Totally Awesome Computing

Python as a General-purpose Object-oriented Programming Language

Chuck Allison

About Me

- Associate Professor of CS
  - Utah Valley State College
- A veteran C/C++ guy
  - Two books
  - 80+ articles
  - Past Senior Editor of C/C++ Users Journal
  - Past contributing member of C++ Standards Committee

About You

- You're very bright
  - You came to OOPSLA!
- Love to code
  - Problem-solving mentality
- Can stay alert while sitting for extended periods
- Dislike annoying slide animations

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
- 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

Python on The Web

- Visit www.python.org
- Can download Python and many related items of interest
- 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
On Perl vs. Python

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.

Agenda

1. The Nickel Tour
2. Built-in Types and Operations
3. Statements and Control Structures
4. Functions
5. Object-oriented Programming

Nickel Tour

- Starting Python
- Simple Expressions
- Variables and Dynamic Typing
- Saving and Running Programs

Starting Python

- Several ways:

1. Startup IDLE (or some other IDE)
2. Type “python” at the command prompt
3. Run a Python program
   - Click on a .py file, or
   - Enter “python <prog>” at command prompt

Prelude to Functions and Testing

- Define functions with the **def** statement
- Must indent body
  - And *all* subordinate blocks
    - **if**, **while**, etc.!
    - Introduced by a colon
- Test frameworks:
  - doctest (matches output in comments)
  - pyUnit (Python XUnit framework)
    - Will see later after classes are introduced

helloworld.py

```
''' helloworld.py:

    Hello World for Python. Illustrates doctest.

    >>> hello()
    Hello, world!
'''

def hello():
    print "Hello, world!"

import doctest
doctest.testmod()
```


“Life is Better Without Braces”

- No more religious wars on where to put the { and }
- :-)

Built-in Types and Operations

Python Built-in Types

- Numbers
- Sequences
  - Linear data structures
- Dictionaries & Sets
  - No duplicates
- Files

Numbers

- Integers
  - 32-bit: from -2,147,483,648 to 2,147,483,647
- Long integers
  - Unlimited size! (Automatic promotion from **int**)
- Real numbers (decimals)
  - Same as **double** in C
- Complex numbers
  - 2 + 3j

Numeric Operators
C plus 2 more

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

Division Example

```
>>>1 / 2
0
>>> 1.5 / 2.5
0.59999999999999998
>>> 1.5 // 2.5
0.0
>>> from __future__ import division
>>> 1 / 2
0.5
```


Long Integers

- Arbitrarily large
  - Regular integers convert automatically to long:
-
```
>>> x = 12345
>>> type(x)
<type 'int'>
>>> x **= 5
>>> x
28671833852463546562 5L
>>> type(x)
<type 'long'>
>>> x ** 5
19376590304114639356511673916564226265776144115861523176748692334640199227714321588721871376037597656 25L
```


Sequences

- Linear, expandable collections accessed by position
  - (0-based)
- 3 types:
  - Strings
  - Lists
  - Tuples
- Note: strings and tuples are *immutable*

Common Sequence Operations

- Functions: len( ), min( ), max( ), zip( ), sum( ) (**sum** doesn't work on strings)
- Comparisons: <, <=, >, >=, ==, !=
- Concatenation with +
- Replication with *
- Membership tests with **in**
- Iteration with **for**
- Slices (subsequences)

Using split( ) and join( )
Example

```
>>> s = 'a,b,c d,e'
>>> s.split()
['a,b,c', 'd,e']
>>> s.split(',')
['a', 'b', 'c d', 'e']
>>> (s+',,').split(',')
['a', 'b', 'c d', 'e', '', '']
>>> ','.join(['x','y','z'])
'x,y,z'
>>> ''.join(['x','y','z'])
'xyz'
```


