The Python Programming Language

Presented at USENIX Technical Conference
June 14, 2009

David M. Beazley
http://www.dabeaz.com

(Part 2 - Libraries and Program Organization)

Problem

- If you write a useful script, it will grow features
- You may apply it to other related problems
- Over time, it might become a critical application
- And it might turn into a huge tangled mess
- So, let's get organized...

Part 2 Overview

- In this section, we focus our attention on writing larger programs
  - Functions
  - Modules and libraries
  - Creating user-defined objects (classes)
  - Metaprogramming
- We'll also look at the standard library

Part 1

Defining Functions

Functions

- Create functions with the def statement

```
def read_portfolio(filename):
    stocks = []
    for line in open(filename):
         fields = line.split()
         record = { 'name' : fields[0],
                    'shares' : int(fields[1]),
                    'price' : float(fields[2]) }
         stocks.append(record)
    return stocks
```

- Using a function

```
stocks = read_portfolio('portfolio.dat')
```


Function Examples

```
# Read prices into a dictionary
def read_prices(filename):
    prices = { }
    for line in open(filename):
         fields = line.split(',')
         prices[fields[0]] = float(fields[1])
    return prices

# Calculate current value of a portfolio
def portfolio_value(stocks,prices):
    return sum([s['shares']*prices[s['name']]
                for s in stocks])
```

Function Examples

- A program that uses our functions

```
# Calculate the value of Dave's portfolio

stocks = read_portfolio("portfolio.dat")
prices = read_prices("prices.dat")
value  = portfolio_value(stocks,prices)

print "Current value", value
```

- Commentary: There are no major surprises with functions--they work like you expect

What is a function?

- A function is a sequence of statements

```
def funcname(args):
    statement
    statement
    ...
    statement
```

- Any Python statement can be used inside
- This even includes other functions
- So, there aren't many rules to remember

Function Definitions

- Functions can be defined in any order

```
def foo(x):
    bar(x)

def bar(x):
    statements
```

```
def bar(x)
    statements

def foo(x):
    bar(x)
```

- Functions must only be defined before they are actually used during program execution

```
foo(3)        # foo must be defined already
```

- Stylistically, it is more common to see functions defined in a "bottom-up" fashion

Bottom-up Style

- Functions are treated as building blocks
- The smaller/simpler blocks go first

```
# myprogram.py
def foo(x):
    ...
def bar(x):
    ...
    foo(x)
    ...
def spam(x):
    ...
    bar(x)
    ...

spam(42)      # Call spam() to do something
```

Later functions build upon earlier functions

Code that uses the functions appears at the end

A Definition Caution

- Functions can be freely redefined!

```
def foo(x):
    return 2*x

print foo(2)       # Prints 4

def foo(x,y):      # Redefine foo().  This replaces
   return x*y      # foo() above.

print foo(2,3)     # Prints 6
print foo(2)       # Error : foo takes two arguments
```

- A repeated function definition silently replaces the previous definition
- No overloaded functions (vs. C++, Java).

Scoping of Variables

- All variables assigned in a function are local

```
def read_prices(filename):
    prices = { }
    for line in open(filename):
        fields = line.split(',')
        prices[fields[0]] = float(fields[1])
    return prices
```

- So, everything that happens inside a function call stays inside the function
- There's not much to say except that Python behaves in a "sane" manner

Global Scoping

- Functions can access names that defined in the same source file (globals)

```
delimiter = ','
def read_prices(filename):
    ...
    fields = line.split(delimiter)
    ...
```

- However, if you're going to modify a global variable, you must declare it using 'global'

```
line_num = 1     # Current line number (a global)
def parse_file(filename):
    global line_num
    for line in open(filename):
        # Process data
        ...
        line_num += 1
```


Default and Keyword Args

- Functions may take optional arguments

```
def read_prices(filename,delimeter=None):
    prices = { }
    for line in open(filename):
         fields = line.split(delimeter)
         prices[fields[0]] = float(fields[1])
    return prices
```

- Functions can also be called with named args

```
p = read_prices(delimeter=',',filename="prices.csv")
```

- There is a lot of flexibility, but the only requirement is that all arguments get values.

Parameter Passing

- Parameters are just names for values--no copying of data occurs on function call

```
def update(prices,name,value):
    # Modifies the prices object.  This
    # does not modify a copy of the object.
    prices[name] = value
```

- If you modify mutable data (e.g., lists or dicts), the changes are made to the original object

```
>>> prices = { }
>>> update(prices,'GOOG',490.10)
>>> prices
{ 'GOOG' : 490.10 }
>>>
```


Return Values

- return statement returns a value

```
def square(x):
    return x*x
```

- If no return value, None is returned

```
def bar(x):
    statements
    return

a = bar(4)    # a = None
```

- Return value is discarded if not assigned/used

```
square(4)     # Calls square() but discards result
```

Multiple Return Values

- A function may return multiple values by returning a tuple

```
def divide(a,b):
    q = a // b      # Quotient
    r = a % b       # Remainder
    return q,r      # Return a tuple
```

- Usage examples:

```
x   = divide(37,5)    # x = (7,2)

x,y = divide(37,5)    # x = 7, y = 2
```


Advanced Functions

- Normally, you think of a function as receiving a set of inputs (parameters) and producing a single output
- Python has a few advanced function types that have different behavior
    - Generators
    - Coroutines
- A big topic, but I'll give a small taste

Generator Functions

- A function that generates a sequence of output values (using yield)

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

- Instead of a single result, this kind of function produces a sequence of results that you usually process with a for-loop

```
>>> for x in countdown(10):
...     print x,
...
10 9 8 7 6 5 4 3 2 1
>>>
```


Generator Functions

- There are many interesting applications
- For example, this function mimics 'tail -f'

```
import time

def follow(f):
    f.seek(0,2)     # Seek to EOF
    while True:
        line = f.readline()
        if not line:
            time.sleep(0.1)
            continue
        yield line
```

- Example:

```
for line in follow(open("access-log")):
    print line,
```

Coroutines

- A function that receives a series of input values

```
def grepper(pat):
    while True:
        line = (yield)
        if pat in line:
           print line
```

- This is a function that you launch and then subsequently "send" values to

```
>>> g = grepper('python')
>>> g.next()              # Required
>>> g.send('yeah, but no, but yeah')
>>> g.send('python is nice')
python is nice
>>> g.send('boy eaten by huge python')
boy eaten by huge python
>>>
```


More Information

- Generators and coroutines have a variety of practical applications, but they require their own tutorial
- "Python Generator Hacking" (Tomorrow)

Part 2

Modules

From Functions to Modules

- As programs grow, you will probably want to have multiple source files
- Also to create programming libraries
- Any Python source file is already a module
- Just use the import statement

