An Incremental Introduction to Python

Ralph Noack
Mechanical Engineering Department
University of Alabama at Birmingham

What is Python?

- A programming language named after the BBC show "Monty Python's Flying Circus".
- "Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms." Guido van Rossum

Why use Python?

- Easy to learn: Clear, simple, and elegant syntax make it an easy language to learn.
- Readability: Python's clear and elegant syntax, and emphasis on proper indentation of code blocks, improves code readability
- Conciseness: Programs written in Python are typically much shorter than equivalent C or C++ programs.
- Object oriented framework:
  - Built from the ground up as an object oriented language
- Extensibility: Python makes it easy to add new capabilities to your code
- Portability: Python distributions are available for a wide variety of operating systems
- Open licensing model: Python is available for free

Why Teach Python?

- Quoting Quido:
  - Easy to teach the principles
    - see trees through forest
    - structured programming
    - object-oriented programming
    - programming large systems
  - Interesting, realistic examples
    - connect to real applications
- Simple to learn. Clean syntax.
- Low overhead.
  - Do not have to learn/type as much as other languages
- Has power to do complex tasks.
- Can focus on learning to program without having to learn a complex language.
- Can learn object oriented programming with minimal overhead

Example Java vs Python

- Java

```
public class HelloWorld {
  public static void main( String [] args) {
    System.out.println("Hello, world.");
  }
}
```

- Python

```
print "Hello, world."
```

About this Tutorial

- We will attempt to teach the highlights of the language
- There are many concepts and capabilities that will not be covered
- The intent is to introduce you to python and let you begin to write some python programs
- We'll incrementally introduce syntax and concepts
- Won't cover all the details but will cover enough to get you started
- The full details and rest of the capability can be learned as you continue to use python

Where to get Python

- Web site is http://www.python.org
  - Download latest source
  - Precompiled binaries for easy install are available for Windows
  - Lots of documentation and other useful packages
- Python is installed by default in most Linux distributions.

Python Resources

- http://www.python.org
- http://www.python-eggs.org
- Numerous tutorials/articles on web sites
- Good books for learning Python
  - **Learning Python**, Mark Lutz, David Ascher, ISBN 1565924649
  - **The Quick Python Book**, Daryl D. Harms, Kenneth McDonald, ISBN 1884777740
- Python reference books
  - **Python in a Nutshell**, Alex Martelli, ISBN 0596001886
  - **Python Essential Reference**, David Beazley, ISBN 0735710910
  - **Python Cookbook**, Alex Martelli, David Ascher, ISBN 0596001673
- Tkinter programming
  - **Python and Tkinter Programming**, John E. Grayson, ISBN 1884777813

Running Python

At the command prompt type:

python

Prints version and copyright information and python prompt ">>>"

Python 1.5.2 (#1, Jan 31 2003, 10:58:35) [GCC 2.96 20000731 (Red Hat Linux 7.3 2 on linux-i386

Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam

>>>

Use python as a calculator

```
>>> 2+2
4
>>> # a pound sign # starts a comment, prompt
  will change after it
... 2+3
5
>>> 2.0+3.0
5.0
>>> # python automatically determines if the
  number is an integer or floating point value
... 2.0+3.0
5.0
>>> 2+3.0
5.0
```

Use python as a calculator

```
>>> 2*3 # you can add a comment on the same line
   as code
6
>>> 6/2
3
>>> 6.5/2
3.25
>>> 7/2
3
```

Python has the usual set of math operators

+ for addition
- for subtraction
* for multiplication
/ for division
** for exponentiation
and others

```
>>> 2**3
8
```

- There are precedence rules for mixed expressions
- Use parentheses to force a particular order

```
>>> 1+2*3
7
>>> (1+2)*3
9
```

Command line editing

- You can recall previous commands to use as the current input and edit the current line of input.
- The up arrow or the Control-p key moves up to the previous input line.
- The down arrow or the Control-n key moves down to the next input line.
- The left arrow or the Control-b key moves the cursor to the left on the input line.
- The right arrow or the Control-f key moves the cursor to the left on the input line.

Command line editing

- The backspace key will delete the character to the left of the cursor.
- The Control-d key will delete the character under the cursor.
- The Control-a key will move the beginning of a line.
- The Control-e key will move the end of a line.
- The cursor does not have to be at the end of the line to press the Enter key to accept the input.