Slices

- Extracts an indexed subsequence
- Syntax: **s[start:endp1]**
  - For example **"hello"[1:3]** == **"el"**
- **s[p:]** extracts from position **p** to the end
- **s[:p]** extracts from the beginning up to *but not including* position **p** (and **p** items are extracted)

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)]**

Working with Lists

- Lists are *mutable* sequences of values
  - Can change them with list methods
- Delimited by brackets: [a,b,c]
- Can store any mixture of types
- Lists store a *reference* to the objects they logically contain
  - It is possible (and common) for different lists to refer to common objects

List Methods

- count(x) — *Number of x's*
- index(x) — *Where first x is*
- pop(i = -1) — *Removes at position*
- append(x) — *Appends an element*
- extend(seq) — *Appends a sequence*
- insert(i,x) — *Inserts x at position i*
- remove(x) — *Removes first x*
- reverse() — *Obvious!*
- sort(f = cmp) — *Sorts in place*

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']
```


List Comprehensions

- A powerful list-creation facility
- Uses special expression syntax in brackets
- The result list is returned:

```
>>> 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]
```


QuickSort with List Comprehensions

```
def qsort(L):
   if len(L) <= 1: return L
   return qsort([lt for lt in L[1:] if lt < L[0]]) \
        + [L[0]] \
        + qsort([gt for gt in L[1:] if gt >= L[0]])
```


Working with Tuples

- Tuples are like lists, but they are *immutable*
- Delimited by parentheses: (a,b,c)
- Commonly used to return multiple values from a function
- Also used for multiple assignment
  - Swap idiom (see next slide)

Multiple Assignment with Tuples

- You can place tuples on either side of an assignment
  - Parentheses are optional in this case
- The assignments are made component-wise:

```
>>> x,y = 1,2
>>> x
1
>>> y
2
>>> x,y = y,x
>>> x
2
>>> y
1
```


Dictionaries

- Hash Tables
- Associate a key with a value
  - Stored as unordered pairs
  - Keys must be immutable and unique
- Fast lookup by key
  - Not position
  - Uses index notation

Working with Dictionaries

- Delimited by braces: m = {'a' : 1, 'b' : 2}
- Access by key with brackets: m['a'] == 1
- Add pairs the same way: m['c'] = 3
  - Now m == {'a' : 1, 'b' : 2, 'c' : 3}
  - This also can update a value: m['a'] = 20
    - Now m == {'a' : 20, 'b' : 2, 'c' : 3}

Dictionary Methods

- has_key(x) (can also use “in”)
- keys( )
- values( )
- items( ) (returns list of pairs)
- copy( ) (shallow copy)
- d1.update(d2) (merge 2 dictionaries)
- get(key, def = None)
- setdefault( ) (= get( ) + create)
- Use **del** to remove items by key

Dictionary Example

```
>>> phonelist = {}
>>> phonelist['joe'] = '123-4567'
>>> phonelist['bob'] = '890-1234'
>>> keys = phonelist.keys()
>>> keys.sort()
>>> keys.items()
>>> keys.sort(lambda s, t: -cmp(s,t))
>>> keys
['joe', 'bob']
>>> [(name, phonelist[name]) for name in keys]
[('joe', '123-4567'), ('bob', '890-1234')]
>>> phonelist.setdefault('jane', 'unlisted')
'unlisted'
>>> phonelist
{'jane': 'unlisted', 'bob': '890-1234', 'joe': '123-4567'}
```


File Methods

- open(<fname>, <mode> = ‘r’)
- read( ) *(whole file into a string)*
- read(n) *(n bytes)*
- readline( ) *(next line)*
- readlines() *(all lines; also xreadlines( ))*
- write(s)
- writelines(L)
- close( )

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()

>>> f = open('test.dat')
>>> for line in f: print line,
This is line 1
line 2 start line 2 end
line 3
```


Statements and Control Structures

Programs

- Python programs are made of modules
  - Including one **main** module (the one you launch Python with)
  - Modules may be gathered into *packages*
- Modules are made up of statements
- Statements consist of keywords and/or expressions

