to an arbitrary variable
- Two flavors:
  - Unbound method (**self** is an open variable)
  - Bound method (**self** object is *fixed*)
    - Like *delegates* in C# and D
    - A "closure" for objects; interchangeable with functions
- Handy for callbacks

Unbound Methods

```
op = Dog.whoAmI
print op
print op(muffy) # same as muffy.whoAmI()

# Output:
<unbound method Dog.whoAmI>
Muffy
```


Bound Instance Methods

```
sheba = Dog('Sheba')
op = sheba.whoAmI
print op
print op()      # same as sheba.whoAmI()
map(muffy.eat, ['melon', 'bones'])

# Output:
<bound method Dog.whoAmI of <__main__.Dog object
at 0x009FF130>>
Sheba
Muffy eating melon
Muffy eating bones
```


Bound Class Methods

```
# Class Methods are Bound Methods
# (Bound to their class object, of course)

m = Line.showCount
print m
m()

# Output:
<bound method type.showCount of <class '__main__.Line'>>
Class Line has count: 1
```

Metaclasses

- All objects have a type
- The type of a *class object* is a *metaclass*
- The standard metaclass for all built-in types and class types is the metaclass **type**
  - You can provide your own
- The class statement calls the metaclass to generate a new *class object*

The **type** Metaclass

```
>>> class C(object) : pass
>>> c = C()
>>> type(c)
<class '__main__.C'>
>>> type(C)
<type 'type'>

>>> type(1)
<type 'int'>
>>> type(int)
<type 'type'>

>>> type(type)
<type 'type'>
```


A Custom Metaclass

```
class MyMetaClass(type):  # derive from type
    def __str__(cls): return 'Class ' + cls.__name__

class C(object):
    __metaclass__ = MyMetaClass   # Assign metaclass

x = C()
print type(x)

# Output:
Class C
```

Metaclasses and __new__

- __new__ is different for metaclasses
  - It takes the class object, class name, list of base classes, and the class's namespace dictionary as arguments
- type.__new__ creates a new class object
- You can override it
- See next slide
  - Illustrates **`__slots__, getattr()`**

Adding Getters Automatically

```
class Getters(type):
    def __new__(cls, name, bases, d):
        for var in d.get('__slots__'):
            def getter(self, var=var):
                return getattr(self, var)
            d['get' + var] = getter
        return type.__new__(cls, name, bases, d)
class G(object):
    def __init__(self, f, b):
        self.foo = f
        self.bar = b
    __metaclass__ = Getters
    __slots__ = ['foo', 'bar']

g = G(1,2)
print g.getfoo(), g.getbar()     # 1 2
```


Course Summary

A Python Mantra – Courtesy Tim Peters

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Python is Better!