Variables

- A variable is a name that can hold a value
- You assign a value to the name on the left of an =
- Case sensitive SPAM, spam, Spam are all different variables
- Names are created when first assigned

```
>>> a=2
>>> b=3
>>> B=6
>>> a+b
5
>>> a+B
8
```

Variables

- Cannot use a name that has not been created/assigned

```
>>> q
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'q' is not defined
```

- Certain words (reserved words) cannot be used as variable names

```
>>> if=2
  File "<stdin>", line 1
    if=2
      ^
SyntaxError: invalid syntax
```

Variables

- Notice that no declaration of variable type is required.
  - Some languages require you to declare a variable and its type

    Fortran: integer i

    real a

    C: int i; float a;
- Python automatically determines the type of a variable
  - Use float(), int() to convert a type

```
>>> a=2.5
>>> b=4
>>> b/int(a)
2
>>> a=3
>>> b=2
>>> a/float(b)
1.5
```

Complex Numbers

- Complex numbers have a real component and an "imaginary" component.
- The imaginary component multiplies j=sqrt(-1)
- In python you can write a complex number as (real+imagj)
- or use the function: complex(real, imag)

```
>>> 1+2j
(1+2j)
>>> 1j*1j
(-1+0j)
>>> (2+3j)*complex(1,2)
(-4+7j)
```

Complex Numbers

- To extract real and imaginary parts from a complex number a, use a.real and a.imag.

```
>>> a=(2+3j)
>>> b=complex(1,2)
>>> a.real
2.0
>>> a.imag
3.0
>>> b.real
1.0
>>> b.imag
2.0
```

Complex Numbers

- Get the magnitude of the complex number with abs() = sqrt(a.real*a.real+a.imag*a.imag)

```
>>> abs(a)
3.605551275463989
>>> abs(b)
2.2360679774997898
```

Strings

- Python strings are enclosed in single ' or double " quotes:

```
>>> "hello world"
'hello world'
>>> 'hello world'
'hello world'
```

- Can assign a string to a variable:

```
>>> hello="Hello World!"
>>> print hello
Hello World!
```

Strings

- Can include special characters in the string.

```
>>> hello="Hello\nWorld!"
>>> print hello
Hello
World!
```

- The usual special characters are:

"\n" newline

"\r" carriage return

"\t" horizontal tab

Long strings

- Enclose them in triple quotes, either double quote character: """ or single quote character: '

```
>>> hello="""This is a long line
... spread across multiple lines
...    notice the indentation/leading white space"""
>>> print hello
This is a long line
spread across multiple lines
   notice the indentation/leading white space
```

Long strings

```
>>> hello="This is another long string\n\
... also spread across multiple lines\n\
... Use the trailing backslash to escape or hide the trailing
    newline\n\
... Need the embedded newlines to get the lines to split"
>>> print hello
This is another long string
also spread across multiple lines
Use the trailing backslash to escape or hide the trailing
    newline
Need the embedded newlines to get the lines to split
```

String concatenation

- Join two strings with the + operator

```
>>> a="first string-"
>>> b='second string'
>>> a+b
'first string-second string'
```

Extracting substrings

- Python strings can be subscripted to extract a substring.
- First character is at index 0
- Length of the string is given by len() function

```
>>> s="123456"
>>> print s[0]
1
>>> print len(s)
6
```

Extracting substrings

- Substring specified using the slice notation: first index:second index

```
>>> s[0:2]
'12'
>>> s[3:5]
'45'
```

Extracting substrings

- Empty first index is the same as 0
- Empty second index is the same as the string length

```
>>> s[0:3]
'123'
>>> s[:3]
'123'
>>> s[3:len(s)]
'456'
>>>
>>> s[3:]
'456'
>>>
```

Extracting substrings

- Can use negative numbers: Counts backwards from the upper bound

```
>>> s[-1]
'6'
>>> s[-2:]
'56'
>>> s[:-4]
'12'
>>> s[-6:-1]
'12345'
```

Extracting substrings

- Think of the slice indices as pointing between characters.
- The left edge of the first character is numbered 0.
- The right edge of the last character of a string of n characters has index n.
- Consider “123456”

```
+---+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Python strings are immutable

- Python strings are immutable (cannot be changed)