Program => Modules => Statements

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

The **if** statement

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

Loops

- **for** <item> **in** <iterable object>
- **while** <condition>:
  <suite>
  [**else:**
  <suite>]
  - Continues until the condition becomes **False**
  - Tests first before executing the block
  - An optional **else** clause executes if the loops completes

Generators

- Defines an iterable object with function syntax
  - Returns a “generator”
  - Can call the next( ) explicitly, or can just iterate over it
- 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"""   # A doc string
   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

- Similar to list comprehensions
- Defines generators on-the-fly
- For simple generators
  - **yield** is not used
  - The iteration is implicit from the generated sequence
- Just place the expression in parentheses

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()
25
nums = (i*i for i in range(5))
sum(nums)
30
sum(nums)
0
```


Functions

Topics

- Duck Typing
- Parameter Lists
- Decorators
- Namespaces and Name Lookup (Scope)

A Fibonacci Function

```
#Returns the nth Fibonacci number
def fibonacci(n):
    curr,prev = 0,1
    for i in range(n):
        curr,prev = prev+curr,curr
    return curr

# Note: We didn't "declare" n to be int
```

Parametric Polymorphism

- If a function can take different types of parameters, it is called “polymorphic”
  - In particular, it exhibits “parametric polymorphism”
- This is also called “duck typing”
  - “If it looks like a duck, quacks like a duck,…”
  - aka “Structural Conformance”
- C++ uses function templates

Illustrating Duck Typing

- Python’s **min( )** function
- Can process sequences of any type
  - The sequence must contain objects that can be compared compatibly
    - For example, not a mixture of numbers and strings
- The contained type(s) must just support a less-than operator
- See next slide

A min( ) function

```
def mymin(stuff):
    smallest = stuff[0]
    for x in stuff[1:]:
        if x < smallest: smallest = x
    return smallest

print mymin([4,3,2,1])
print mymin(['how','now','brown','cow'])
```

Output:

```
1
brown
```


Passing Parameters

- Parameters can have *default values*
  - Must be *immutable* objects
- Parameters can be accessed *positionally* or by *name*
- Variable-length parameter lists supported
- Each argument is *assigned* to its corresponding parameter
  - So the original binding at the call point is undisturbed (like Java; no pass-by-reference)

Parameters as Keywords

- Sometimes functions can have *many parameters*
  - Their order can be hard to remember
- If you know their names, you can use them explicitly in the call
  - And not worry about the order
- See next slide

Using Parameters as Keywords

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


Variable-length Argument Lists

- Arguments passed to a function can be collected into a *tuple*
- You just process the tuple on the receiving end
  - i.e., inside the called function
- Use an *asterisk* before the parm name
- See next slide

Using Variable-length Arg Lists

```
#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)    # same as pow(2,3)
8
```


Variable-length Keyword Args

- You can have any number of keyword parameters
  - Must follow all positional parameters
- Pythpn passes a *dictionary* instead of a tuple
  - Use a *double-asterisk* (**`**parms`**)
- See next slide

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 individial keyword arguments
- See next slide

All About Call Conventions

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

# Predict the output below:
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
```


Decorators

- Functions that alter the definition of other functions
- They act as *wrappers*
  - Your function is passed to them as a parameter
  - They replace the original function with a new function that “decorates” the original
- They can be chained
- See also: @staticmethod, @classmethod

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)”
def foo(parm):
    print parm

@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
```


Namespaces

Name Lookup Tables

- A namespace is created for every:
  - Module
    - This is the “global” namespace/scope
  - Function
    - This is a “local” namespace/scope
    - There is no “block scope”
  - Class
  - Class instance (object)

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

Name Lookup 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__': 'scope.py'}
```