A Sample Module

```
# stockfunc.py

def read_portfolio(filename):
    lines = open(filename)
    fields = [line.split() for line in lines]
    return [ { 'name' : f[0],
               'shares' : int(f[1]),
               'price'  : float(f[2]) } for f in fields]
# Read prices into a dictionary
def read_prices(filename):
    prices = { }
    for line in open(filename):
         fields = line.split(',')
         prices[fields[0]] = float(fields[1])
    return prices

# Calculate current value of a portfolio
def portfolio_value(stocks,prices):
    return sum([s['shares']*prices[s['name']]
                  for s in stocks])
```


Using a Module

- importing a module

```
import stockfunc

stocks = stockfunc.read_portfolio("portfolio.dat")
prices = stockfunc.read_prices("prices.dat")
value  = stockfunc.portfolio_value(stocks,prices)
```

- Modules serve as a container for all "names" defined in the corresponding source file (i.e., a "namespace").
- The contents accessed through module name

Module Execution

- When a module is imported, all of the statements in the module execute one after another until the end of the file is reached
- If there are scripting statements that carry out tasks (printing, creating files, etc.), they will run on import
- The contents of the module namespace are all of the global names that are still defined at the end of this execution process

Globals Revisited

- Everything defined in the "global" scope is what populates the module namespace

```
# foo.py
x = 42
def grok(a):
    ...
```

```
# bar.py
x = 37
def spam(a):
    ...
```

These definitions of x are different

- Different modules can use the same names and those names don't conflict with each other (modules are isolated)

Module Loading

- Each module loads and executes once
- Repeated imports just return a reference to the previously loaded module
- sys.modules is a dict of all loaded modules

```
>>> import sys
>>> sys.modules.keys()
['copy_reg', '__main__', 'site', '__builtin__',
'encodings', 'encodings.encodings', 'posixpath', ...]
>>>
```


Locating Modules

- When looking for modules, Python first looks in the current working directory of the main program
- If a module can't be found there, an internal module search path is consulted

```
>>> import sys
>>> sys.path
['', '/Library/Frameworks/
Python.framework/Versions/2.5/lib/
python25.zip', '/Library/Frameworks/
Python.framework/Versions/2.5/lib/
python2.5', ...
]
```

Module Search Path

- sys.path contains search path
- Can manually adjust if you need to

```
import sys
sys.path.append("/project/foo/pyfiles")
```

- Paths also added via environment variables

```
% env PYTHONPATH=/project/foo/pyfiles python
Python 2.4.3 (#1, Apr  7 2006, 10:54:33)
[GCC 4.0.1 (Apple Computer, Inc. build 5250)] on darwin
>>> import sys
>>> sys.path
['','/project/foo/pyfiles', '/Library/Frameworks ...]
```


Modules as Objects

- When you import a module, the module itself is a kind of "object"
- You can assign it to variables, place it in lists, change it's name, and so forth

```
import math

m = math          # Assign to a variable
x = m.sqrt(2)     # Access through the variable
```

- You can even store new things in it

```
math.twopi = 2*math.pi
```

What is a Module?

- A module is actually just a thin-layer over a dictionary (which holds all of the contents)

```
>>> import stockfunc
>>> stockfunc.__dict__.keys()
['read_portfolio','read_prices','portfolio_value',
 '__builtins__','__file__','__name__', '__doc__']
>>> stockfunc.__dict__['read_prices']
<function read_prices at 0x69230>
>>> stockfunc.read_prices
<function read_prices at 0x69230>
>>>
```

- Any time you reference something in a module, it's just a dictionary lookup

import as statement

- Changing the name of the loaded module

```
# bar.py
import stockfunc as sf

portfolio = sf.read_portfolio("portfolio.dat")
```

- This is identical to import except that a different name is used for the module object
- The new name only applies to this one source file (other modules can still import the library using its original name)

from module import

- Selected symbols can be lifted out of a module and placed into the caller's namespace

```
# bar.py
from stockfunc import read_prices

prices = read_prices("prices.dat")
```

- This is useful if a name is used repeatedly and you want to reduce the amount of typing
- And, it also runs faster if it gets used a lot

from module import *

- Takes all symbols from a module and places them into the caller's namespace

```
# bar.py
from stockfunc import *

portfolio = read_portfolio("portfolio.dat")
prices = read_prices("prices.dat")
...
```

- As a general rule, this form of import should be avoided because you typically don't know what you're getting as a result
- Namespaces are good

Part 3

Standard Library Tour

Python Standard Library

- Python comes with several hundred modules
  - Text processing/parsing
  - Files and I/O
  - Systems programming
  - Network programming
  - Internet
  - Standard data formats
- Let's take a little tour

sys module

- Information related to environment
- Version information
- System limits
- Command line options
- Module search paths
- Standard I/O streams

sys: Command Line Opts

- Parameters passed on Python command line

```
sys.argv
```

- Example:

```
# opt.py
import sys
print sys.argv
```

```
% python opt.py -v 3 "a test" foobar.txt
['opt.py', '-v', '3', 'a test', 'foobar.txt']
%
```

- Other libraries are available for parsing options (e.g., getopt, optparse).

sys: Standard I/O

- Standard I/O streams

```
sys.stdout
sys.stderr
sys.stdin
```

- By default, print is directed to sys.stdout
- Input read from sys.stdin
- Can redefine or use directly

```
sys.stdout = open("out.txt","w")
print >>sys.stderr, "Warning. Unable to connect"
```


math module

- Contains common mathematical functions

```
math.sqrt(x)
math.sin(x)
math.cos(x)
math.tan(x)
math.atan(x)
math.log(x)
...
math.pi
math.e
```

- Example:

```
import math
c = 2*math.pi*math.sqrt((x1-x2)**2 + (y1-y2)**2)
```

os Module

- Contains operating system functions
- Example: Executing a system command

```
import os
os.system("mkdir temp")
```

- Walking a directory tree

```
for path,dirs,files in os.walk("/home"):
     for name in files:
          print "%s/%s" % (path,name)
```


Environment Variables

- Environment variables (typically set in shell)

```
% setenv NAME dave
% setenv RSH ssh
% python prog.py
```

- os.environ dictionary contains values

```
import os
home = os.environ['HOME']
os.environ['HOME'] = '/home/user/guest'
```

- Changes are reflected in Python and any subprocesses created later

Getting a Directory Listing

- os.listdir() function

```
>>> files = os.listdir("/some/path")
>>> files
['foo','bar','spam']
>>>
```

- glob module

```
>>> txtfiles = glob.glob("*.txt")
>>> datfiles = glob.glob("Dat[0-5]*")
>>>
```

- glob understands Unix shell wildcards (on all systems)

os.path Module

- Portable management of path names and files
- Examples:

```
>>> import os.path
>>> os.path.basename("/home/foo/bar.txt")
'bar.txt'
>>> os.path.dirname("/home/foo/bar.txt")
'/home/foo'
>>> os.path.join("home","foo","bar.txt")
'home/foo/bar.txt'
>>>
```