```
>>> s='123456'
>>> s[0]='0'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

- Create a new string to achieve the same effect

```
>>> t='0' + s[1:]
>>> print t
023456
```

Changing into a string

- Cannot append a non-string with a string

```
>>> a='a string'
>>> b=5.0
>>> a+b
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in
  operation
```

- Can create a string representation of a non-string by using the str() function

```
>>> a+str(b) + "-"+ str(1.0)
'a string5.0-1.0'
```

Formatting into a string

- You can create a string from other objects with a formatting operation.
- Specify a format string and the list of objects to be used in the formatting:
  - Format % tuple of arguments

```
>>> "an integer %d a floating point number %f" %
   (1,2.0)
'an integer 1 a floating point number 2.000000'
```

Formatting Codes

- The formatting string uses codes to determine where to place the representation of the value
- The code must match the type of object
- Available codes:

%s	String	%X	Hex integer (uppercase)
%c	Character	%e	Floating-point number
%d	Decimal integer	%E	Floating-point number
%i	Integer	%f	Floating-point number
%u	Unsigned Integer	%g	Floating-point number
%o	Octal Integer	%g	Floating-point number
%x	Hex Integer	%%	Literal %

Formatting Codes

```
>>> "%c %c"%('a',100)
'a d'
>>> "%e %E %f %g %G"%(1.0,2.0,3.0,4.0e5,4.0e15)
'1.000000e+00 2.000000E+00 3.000000 400000 4E+15'
>>>"unsigned int of %d is %u"%(-3,-3)
'unsigned int of -3 is 4294967293‘
>>>’for the integer %d octal=%o hex=%x’%(
  123456,123456,123456)
'for the integer 123456 octal=361100 hex=1e240'
```

Python Lists

- The Python list is used to group together other values.
- The list is written as a list of comma-separated values (items) between square brackets.
- List items need not all have the same type.

```
>>> a=[1,2,3,4,5]
>>> print a
[1, 2, 3, 4, 5]
>>> b=[1,'two',3j,"FOUR",5]
>>> print b
[1, 'two', 3j, 'FOUR', 5]
```

Python Lists

- List indices start at 0, can be sliced, and concatenated:

```
>>> print a[0]
1
>>> print a[0:2]
[1, 2]
>>> ab = a[0:2] + b[-3:]
>>> print ab
[1, 2, (1+3j), 'FOUR', 5]
```

Python Lists

- A list can hold any other Python object.
- A list with each item being another list creates a doubly subscripted list/array.

```
>>> c=[[0,1,2],[3,4,5],[6,7,8]]
>>> print c
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> print c[0]
[0, 1, 2]
>>> print c[0][0]
0
>>> print c[1][0]
3
>>> print c[2][0]
6
```

Python Lists are mutable

- Python lists can be changed

```
>>> c=[[0,1,2],[3,4,5],[6,7,8]]
>>> c[1]="python is great"
>>> print c
[[0, 1, 2], 'python is great', [6, 7, 8]]
```

- Lists can be extended by appending a new value to the list

```
>>> c.append("a new item")
>>> print c
[[0, 1, 2], 'python is great', [6, 7, 8], 'a new item']
```

Python Lists are mutable

- Use the insert method to add an item in the middle of a list.
- Index refers to the location between items like slicing.

```
>>> c.insert(2,"and easy to learn")
>>> print c
[[0, 1, 2], 'python is great', 'and easy to learn', [6,
   7, 8], 'a new item']
```

- The len() function gives you the length of the list.

```
>>> print len(c)
5
```

- There are other functions that will operate on a list:
  - del, sort(), reverse(), and others

Pre-allocate a list

- If you know the size of the list beforehand you can create the list with the desired length
- Create it with: [anyvalue]*desiredLenth
- [0]*n or [1]*n

```
>>> n=5 # the desired/maximum length of the list
>>> a=[0]*n
>>> print a
[0, 0, 0, 0, 0]
>>> print len(a)
5
>>> b=[1]*n
>>> print b
[1, 1, 1, 1, 1]
```

Other functions that will operate on a list

- del list will remove the whole list
  - Remove an item from a list

```
>>> a=[1,2,3,4,5]
>>> del a[3]
>>> print a
[1, 2, 3, 5]
>>> del a[0]
>>> print a
[2, 3, 5]
```