Modifying Global Variables

- Remember that an **assignment** introduces a **new binding**
  - So you need some special feature to modify a global variable
  - Otherwise a new local is created
- *Note*: you can't directly modify any unqualified, non-local variables other than globals
- 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
```


Object-oriented Programming

Topics

- Classes
- Inheritance
- Accessibility and Properties
- Static methods vs. Class methods
- Bound vs. Unbound methods
- Metaclasses
- Operator Overloading (if time allows)

Defining Classes

- The **class** statement
- Evaluated at *runtime*
  - Like everything else in Python
- A *class object* is created in the current namespace
- The class object is a *factory*:
  - You "call" it to create instances
- Example starts on next slide

An Animal Class

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

>>> a = Animal('Rocky Raccoon')
>>> print a.whoAmI()
Rocky Raccoon
```


Things to Note

- All top-level classes should derive from the built-in class **object**
  - New Style vs. Classic classes
- *Instance methods* receive a hidden first parameter pointing to the object
  - Called **self** by convention
- *Instance attributes* are defined dynamically by assignment
  - Like any other Python attribute

Special Methods

- Names beginning and ending with double underscores are special
- **__init__( )** is a *constructor*
  - Called to initialize an object at creation
- There are many special methods
  - e.g., operator overloading: __add__, __call__

Unit Testing with PyUnit

```
# hellotest.py
import unittest
import hello

class HelloTestCase(unittest.TestCase):
    def testHello(self):
        assert hello.sayhello() == "Hello, world"

if __name__ == "__main__":
    unittest.main()
```


Specific Animal Classes

Inheritance

```
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>"
```


More Observations

- Note the explicit calls to the **Animal** constructor
  - It's not done automatically
    - For good reason (lookup algorithm)
  - And the object reference (**self**) is also passed explicitly
- The **speak** method is introduced
  - The most derived method is chosen by the lookup algorithm (explained shortly)

Bringing Animals to Life

```
dog = Dog("Fido")
print dog.whoAmI(),':',
dog.speak()

ant = Antelope("Arnie")
print ant.whoAmI(),':',
ant.speak()

# Output:
Fido : Bark!
Arnie : <silent>
```


Binding Attributes

- You can add attributes to modules, classes, and objects *anywhere* via assignment:
  - Dog.genus = ‘canus’ # class attribute
  - dog.scent = ‘musty’ # instance attribute
- If you bind a list of variable name strings to a class attribute named __**slots**__, then only those attributes are allowed in objects of that class

Accessibility

- All names are public!
  - Can cause name collisions
- There is a *convention* to “reduce” name visibility:
  - Start attributes with two underscores
    - Do not use two trailing underscores
  - Such names are “mangled”
    - By prepending ‘_’ plus the class name
    - See next two slides

Pseudo-Private Attributes

```
class Controlled(object):
    def __init__ (self, x, y):
        self.__x = x
        self.__y = y
    __z = 'zee'

    def display(self):
        print 'x:',self.__x,'y:',self.__y, \
              'z:',Controlled.__z
```


Output

```
c = Controlled('ecks', 'why')
c.display()
print dir(c) # print c’s namespace
c.__x

# Output:
x: ecks y: why z: zee
['_Controlled__x', '_Controlled__y', '_Controlled__z',
'__doc__', '__init__', '__module__', 'display']
Traceback (most recent call last):
  File "C:\Presentations\OOPSLA\classes.py", line 106, in ?
    c.__x
AttributeError: Controlled instance has no attribute '__x'
```


Properties

- A way to transparently use getters and setters through data-like attribute reference
  - More efficient than getters if no computation is involved
- Use the **property** special function
  - 4 parms: getter, setter, deleter, doc string
  - Last 3 default to **None**
  - Omitted operations are not available