File Tests

- Testing if a file exists

```
>>> os.path.exists("foo.txt")
True
>>>
```

- Testing if a filename is a regular file

```
>>> os.path.isfile("foo.txt")
True
>>> os.path.isfile("/usr")
False
>>>
```

- Testing if a filename is a directory

```
>>> os.path.isdir("foo.txt")
False
>>> os.path.isdir("/usr")
True
>>>
```


File Metadata

- Getting the file size

```
>>> os.path.getsize("foo.txt")
1344L
>>>
```

- Getting the last modification/access time

```
>>> os.path.getmtime("foo.txt")
1175769416.0
>>> os.path.getatime("foo.txt")
1175769491.0
>>>
```

- Note: To decode times, use time module

```
>>> time.ctime(os.path.getmtime("foo.txt"))
'Thu Apr  5 05:36:56 2007'
>>>
```

Shell Operations (shutil)

- Copying a file

```
shutil.copy("source","dest")
```

- Moving a file (renaming)

```
shutil.move("old","new")
```

- Copying a directory tree

```
shutil.copytree("srcdir","destdir")
```

- Removing a directory tree

```
shutil.rmtree("dir")
```


Subprocesses

- subprocess module can launch other programs and collect their output

```
import subprocess
p = subprocess.Popen(['ls','-l'],
                     stdout=subprocess.PIPE)
result = p.stdout.read()
p.wait()
```

- This is a very powerful module that provides many options and which is cross platform (Unix/Windows)

re Module

- Regular expression pattern matching
- Example : Extract all dates of the form MM/DD/YYYY from this string:

```
"Guido will be away from 12/5/2012 to 1/10/2013."
```

- To do this, you first need a regex pattern

```
r'(\d+)/(\d+)/(\d+)'
```

- Typically written out as a Python raw string (which leave \ characters intact)

re Compilation

- Regex patterns then have to be compiled

```
import re
date_pat = re.compile(r'(\d+)/(\d+)/(\d+)'
```

- The resulting object (date_pat) is something that you use to perform common operations
  - Matching/searching
  - Pattern replacement

Finding all Matches

- How to find and print all matches

```
text = "Guido will be away from 12/5/2012 to 1/10/2013."

for m in datepat.finditer(text):
    print m.group()
```

- This produces the following

```
12/5/2012
1/10/2013
```


Accessing Pattern Groups

- Regular expressions may define groups ()

```
pat = r'(\d+)/(\d+)/(\d+)'
```

- Groups are assigned numbers

```
pat = r'(\d+)/(\d+)/(\d+)'
          ↑     ↑     ↑
          1     2     3
```

- Here's how to find and extract groups

```
for m in datepat.finditer(text):
    month = m.group(1)
    dat   = m.group(2)
    year  = m.group(3)
    ...
```

Pattern Replacement

- Change all dates into DD/MM/YYYY

```
newtext = date_pat.sub(r'\2/\1/\3',text)
```

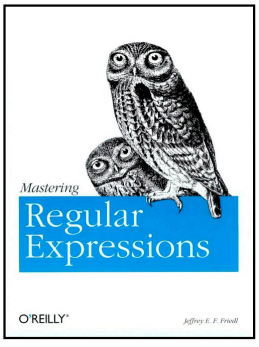
- Change dates to have a month name

```
mon_names = [None, 'Jan','Feb','Mar','Apr','May','Jun',
                   'Jul','Aug','Sep','Oct','Nov','Dec']

def fix_date(m):
    month = mon_names[int(m.group(1)]
    day = m.group(2)
    year = m.group(3)
    return "%s %s, %s" % (month,day,year)

newtext = date_pat.sub(fix_date,text)
```


re: Comments

- re module is very powerful
- I have only covered the essential basics
- Strongly influenced by Perl
- However, regexs are not an operator
- Reference:

Jeffrey Friedl, "Mastering Regular Expressions", O'Reilly & Associates, 2006.



Fetching Remote Data

- urllib and urllib2 modules

```
>>> import urllib
>>> u = urllib.urlopen("http://www.python.org")
>>> data = u.read()
>>>
```

- This opens a URL (http, https, ftp, file) and gives you a file-like object for reading the corresponding data
- Has optional features for submitting forms, emulating different user-agents, etc.

Fetching Remote Data

- Example : Here's a function to access the Chicago Transit Authority bus predictor...

```
import urllib
busurl = "http://ctabustracker.com/bustime/"\
         "map/getStopPredictions.jsp"

def bus_predictions(route,stop):
    fields = {'route': route,
              'stop': stop }
    parms = urllib.urlencode(fields)
    u = urllib.urlopen(busurl+"?"+parms)
    return u.read()
```

Fetching Remote Data

- Example use: Find out how long tourists trying to find Obama's house will have to wait for the #6 bus to go back downtown

```
>>> print bus_predictions(6,5037)
<?xml version="1.0"?>
<stop>
<id>5037</id>
   <nm><![CDATA[Lake Park & E. Hyde Park Blvd]]></nm>
   <sri>
      <rt><![CDATA[6]]></rt>
      <d><![CDATA[North Bound]]></d>
   </sri>
   <cr>6</cr>
   <pre>
      <pt>APPROACHING</pt>
      <fd><![CDATA[Wacker/Columbus]]></fd>
   ...
```


Internet Data Handling

- Library modules provide support for parsing common data formats
- Examples:
  - CSV
  - HTML
  - XML
  - JSON
- Will show an example of XML parsing...

XML Parsing: ElementTree

- Parse the XML bus prediction data and return a simple list of prediction times

```
import urllib
from xml.etree.ElementTree import parse

busurl = "http://ctabustracker.com/bustime/"\
         "map/getStopPredictions.jsp"

def bus_predictions(route,stop):
    fields = {'route': route,
              'stop': stop }
    parms = urllib.urlencode(fields)
    u = urllib.urlopen(busurl+"?"+parms)
    doc = parse(u)     # Parse XML
    predictions = doc.findall("//pre/pt")
    return [p.text for p in predictions]
```

- Three lines of extra code!