  - Will also delete a slice from a list

```
>>> a=[1,2,3,4,5]
>>> del a[1:3]
>>> print a
[1, 4, 5]
```

Other functions that will operate on a list

- reverse()
  - Reverses the list in place

```
>>> a=[1,2,3,4,5]
>>> a.reverse()
>>> print a
[5, 4, 3, 2, 1]
```

- sort()
  - Sorts the list in place

```
>>> a=[1,2,3,4,5]
>>> a.reverse()
>>> print a
[5, 4, 3, 2, 1]
>>> a.sort()
>>> print a
[1, 2, 3, 4, 5]
```

Tuples

- Tuples are lists that are immutable/cannot be changed.
- Create a tuple by enclosing the list in parentheses ()
- In some cases do not need the parentheses.

```
>>> t=(1,2,3,4,5,6)
>>> print t
(1, 2, 3, 4, 5, 6)
>>> p=1,2,3,4,5,6
>>> print p
(1, 2, 3, 4, 5, 6)
```

Tuples

- Single item needs trailing comma to indicate that it is a tuple:

```
>>> one=(1,)
>>> print one
(1,)
>>> one=1,
>>> print one
(1,)
```

Tuples

- Use indexing and slicing just like lists

```
>>> print p[0]
1
>>> print p[2:]
(3, 4, 5, 6)
```

Tuples

- Cannot change or extend a tuple

```
>>> p=1,2,3,4,5,6
>>> p[2]='this will not work'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment

>>> p.append("appending to a tuple will not
  work")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute
  'append'
```

Python Dictionaries

- Lists and tuples are an ordered collection of objects.
  - Indexed by an integer
- Dictionaries are an unordered collection of objects.
  - Indexed by a key, which can be anything.
- Very powerful. Greatly simplifies certain tasks.
- Sometimes called a hash table.
- A foundational data structure for python.
- Declare a dictionary with curly braces.
- An empty dictionary is created by an empty set of braces {}.
- Specify a particular item with dictionary[key]

Python Dictionaries

- Can add an item by assignment:

```
>>> d={}  # an empty dictionary
>>> d["first item"]="ONE" # add an item
>>> print d
{'first item': 'ONE'}
```

- Value to left of colon is the key, value to the right of colon is the item.
  - the key is **'first item'** and 'ONE' is the value assigned to the key

Python Dictionaries

- Can create a non-empty dictionary by specifying key:value pairs:

```
>>> e={'first':1,'second':2, 3:3}
>>> print e
{'second': 2, 3: 3, 'first': 1}
```

- Notice that the order is not guaranteed.

```
>>> print e['first']
1
>>> print e[3]
3
```

- len() function returns the number of items in the dictionary

```
>>> print len(e)
3
```

Python Dictionaries

- The keys() method returns a list of all the keys in the dictionary:

```
>>> e={'first':1,'second':2, 3:3}
>>> print e.keys()
['second', 3, 'first']
```

Python Dictionaries

- Can also use dictionaries as multidimensional arrays

```
>>> a={}
>>> a[1,2]=1
>>> a[2,2]=2
>>> a[1,1]=3
>>> print a
{(1, 1): 3, (1, 2): 1, (2, 2): 2}
```

  - Notice that the key is a tuple

```
>>> i=2
>>> print a[1,i]
1
>>> print a[i,i]
2
>>> print a[1,1]*a[i,i]
6
```

Conditionals

- A conditional is an expression that indicates True or False.
- Any non-zero value is True.
- A zero value or empty list is False.
- Python also has a special object/value called None that is also interpreted as False
- Conditional operators:
  - a == b is true if a is equal to b
  - a != b is true if a is not equal to b
  - a > b is true if a is greater than b
  - a < b is true if a is less than b
  - a >= b is true if a is greater than or equal to b
  - a <= b is true if a is less than or equal to b

Conditionals

```
>>> print 1 == 1
1
>>> print 1 == 2
0
>>> print 1 != 2
1
>>> print 1 > 2
0
>>> print 1 < 2
1
```