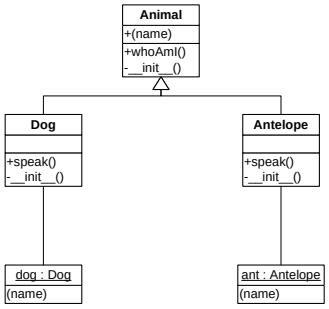
Properties

```
class HasProp(object):
    def setProp (self, p):
        print 'setProp trace'
        self.__prop = p
    def getProp (self):
        print 'getProp trace'
        return self.__prop
    prop = property(getProp, setProp, None, None)

x = HasProp()
x.prop = 'theProperty'
print x.prop

# Output:
setProp trace
getProp trace
theProperty
```


Classes are Objects



Classes are Objects

- *Objects* named **Animal**, **Dog**, and **Antelope** exist
  - Created by their respective **class** statements
- They have a *namespace* (dictionary) mapping their attributes to values
  - **Animal** has **__init__** and **whoAmI**
  - But not **name**!
  - **name** belongs to the object bound to **self**

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, 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*

Multiple Inheritance

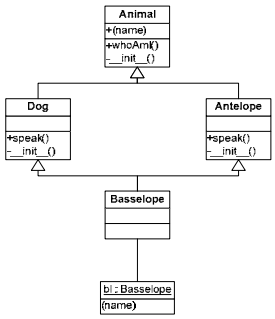
```
class Baselope(Dog, Antelope):
    def __init__(self, name):
        Animal.__init__(self, name)

# Note the direct reference to Animal.__init__
# Alternatively:
#       super(Baselope,self).__init__(name)

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

# Output:
Rosebud : Bark!
```


An MI Graph



Static (Class-based) Data

- Assignments performed directly inside a class define *static* (aka *class*) data attributes
- They are added to the namespace of the class
- Usually accessed as <*class*>.<*attribute-name*>
- (See next slide)
- **Note**: In methods, all attributes of a class or object must be accessed via the class or an object name (e.g., **self.attr**) – *unqualified names* in methods are considered *global*!

Static Data

```
class Foo(object):
    data = 'bar'
    def __init__(self,data):
        self.data = data
    def display(self):
        print self.data, Foo.data

foo = Foo('baz')
foo.display()

# Output:
baz bar
```


Static Methods

- You can define methods that aren't connected to an object
- Normally accessed as:
  - *<class-name>.<method-name>*
- Use the **@staticmethod** *decorator*
- See next slide

Static Stuff

```
class StaticStuff(object) :
    foo = 1
    @staticmethod
    def bar () :
        return StaticStuff.foo

x = StaticStuff()
print x.foo
print StaticStuff.bar()

# Output:
1
1
```


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

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()
```


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
```


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)

C++ Version

A Curiously Recurring Template Pattern

```
template<class T> class Counted {
    static int count;
public:
   Counted() { ++count; }
   Counted(const Counted<T>&) { ++count; }
   ~Counted() { --count; }
   static int getCount() { return count; }
};
template<class T> int Counted<T>::count = 0;

class CountedClass : public Counted<CountedClass> {};
class CountedClass2 : public Counted<CountedClass2> {};
```


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

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

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


Another Example

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

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

dogs = [muffy, sheba]
food = ['melon', 'bones']
for i in range(2):
    eat(dogs[i], food[i])

# Output:
Muffy eating melon
Sheba eating bones
```


Pairing Objects and Arguments

```
pairs = zip(dogs, food)
print pairs
for pair in pairs:
    eat(*pair)

# Output:
[(<__main__.Dog object at 0x00A03090>, 'melon'),
(<__main__.Dog object at 0x00A030D0>, 'bones')]
Muffy eating melon
Sheba eating bones
```


Bound Instance Methods

```
sheba = Dog('Sheba')
op = sheba.whoAmI
print op
print op()      # same as sheba.whoAmI()
map(muffy.eat, food)

# 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 the class, 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 its *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

x = C()
print type(x)