Fetching Remote Data

- Example use: Find out how long tourists trying to find Obama's house will have to wait for the #6 bus to go back downtown

```
>>> bus_predictions(6,5037)
['APPROACHING', '5 MIN', '12 MIN', '23 MIN']
>>>
```

- Commentary : gathering data, dealing with different data formats, and processing results is usually pretty straightforward

socket Module

- Provides access to low-level networking
- A simple TCP server (Hello World)

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello %s\n" % a[0])
    c.close()
```

- API is similar to sockets in other languages, but significantly less grungy

Application Protocols

- The Python library provides support for a range of different application protocols
- For example, this is a stand-alone web server

```
from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler
import os
os.chdir("/home/docs/html")
serv = HTTPServer(("",8080),SimpleHTTPRequestHandler)
serv.serve_forever()
```

- Connect with a browser and try it out

Interlude

- Functions and modules form the foundation of a lot of Python programming
- The final step is to define your own objects
- Object-oriented programming

Part 4

Creating New Kinds of Objects

Using Objects

- In a nutshell, object oriented programming is largely about how to bind data and functions together (i.e., packaging)
- You have already been using objects
- For example : strings

```
>>> s = "Hello World"
>>> s.upper()
'HELLO WORLD'
>>> s.split()
['Hello', 'World']
>>>
```

- A string has data (text), but also methods that manipulate that data

Defining New Objects

- You can define your own custom objects
- Use the class statement

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
    def value(self):
        return self.shares * self.price
    def sell(self,nshares):
        self.shares -= nshares
```

- It's just a collection of functions (usually)

Using an Object

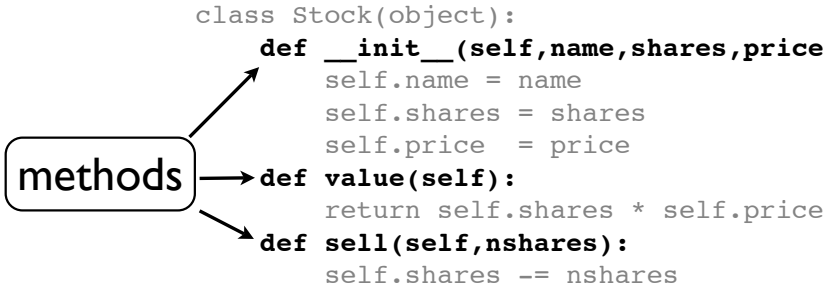
- Creating an object and calling methods

```
>>> s = Stock('GOOG',100,490.10)
>>> s.name
'GOOG'
>>> s.shares
100
>>> s.value()
49010.0
>>> s.sell(25)
>>> s.shares
75
```

- You'll notice how your new object works just like other objects (e.g., calling methods, etc.)

Classes and Methods

- A class is a just a collection of "methods"
- Each method is just a function



- Note: these are just regular Python function definitions (so nothing magical)

Creating Instances

- To create instances, use the class as a function
- This calls __init__() (Initializer)

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
```

```
>>> s = Stock('GOOG',100,490.10)
>>> print s
<__main__.Stock object at 0x6b910>
>>>
```


Instance Data

- Each instance typically holds some state
- Created by assigning attributes on self

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
    def value(self):
        return self.shares * self.price
    def sel
        sel
```

Instance data

```
>>> s = Stock('GOOG',100,490.10)
>>> s.name
'GOOG'
>>> s.shares
100
>>>
```

Methods and Instances

- Methods always operate on an "instance"
- Passed as the first argument (self)

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
    def value(self):
        return self.shares * self.price
    def sell(self,nshares):
        self.shares -= nshares
```

- "self" is where the data associated with each instance is located

Calling Methods

- Methods are invoked on an instance
- Instance is passed as first parameter

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
    def value(self):
        return self.shares * self.price
    def sell(self,nshares):
        self.shares -= nshares
```

```
>>> s = Stock('GOOG',100,490.10)
>>> s.value()
49010.0
>>> s.sell(50)
>>>
```

Special Methods

- Classes can optionally hook into operators and other parts of the language by defining so-called "special methods"

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
```

```
__init__      Initialize an object
__str__       Make a string for printing
__repr__      Make a string representation
__getitem__   self[n]
__setitem__   self[n] = value
__len__       len(self)
__getattr__   self.attr
__setattr__   self.attr = value
...
```

Can completely customize object behavior

Playing with Special Methods

- If you use dir() on built-in objects, you will see a lot of the special methods defined

```
>>> x = 42
>>> dir(x)
['__abs__', '__add__', '__and__', '__class__', '__cmp__',
'__coerce__', '__delattr__', '__div__', '__divmod__',
'__doc__', '__float__', '__floordiv__', '__format__',
'__getattribute__', '__getnewargs__', '__hash__',
'__hex__', '__index__', '__init__', '__int__',
...]
>>> x.__add__(10)
52
>>> x.__str__()
'52'
>>> x.__hex__()
'0x2a'
>>>
```

If you're inclined, you can call them directly (although you wouldn't normally do this in most programs)

Special Method Example

- Fixing the text representation of an object

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
    def value(self):
        return self.shares * self.price
    def sell(self,nshares):
        self.shares -= nshares
    def __repr__(self):
        return "Stock('%s',%d,%0.2f)" % \
            (self.name,self.shares,self.price)
```

- Example

```
>>> s = Stock('GOOG',100,490.10)
>>> print s
Stock('GOOG',100,490.10)
>>>
```


Inheritance

- A tool for specializing objects

```
class Parent(object):
    ...

class Child(Parent):
    ...
```

- New class called a derived class or subclass
- Parent known as base class or superclass
- Parent is specified in () after class name

Inheritance

- What do you mean by "specialize?"
- Take an existing class and ...
    - Add new methods
    - Redefine some of the existing methods
    - Add new attributes to instances

Inheritance Example

- Suppose you wanted to make a special Stock object for Bernie Madoff
- A stock holding whose reported "value" apparently never goes down
- Sign me up now!

Inheritance Example

- Specializing a class

```
class MadoffStock(Stock):
    def __init__(self,name,shares,price):
        Stock.__init__(self,name,shares,price)
        self.maxprice = price
    def value(self):
        if self.price > self.maxprice:
            self.maxprice = self.price
        return self.shares * self.maxprice
    def actualvalue(self):
        return Stock.value(self)
```

- This new version inherits from Stock and
  - adds a new attribute (maxprice)
  - redefines an existing method (value)
  - adds a new method (actualvalue)

Inheritance Example

- It works the same as before except with some new behavior and methods

```
>>> s = MadoffStock('GOOG',100,490.10)
>>> s.value()
49010.0
>>> s.price = 210.03
>>> s.value()          # Using modified value()
49010.0
>>> s.actualvalue()
21003.0
>>>
```

Inheritance and __init__

- If a subclass defines __init__, it must call the parent __init__ (if there is one)

```
class Stock(object):
    def __init__(self,name,shares,price):
        ...

class MadoffStock(Stock):
    def __init__(self,name,shares,price):
        Stock.__init__(self,name,shares,price)
        self.maxprice = price
    def value(self):
        if self.price > self.maxprice:
            self.maxprice = self.price
        return self.shares * self.maxprice
    def actualvalue(self):
        return Stock.value(self)
```


Calling Parent Methods

- To call original methods in the parent, just use the class name explicitly

```
class Stock(object):
    def value(self):
        ...

class MadoffStock(Stock):
    ...
    def value(self):
        if self.price > self.maxprice:
            self.maxprice = self.price
        return self.shares * self.maxprice
    def actualvalue(self):
        return Stock.value(self)
```

- Only necessary if a subclass is redefining one of the parent methods you want to call

object base class

- If a class has no parent, use object as base

```
class Foo(object):
    ...
```

- object is the parent of all objects in Python
- Note : Sometimes you will see code where classes are defined without any base class. That is an older style of Python coding that has been deprecated for almost 10 years. When defining a new class, you always inherit from something.