Boolean Expressions

- Boolean expressions are used to combine conditional expressions to produce more complex conditionals.
  - **a or b** is true if **a** or **b** is true.
    - If **a** is true then **b** is not evaluated.
  - **a and b** is true if **a** and **b** is true.
    - If **a** is not true then **b** is not evaluated.
  - **not a** is true if **a** is false (not true).
  - where **a** and **b** can be any conditional expression.
- Can use parentheses for grouping.
  - **(a or b) and (d and e)**

Boolean Expressions

```
>>> (1 == 1) and (2 == 2)
1
>>> (1 == 1) and (2 == 1)
0
>>> (1 == 1) or (2 == 1)
1
>>> not (1 == 1)
0
>>> not (2 == 1)
1
```

Boolean Expressions

- **Can use a variable as the conditional**

```
>>> a=100
>>> a and (1 == 1)
1
>>> a or (1 == 2)
1
>>> a and (1 == 2)
0
```

Blocks of code

- Python uses indentation to determine if a set of statements are in a block of code.
- Statements that are indented to the same level are associated with the same block of code.
- This improves the readability of the code.

Flow control

- A useful program will require decisions or branching to occur.
- Examples:
  - If the temperature is greater than 79 degrees turn the air conditioner on.
  - While the speed is less than 55 increase the throttle setting.
  - For each student in the class, print their last test score

if statement

- The if statement determines if a block of associated code will be executed.
- The end of the if statement is indicated by the colon character.
- The block is executed if the conditional is evaluated to be True.
- The block is terminated by the first statement at the previous indentation level.
- Syntax:

if conditional:

statements

```
>>> if 1 == 1:
...     print "this is true"
...
this is true
```

if statement

- The general if statement is

if <test1>:

<statements1>

elif <test2>:

<statements2>

else:

<statements3>

- elif is a contraction for else if
- The elif conditional is tested if the test1 is false and the block of statements following the elif is executed if the conditional evaluates to true.
- Likewise the statements following the else statement are executed if none of the prior conditionals were true.

if statement

```
>>> if 1 == 2:
...     print "is 1 == 2"
... elif 3 < 4:
...     print "but we know 3 is less than 4"
...
but we know 3 is less than 4
>>> a=0
>>> b=None
>>> if a:
...     print "first test was true"
... elif b:
...     print "second test was true"
... else:
...     print "none of the tests were true"
...
none of the tests were true
```

Colon Character indicates a new block

- As we have seen in the previous statements the colon character ":" is used to terminate certain statements
- These statements all precede a new block of code
- The new block of code will be indented relative to the originating statement
- Other statements to be discussed will also be consistent with this syntax
- So if you know that you are going to start a new block of code then the preceding statement must end with a colon

Looping with while

- The first looping statement is the while loop.

while <test>:

    <statements>

else:

    <statements2>

- The condition is first evaluated and if it is true the block of statements is executed. Then the code loops back to the while statement and begins again.
- The statements following the else are executed if the loop terminates normally.

Looping with while

```
>>> a=0
>>> while a < 10:
...     print a
...     a=a+1
...
0
1
2
3
4
5
6
7
8
9
```

Looping with while

```
>>> a=0
>>> while a < 10:
...     print a
...     a=a+1
... else:
...     print "the final value of a is",a
...
0
1
2
3
4
5
6
7
8
9
the final value of a is 10
```

Break out of while loop

- The break statements cause the looping to terminate and the optional else statements are skipped.
- **while 1:** is an infinite loop. Must use a break to exit the loop

```
>>> a=0
>>> while 1:
...     if a >= 6:
...         break
...     print a
...     a=a+1
... else:
...     print "the loop terminated normally"
...
1
2
3
4
5
```

Continue within while loop

- The continue statement transfers the execution back to the top of the loop.

```
>>> a=0
>>> while 1:
...     a=a+1
...     if a < 4:
...             continue
...     print a
...     if a > 10:
...             break
...
4
5
6
7
8
9
10
11
```

for loop

- The Python for loop iterates over the items in any sequence in the order they appear in the sequence
- Think of it as: for each item in the list

for <target> in <object>:

<statements>

else:

<statements executed if terminate normally>

- The **<target>** is a variable that is assigned a value from the sequence defined by **<object>.** The **<object>** is normally a list or tuple.