# Output:
Class C
```


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
```


Operator Overloading

Operator Functions

- All operators have associated functions
  - e.g., '+' corresponds to __**add**__
- Defining these as instance methods overloads the corresponding operator
- Others: __or__, __str__, __call__, __getattr__, __setattr__, __getitem__, __setitem__, __len__, __cmp__, __lt__, __eq__, __iter__, __contains__, …

Numeric Operators

```
class Number(object):
    def __init__(self, num):
        float(num)  # correctness test
        self.__num = num
    def __add__(self, num):
        if (isinstance(num, Number)):
            return Number(self.__num + num.__num)
        return Number(self.__num + float(num))
    def __str__(self):
        return '%g' % self.__num
    def __neg__(self):
        return Number(-self.__num)
    __radd__ = __add__
```


Sample Execution

```
x = Number(1.2)
y = Number(3.4)
print x,y,-x
print x + y
print x + 2
print 2 + x

# Output:
1.2 3.4 -1.2
4.6
3.2
3.2
```


Indexing

- The method **__getitem__(self, i)** is called when fetching **self[i]**
- Likewise, **__setitem__(self, i, x)** processes **self[i] = x**
- See next two slides
- Can also overload **__getslice__** and **__setslice__**

Indexing Example

```
class Stuff(object):
    def __init__(self):
        self.__data = []
    def add(self, x):
        self.__data.append(x)
    def __getitem__(self, i):
        return self.__data[i]
    def __setitem__(self, i, x):
        self.__data[i] = x
    def display(self):
        for item in self.__data:
            print item,
        print
    def __len__(self):
        return len(self.__data)
```


Indexing Example

```
s = Stuff()
s.add(2)
s.add('three')
s.add(4.0)
s.display()
for i in range(len(s)):
    print s[i],
print
s[1] = 'one'
s.display()

# Output:
2 three 4.0
2 three 4.0
2 one 4.0
```


Iteration

- You can define your own iterators
  - They can be used for loops and in other iterable contexts
  - Loops call **iter( )** to get an iterator
- The **iter( )** built-in will in turn call your **__iter__**, or **__getitem__** (in that order)
  - Iteration is automatic if you define **__getitem__**!

Iteration via __getitem__

```
for x in s:       # s from 2 slides ago (Stuff)
    print x,
print
print 'one' in s
print map(None, s)
print list(s)
print tuple(s)

# Output:
2 one 4.0
True
[2, 'one', 4.0]
[2, 'one', 4.0]
(2, 'one', 4.0)
```


Using __iter__

- Better for one-pass traversal than __**getitem**__
  - You must implement a **next** method and maintain state
  - Faster than __**getitem**__, since state is maintained across calls
- See next slides

Using __**iter**__ and **next**

```
class MoreStuff(object):
    def __init__(self):
        self.__data = []
    def add(self, x):
        self.__data.append(x)
    def __iter__(self):
        self.__pos = 0
        return self
```


Using __**iter**__ and **next**

```
    def next(self):
        if self.__pos == len(self.__data):
            raise StopIteration
        val = self.__data[self.__pos]
        self.__pos += 1
        return val
    def display(self):
        for item in self.__data:
            print item,
        print
```


Sample Execution

```
s = MoreStuff()
s.add(2)
s.add('three')
s.add(4.0)
s.display()
for x in s:
    print x,
print

# Output:
2 three 4.0
2 three 4.0
```


Summary

- Python is easy to learn
  - Clean syntax
  - Orthogonal feature set
- Python is easy to use
- Python is “fast enough”
- Python is fully object-oriented

Python is Better

A Python Community “Mantra”

- 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.
- Python is better.