Multiple Inheritance

- You can specify multiple base classes

```
class Foo(object):
    ...
class Bar(object):
    ...
class Spam(Foo,Bar):
    ...
```

- The new class inherits features from both parents
- Rule of thumb : Don't do this
- Multiple inheritance in Python is just as evil as it is in other programming languages with this feature

Inheritance in the Library

- Many of Python's library modules require you to define objects that inherit from a base class and implement some set of expected methods
- An example : Thread programming

threading

- A library module for thread programming

```
import threading
import time

class ChildThread(threading.Thread):
    def run(self):
        while True:
             time.sleep(15)
             print "Are we there yet?"

child = ChildThread()   # Create a thread
child.start()           # Launch it.  This executes run()
```

- To use, you inherit and implement run()
- Obviously, omitting some further details here

Interlude

- Believe it or not, we've covered most of the Python object system
    - Classes are a collection of methods
    - Class instances hold data
    - Classes can inherit from other classes
- Python doesn't have a lot of other OO baggage
    - Access control (public, private, etc.)
    - Templates/generics

Part 5

Understanding Python Objects

Dictionaries Revisited

- A dictionary is a collection of named values

```
stock = {
          'name'   : 'GOOG',
          'shares' : 100,
          'price'  : 490.10
        }
```

- Dictionaries are commonly used for simple data structures (shown above)
- This is actually rather similar to what you get when you manipulate a class instance

Dicts and Objects

- User-defined objects are built using dicts
  - Instance data
  - Class members
- In fact, the entire object system is mostly just an extra layer that's put on top of dictionaries
- Let's take a look...

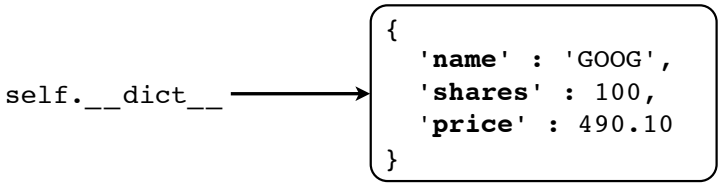
Dicts and Instances

- A dictionary holds instance data (__dict__)

```
>>> s = Stock('GOOG',100,490.10)
>>> s.__dict__
{'name' : 'GOOG','shares' : 100, 'price': 490.10 }
```

- You populate this dict when assigning to self

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
```



instance data

Dicts and Instances

- Critical point : Each instance gets its own private dictionary

```
s = Stock('GOOG',100,490.10)
t = Stock('AAPL',50,123.45)
```

```
{
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.10
}
```

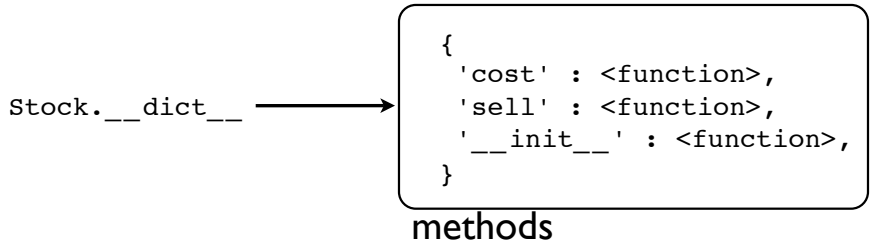
```
{
    'name' : 'AAPL',
    'shares' : 50,
    'price' : 123.45
}
```

- So, if you created 100 instances of some class, there are 100 dictionaries sitting around holding data

Dicts and Classes

- A dictionary holds the members of a class

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
    def cost(self):
        return self.shares*self.price
    def sell(self,nshares):
        self.shares -= nshares
```



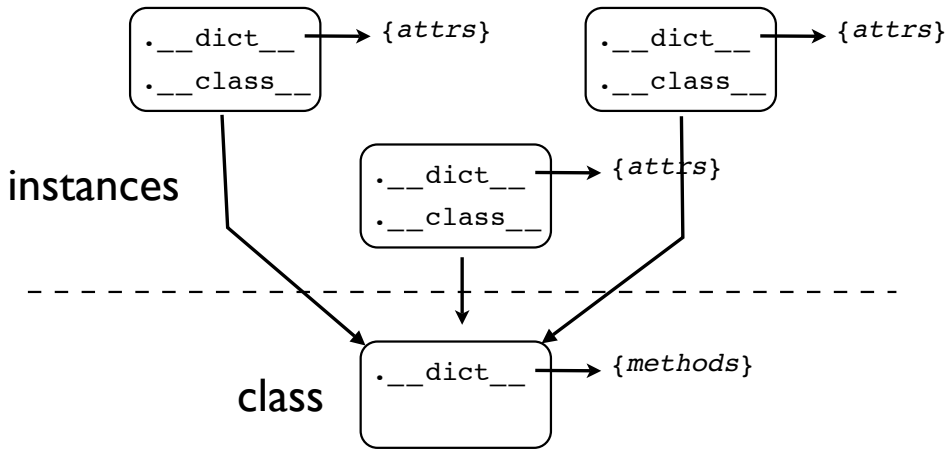
Instances and Classes

- Instances and classes are linked together
- __class__ attribute refers back to the class

```
>>> s = Stock('GOOG',100,490.10)
>>> s.__dict__
{'name':'GOOG','shares':100,'price':490.10 }
>>> s.__class__
<class '__main__.Stock'>
>>>
```

- The instance dictionary holds data unique to each instance whereas the class dictionary holds data collectively shared by all instances

Instances and Classes



Attribute Access

- When you work with objects, you access data and methods using the (.) operator

```
x = obj.name      # Getting
obj.name = value  # Setting
del obj.name      # Deleting
```

- These operations are directly tied to the dictionaries sitting underneath the covers

Modifying Instances

- Operations that modify objects always update the underlying dictionary

```
>>> s = Stock('GOOG',100,490.10)
>>> s.__dict__
{'name':'GOOG', 'shares':100, 'price':490.10 }
>>> s.shares = 50
>>> s.date = "6/7/2007"
>>> s.__dict__
{ 'name':'GOOG', 'shares':50, 'price':490.10,
  'date':'6/7/2007'}
>>> del s.shares
>>> s.__dict__
{ 'name':'GOOG', 'price':490.10,  'date':'6/7/2007'}
>>>
```