Appendix

Exercises

Homework
Strings and Slices

- Assign the string "how now brown cow" to a variable
- Compute the length of the string
- Extract the word "how"
- Extract the word "cow"
- Extract the word "brown"
- Extract all characters up through position 6 (7 letters)
- Extract all characters from position 7 to the end
- Find all positions of the substring "ow" (use **.find( )**)
- Prepend the string "Well, ", overwriting the variable
- Append a question mark (overwriting again)
- Compute the new length

Homework
Lists and Files

- Write a program that reads a text file, and reports the number of text lines in the file, the length of the longest line, the length of the shortest line, and then prints the lines out in the order they appear in the file.
  - (You will need at least one **for**-loop)

Homework
Lists and Files

- Write a program that prints out the number of characters, words, and lines in a text file

Homework
Dictionaries, Lists

- Create a small phone book using a dictionary. The key is a name string and the value is a string containing the person's phone number. Start by creating an empty dictionary: **pbook = { }**. Then add 3 entries and print the dictionary. Delete one of the entries with **del** and reprint**.**
- Extra credit: Have the value be a *list* of strings containing all the person's phone numbers.
- Extra-extra credit. Have the value be a *dictionary* indicating whether the number is 'home', 'work', 'cell', or whatever.

Exercise
Dictionaries and Files

- Read a text file and report how many occurrences of each word there are.

Homework
Dictionaries, Lists and Files

- Repeat the word frequency program, but print out the 10 most-frequently occurring words and their count, for example as follows:

```
    the:   101
      a:    54
     to:    46
 string:    46
     in:    44
      i:    40
    and:    38
     of:    35
     is:    22
    for:    22
```


Exercises
Loops

- Print out every other element of a list

Exercises
Parameter Passing Mechanisms

- Write a function named **superpower** that will raise its arguments to powers in succession. For example, the call **superpower(2,3,4)** computes **2**3**4**, and superpower**(2,3,4,5)** computes **2**3**4**5**. Remember that this operator associates right-to-left.

Exercise
Recursion

- Write a recursive version of **superpower**.

Programming Assignment #1
Functions

- Write a function named **sentence( )** that generates random sentences according to the following grammar:

  <S> => the <NP> <VP>
  <NP> => <N> | <ADJ> <NP>
  <VP> => <V> | <V> <ADV>
  <N> => dog | cat | professor | student | rat
  <V> => ran | ate | slept | drank
  <ADJ> => red | slow | dead
  <ADV> => quickly | happily | well

  You can just define the words above as lists:

```
nounlist = ['dog','cat','professor','student','rat']
verblist = ['ran','ate','slept','drank']
adjlist = ['red','slow','dead']
advlist = ['quickly','happily','well']
```


Programming Assignment #1
(continued)

- As you can by the grammar above, a *sentence* is just the word "the" followed by a noun-phrase followed by a verb-phrase. A *verb-phrase* is either a verb or a verb followed by an adverb. A *noun-phrase* is either a noun or an adjective followed by a *noun-phrase(!)*. Write functions named **verbphrase( )** and **nounphrase( )** that are called by sentence. Note that **nounphrase( )** is recursive. You also will want to use the function **randrange( )** in the **random** module. Here is the output from several successive calls to **sentence( )**:

```
The rat slept well.
The slow dead professor drank well.
The dog drank quickly.
The dead dead slow cat slept quickly.
```


Programming Assignment #2
Functions, Dictionaries

Write a function **xref(s)** that reads the text file named by **s** line by line and returns a dictionary with each word in the file as a key paired with a list of the line numbers where the word appears. Use **xref** to print out a *cross-reference listing*, which lists each word in alphabetic order, ignoring case. Here is what the output should resemble:

```
A : [48]
a : [9, 10, 12, 14, 17, 19, 26, 27, 28, 39, 41, 43, 45, 46, 49,
   50, 51, 56, 81, 82, 94, 111, 112, 114, 117, 132, 135, 138, 142,
   143, 144, 152, 156, 161, 163, 164, 167, 169, 175, 182, 190,
   192]
about : [16, 29, 166, 190, 191]
…
yet : [191]
Yet : [181]
You : [45, 169]
you : [19, 41, 44, 81, 90, 112, 113, 134, 135, 143, 148, 170, 174,
   179, 188, 190]
```