Modifying Instances

- It may be surprising that instances can be extended after creation
- You can freely change attributes at any time

```
>>> s = Stock('GOOG',100,490.10)
>>> s.blah = "some new attribute"
>>> del s.name
>>>
```

- Again, you're just manipulating a dictionary
- Very different from C++/Java where the structure of an object is rigidly fixed

Reading Attributes

- Suppose you read an attribute on an instance

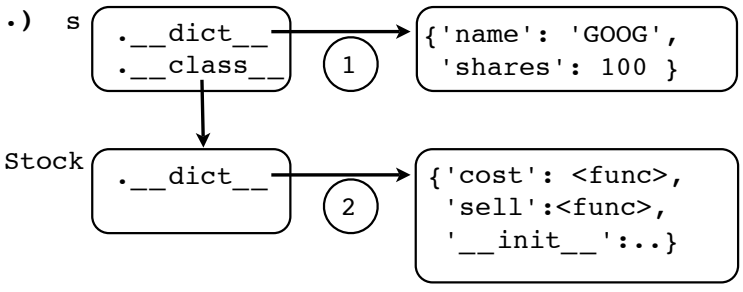
```
x = obj.name
```

- Attribute may exist in two places
  - Local instance dictionary
  - Class dictionary
- So, both dictionaries may be checked

Reading Attributes

- First check in local __dict__
- If not found, look in __dict__ of class

```
>>> s = Stock(...)
>>> s.name
'GOOG'
>>> s.cost()
49010.0
>>>
```



- This lookup scheme is how the members of a class get shared by all instances

How Inheritance Works

- Classes may inherit from other classes

```
class A(B,C):
   ...
```

- Bases are stored as a tuple in each class

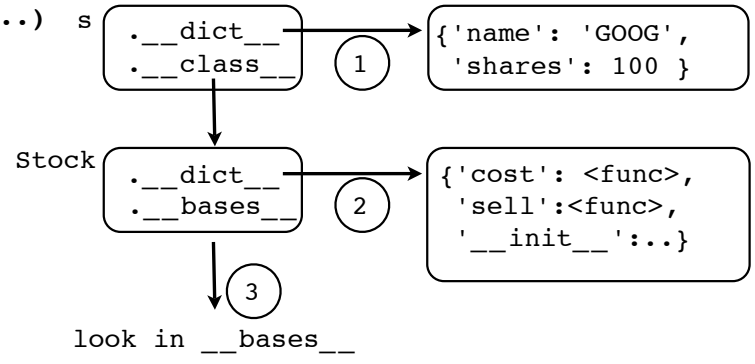
```
>>> A.__bases__
(<class '__main__.B'>,<class '__main__.C'>)
>>>
```

- This provides a link to parent classes
- This link simply extends the search process used to find attributes

Reading Attributes

- First check in local __dict__
- If not found, look in __dict__ of class
- If not found in class, look in base classes

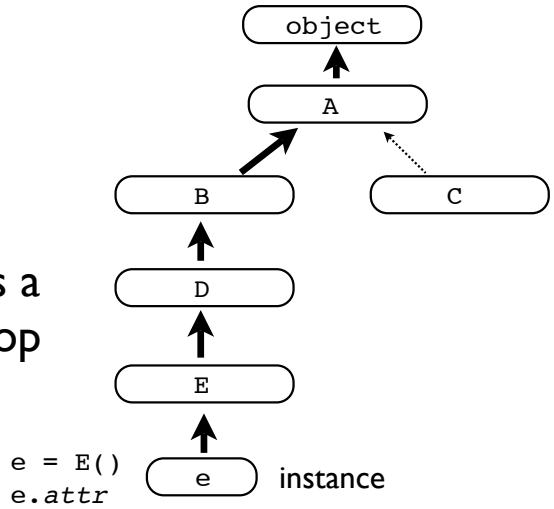
```
>>> s = Stock(...)
>>> s.name
'GOOG'
>>> s.cost()
49010.0
>>>
```



Single Inheritance

- In inheritance hierarchies, attributes are found by walking up the inheritance tree

```
class A(object): pass
class B(A): pass
class C(A): pass
class D(B): pass
class E(D): pass
```

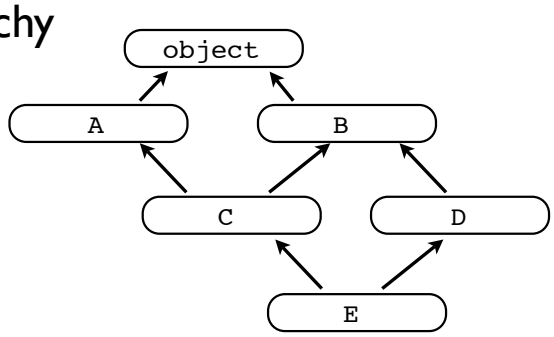


- With single inheritance, there is a single path to the top
- You stop with the first match

Multiple Inheritance

- Consider this hierarchy

```
class A(object): pass
class B(object): pass
class C(A,B): pass
class D(B): pass
class E(C,D): pass
```



- What happens here?

```
e = E()
e.attr
```

- A similar search process is carried out, but there is an added complication in that there may be many possible search paths

Multiple Inheritance

- For multiple inheritance, Python determines a "method resolution order" that sets the order in which base classes get checked

```
>>> E.__mro__
(<class '__main__.E'>,  <class '__main__.C'>,
 <class '__main__.A'>,  <class '__main__.D'>,
 <class '__main__.B'>,  <type 'object'>)
>>>
```

- The MRO is described in a 20 page math paper (C3 Linearization Algorithm)
- Note: This complexity is one reason why multiple inheritance is often avoided

Part 6

Some Encapsulation Tools

Classes and Encapsulation

- One of the primary roles of a class is to encapsulate data and internal implementation details of an object
- However, a class also defines a "public" interface that the outside world is supposed to use to manipulate the object
- This distinction between implementation details and the public interface is important

A Problem

- In Python, almost everything about classes and objects is "open"
  - You can easily inspect object internals
  - You can change things at will
  - There's no strong notion of access-control (i.e., private class members)
- If you're trying to cleanly separate the internal "implementation" from the "interface" this becomes an issue

Python Encapsulation

- Python relies on programming conventions to indicate the intended use of something
- Typically, this is based on naming
- There is a general attitude that it is up to the programmer to observe conventions as opposed to having the language enforce rules

Private Attributes

- Any attribute name with leading __ is "private"

```
class Foo(object):
   def __init__(self):
       self.__x = 0
```

- Example

```
>>> f = Foo()
>>> f.__x
AttributeError: 'Foo' object has no attribute '__x'
>>>
```

- This is actually just a name mangling trick

```
>>> f = Foo()
>>> f._Foo__x
0
>>>
```

Private Methods

- Private naming also applies to methods

```
class Foo(object):
   def __spam(self):
         print "Foo.__spam"
   def callspam(self):
         self.__spam()     # Uses Foo.__spam
```

- Example:

```
>>> f = Foo()
>>> f.callspam()
Foo.__spam
>>> f.__spam()
AttributeError: 'Foo' object has no attribute '__spam'
>>> f._Foo__spam()
Foo.__spam
>>>
```


Commentary

- If you are reading other people's Python code, it is generally understood that names starting with underscores are internal implementation
- If you're writing code that uses a library, you're not supposed to use those names
- However, Python won't prevent access
- There are many tools that need to inspect objects (debuggers, testing, profiling, etc.)

Accessor Methods

- In OO, it's often advised to use accessors

```
class Foo(object):
    def __init__(self,name):
        self.__name = name
    def getName(self):
        return self.__name
    def setName(self,name):
        if not isinstance(name,str):
             raise TypeError("Expected a string")
        self.__name = name
```

- Methods give you more flexibility and may become useful as your program grows
- Allows extra logic to be added later on without breaking everything

Properties

- Python has a mechanism to turn accessor methods into "property" attributes

```
class Foo(object):
    def __init__(self,name):
        self.__name = name
    def getName(self):
        return self.__name
    def setName(self,name):
        if not isinstance(name,str):
            raise TypeError("Expected a string")
        self.__name = name
    name = property(getName,setName)
```

- Properties look like data attributes, but implicitly call the accessor methods.

Properties

- Example use:

```
>>> f = Foo("Elwood")
>>> f.name               # Calls f.getName()
'Elwood'
>>> f.name = 'Jake'      # Calls f.setName('Jake')
>>> f.name = 45          # Calls f.setName(45)
TypeError: Expected a string
>>>
```

- Comment : With properties, you can hide extra processing behind access to data attributes--something that is useful in certain settings
- Example : Type checking, lazy evaluation, etc.

Properties

- Properties are also useful if you are creating objects where you want to have a very consistent user interface
- Example : Computed data attributes

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
    def value(self):
        return self.shares * self.price
    value = property(value)
    def sell(self,nshares):
        self.shares -= nshares
```

Properties

- Example use:

```
>>> s = Stock('GOOG',100,490.10)
>>> s.shares
100
>>> s.value
49010.0
>>>
```

Instance Variable

Computed Property

- Commentary : Notice how there is no obvious difference between the attributes as seen by the user of the object

Properties

- It is very common for a property definition to replace a method of the same name

Notice the identical names

```
class Stock(object):
    ...
    def value(self):
        return self.shares * self.price
    value = property(value)
    ...
```

- When you define a property, you usually don't want the user to explicitly call the method
- So, this trick hides the method and only exposes the property attribute

Decorators

- There is an alternative syntax for this code

```
class Stock(object):
    ...
    def value(self):
        return self.shares * self.price
    value = property(value)
    ...
```

- Use property as a "decorator"

```
class Stock(object):
    ...
    @property
    def value(self):
        return self.shares * self.price
    ...
```

- This syntax specifies that the function to follow gets processed through property()

Commentary

- With properties, you have the ability to add "hidden processing" to objects
- Users of an object generally won't know anything about this extra functionality
- This kind of programming is pretty common in Python
- Especially when combined with special methods that hook into various operators

Lower-level Primitives

- Various facets of Python objects can be customized using lower-level primitives
  - Descriptor objects
  - Attribute binding methods

Descriptors

- A descriptor object

```
class Descriptor(object):
    def __get__(self,instance,cls):
        ...
    def __set__(self,instance,value):
        ...
    def __delete__(self,instance):
        ...
```

- This defines a special object that captures attribute access when its part of a class

```
class Foo(object):
    x = Descriptor()
    y = Descriptor()

f = Foo()
f.x = 3     # Calls f.x.__set__()
print f.x   # Calls f.x.__get__()
```

Attribute Access

- Class can also redefine the (.) operation

```
class Foo(object):
    def __getattr__(self,name):
        ...
    def __setattr__(self,name,value):
        ...
    def __delattr__(self,name)
        ...
```

- If you define any of these, an object can completely change the semantics of attribute binding

Part 7

Metaprogramming

Metaprogramming

- Functions and classes are objects that you can freely manipulate as data

```
def add(x,y):
    return x + y

>>> a = add
>>> a(3,4)
7
>>> d = {'+' : add }
>>> d['+'](3,4)
7
>>>
```

- You can assign to variables, place in data structures, and use just like numbers, strings, or other primitive datatypes

Metaprogramming

- Because you can manipulate the basic elements that make up a program (functions, classes), it is possible to write programs that manipulate their own internal structure
- For example, you can write functions that manipulate other functions
- This is known as "metaprogramming"

Interlude

- It turns out that metaprogramming is one of Python's most powerful features
- It is used a lot by framework builders
- Can be used to create highly-customized domain-specific programming environments
- This is an advanced aspect of Python, but we'll take a look at a couple of examples

Passing Functions

- A function can accept a function as input:

```
def callf(func,x,y):
    print "Calling", func.__name__
    return func(x,y)
```

- Usage:

```
>>> def add(x,y):
...     return x+y
>>> callf(add,3,4)
Calling add
7
>>>
```

- The above code illustrates a "wrapper"

Parameter Passing

- There is a special form of parameter passing that gets used a lot with wrappers

```
def callf(func,*args,**kwargs):
    print "Calling", func.__name__
    return func(*args, **kwargs)
```

- *args - Collects all positional args in a tuple
- **kwargs - Collects keyword args in a dict
- func(*args,**kwargs) - Passes arguments along
- You'll find that the above code now works with any passed function whatsoever

Adding Layers

- Functions that wrap other functions turn out to be a powerful tool for certain kinds of applications
- Example : Logging

```
def logged(func,*args,**kwargs):
    log.debug("Calling %s", func.__name__)
    return func(*args, **kwargs)
```

- Example : Synchronization

```
flock = threading.Lock()
def synchronized(func,*args,**kwargs):
    flock.acquire()
    try:
        return func(*args,**kwargs)
    finally:
        flock.release()
```

Example Use

- How these wrappers might get used

```
# Some function
def foo(x,y,z):
    ...

# Calling a wrapper
r = logged(foo,1,2,3)
r = synchronized(foo,1,2,3)
```

- Comment : A little hacky, but we'll see a cleaner approach in a second

Returning Functions

- A function can create and return a new function

```
def make_adder(x,y):
    def add():
        return x+y
    return add
```

- Usage:

```
>>> a = make_adder(3,4)
>>> a
<function add at 0x7e6b0>
>>> a()
7
>>>
```

- The returned function carries information about the environment in which it was defined (known as a "closure")

Making Wrappers

- You can write functions that create wrapper functions and return them as a result

```
def debug(func):
    def call(*args,**kwargs):
        print "Calling", func.__name__
        return func(*args, **kwargs)
    return call
```

- Example:

```
>>> def add(x,y):
...     return x+y
>>> d_add = debug(add)
>>> d_add(3,4)
Calling add
7
>>>
```


Making Wrappers

- These new wrappers create a function that mirrors the original functions calling signature
- Because of this, the new function can serve as a stand-in for the original function

```
def add(x,y):
    return x+y

# Put a wrapper around it and reassign add
add = debug(add)

>>> add(3,4)
Calling add
7
>>>
```

Decorators

- Wrapping a function is known as "decoration"
- There is a special syntax for doing it

```
@debug
def add(x,y):
    return x+y
```

- It means the same thing as this:

```
def add(x,y)
    return x+y
add = debug(add)       # Wrap it
```

- This aspect of Python is something that's relatively new, but showing up more often in library modules

Metaclasses

- It is also possible to completely customize what happens when classes get defined
- This is an extremely advanced topic
- Due to time constraints, I'm going to skip it
- Bottom line : Using decorators and metaclasses, it is possible to customize the Python environment in almost any way that you can imagine

Part 8

Python Integration

Integration

- Python is commonly integrated with software written in other programming languages
- For example, C, C++, Java, C#, etc.
- In this role, it often serves as a scripting language for low-level components
- Let's briefly take a look at some of this...

ctypes Module

- A standard library module that allows C functions to be executed in arbitrary shared libraries/DLLs
- It's included as part of Python-2.5
- One of several approaches that can be used to access foreign C functions

ctypes Example

- Consider this C code:

```
int fact(int n) {
   if (n <= 0) return 1;
   return n*fact(n-1);
}

int cmp(char *s, char *t) {
   return strcmp(s,t);
}
double half(double x) {
   return 0.5*x;
}
```

- Suppose it was compiled into a shared lib

```
% cc -shared example.c -o libexample.so
```

ctypes Example

- Using C types

```
>>> import ctypes
>>> ex = ctypes.cdll.LoadLibrary("./libexample.so")
>>> ex.fact(4)
24
>>> ex.cmp("Hello","World")
-1
>>> ex.cmp("Foo","Foo")
0
>>>
```

- It just "works" (through heavy wizardry)

ctypes Example

- Well, it *almost* works:

```
>>> import ctypes
>>> ex = ctypes.cdll.LoadLibrary("./libexample.so")
>>> ex.fact("Howdy")
1
>>> ex.cmp(4,5)
Segmentation Fault

>>> ex.half(5)
-1079032536
>>> ex.half(5.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <type
'exceptions.TypeError'>: Don't know how to convert
parameter 1
>>>
```

ctypes Internals

- ctypes is a module that implements a foreign function interface (FFI)
- However, C libraries don't encode any information about type signatures (parameters or return types)
- So, ctypes initially assumes that all functions operate on integers (int) or character strings (char *)

Adding Type Declarations

- ctypes *can* handle other C datatypes
- You just have to provide type information

```
>>> ex.half.argtypes = (ctypes.c_double,)
>>> ex.half.restype = ctypes.c_double
>>> ex.half(5.0)
2.5
>>>
```

- This creates a minimal prototype

```
.argtypes      # Tuple of argument types
.restype       # Return type of a function
```

ctypes Types

- A sampling of datatypes that are available

```
ctypes type        C Datatype
------------------ --------------------------
c_byte             signed char
c_char             char
c_char_p           char *
c_double           double
c_float            float
c_int              int
c_long             long
c_longlong         long long
c_short            short
c_uint             unsigned int
c_ulong            unsigned long
c_ushort           unsigned short
c_void_p           void *
```

- You can also build arrays, structs, pointers, etc.

ctypes Cautions

- Requires detailed knowledge of underlying C library and how it operates
- Function names
- Argument types and return types
- Data structures
- Side effects/Semantics
- Memory management

ctypes and C++

- Not really supported
- This is more the fault of C++
- C++ creates libraries that aren't easy to work with (non-portable name mangling, vtables, etc.)
- C++ programs may use features not easily mapped to ctypes (e.g., templates, operator overloading, smart pointers, RTTI, etc.)

Extension Commentary

- There are more advanced ways of extended Python with C and C++ code
- Low-level extension API
- Code generator tools (e.g., Swig, Boost, etc.)

Jython

- A pure Java implementation of Python
- Can be used to write Python scripts that interact with Java classes and libraries
- Official Location:

http://www.jython.org

Jython Example

- Jython runs like normal Python

```
% jython
Jython 2.2.1 on java1.5.0_13
Type "copyright", "credits" or "license" for more
information.
>>>
```

- And it works like normal Python

```
>>> print "Hello World"
Hello World
>>> for i in xrange(5):
...     print i,
...
0 1 2 3 4
>>>
```

Jython Example

- Many standard library modules work

```
>>> import urllib
>>> u = urllib.urlopen("http://www.python.org")
>>> page = u.read()
>>>
```

- And you get access to Java libraries

```
>>> import java.util
>>> d = java.util.Date()
>>> d
Sat Jul 26 13:31:49 CDT 2008
>>> dir(d)
['UTC', '__init__', 'after', 'before', 'class',
'clone', 'compareTo', 'date', 'day', 'equals',
'getClass', 'getDate', ... ]
>>>
```


IronPython

- Python implemented in C#/.NET
- Can be used to write Python scripts that control components in .NET
- Official Location:

  http://www.codeplex.com/IronPython

IronPython Example

- IronPython runs like normal Python

```
% ipy
IronPython 1.1.2 (1.1.2) on .NET 2.0.50727.42
Copyright (c) Microsoft Corporation. All rights
reserved.
>>>
```

- And it also works like normal Python

```
>>> print "Hello World"
Hello World
>>> for i in xrange(5):
...     print i,
...
0 1 2 3 4
>>>
```


IronPython Example

- You get access to .NET libraries

```
>>> import System.Math
>>> dir(System.Math)
['Abs', 'Acos', 'Asin', 'Atan', 'Atan2', 'BigMul',
'Ceiling', 'Cos', 'Cosh', ...]
>>> System.Math.Cos(3)
-0.9899924966
>>>
```

- Can script classes written in C#
- Same idea as with Jython

Final Words

Summary

- This has been a whirlwind tour of Python
- Some things to take away
  - As a scripting language, Python has a lot of powerful tools for processing data, interacting with the system, and interfacing with other software.
  - As a programming language, Python has a lot of support for creating and managing large applications (functions, modules, objects, metaprogramming, etc.)

Thanks!

- I'd just like to thank everyone for attending
- Contact me at http://www.dabeaz.com
- Shameless plug : I teach hands-on Python courses on-site that go into much more detail than this tutorial