for loop

```
>>> for x in [1,2,3,4,5]:
...     print x
... else:
...     print "for loop terminated normally"
...
1
2
3
4
5
for loop terminated normally
```

for loop

```
>>> a={"A":1,"B":2,"C":"last value"}
>>> for key in a.keys():
...     print "The value for key",key,"is",a[key]
...
The value for key A is 1
The value for key C is last value
The value for key B is 2

>>> for key in a.keys():
...     if key == "D":
...             break
... else:
...     print "Did not find the desired key!"
...
Did not find the desired key!
```

Using Range() function with for loop

- Can use the range() function to create a list of numbers.

```
>>> b=['a','b','c','d']
>>> print len(b)
4
>>> print range(len(b))
[0, 1, 2, 3]
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(2,5)
[2, 3, 4]
```

Using Range() function with for loop

```
>>> b=['a','b','c','d']
>>> for i in range(len(b)):
...     print i,b[i]
...
0 a
1 b
2 c
3 d
```

Break and Continue with for loop

- In the same way as they did for the while loop
  - The break statement terminates the for loop
  - The continue statement transfers the execution back to the top of the for loop.

```
>>> for a in range(20):
...     if a < 6:
...             continue
...     print a
...     if a > 9:
...             break
...
6
7
8
9
10
```

Creating a script file

- Use a text editor to create a new file.
- On Linux/Unix the first line can specify the command/shell that will be used to run the script/program.
- You begin the first line of the file with #! and follow that with the path of the program to be used to run the script.
- For example:

```
#!/usr/bin/python
```

Creating a script file

- The python program may be stored in different locations on different systems.
- The following will find the python executable in your path and run it.

```
#!/usr/bin/env python
```

Creating a script file

- Add the lines to your file

```
#!/usr/bin/env python
print ‘Hello from python!’
```

- Save the file as hello.py
- Go to a command line prompt
- Run the script with:

```
python hello.py
```

Make the script file executable

- Now make the script executable with:

```
chmod 755 ./hello.py
```

- Now you can run the script without specifying the python executable:

```
./hello.py
```

More on print

- A print statement with a trailing comma will suppress the end of line characters, leaving the next print position on the same line:

```
print ‘hello world’
print ‘hello’,
print ‘world’
```

More on print

```
#!/usr/bin/env python

print "Hello from python!"

print "Hello"
print "from"
print "python!"

print "Hello",
print "from",
print "python!"

for i in range(5):
    print i,
print "end of line"
```

Functions

- A function is a block of code that is executed by calling the function name with optional arguments.
- Defines a unit of work that helps to modularize the code and promotes code reuse.
- The keyword **def** introduces a function definition.
  - def name(parameters):
- It is followed by the function name and the parenthesized list of parameters to the function.
- The colon character terminates the function declaration line.
- The block of statements that form the body of the function start at the next line.
- Remember that indentation within Python defines a block of code.

Function Arguments

- The parameters or arguments passed to a function call are added to the symbol table/name space local to the called function
  - Arguments are passed using call by value
  - An object reference is passed when the parameter is an object

```
>>> def a(b):
...  b=1
...  print b
...
>>> d=2
>>> a(d)
1
>>> print d
2
```

Pass and the Simplest Function

- A function begins with the function definition and is followed by a block of code.
- **pass** is a python statement that does nothing
- The simplest function is:

```
def noop():
    pass
```

- The **pass** statement acts as a placeholder to create a block of code where needed but does not do anything
- This is good for creating a function template that will be expanded into something useful at another time

Functions and Docstrings

- The first statement of the function body can optionally be a string literal, which is the function's documentation string, or docstring.
- There are tools within Python to extract the docstrings so it is good practice to document the code.

```
def myFunction(param1,param1):
    '''This is the docstring for myFunction'''
```

Functions and Docstrings

- The return statement within a function terminates the execution of the function and returns a value from a function to the calling routine.
- Examples:

```
return "hello world"
return 1
```

- return without an expression argument returns None.
- Falling off the end of a function also returns None.

Calling a Function

- Call a function by specifying the name and enclose in parentheses any parameters to be passed to the function
- Need the parentheses even if there are no arguments to be passed
- dir() is a built-in function in python
- If you do not include the () you will get the string representation for the dir function:

```
>>> print dir
<built-in function dir>
```

- Include the parentheses and you get the function output:

```
>>> print dir()
['__builtins__', '__doc__', '__name__', 'a', 'b']
```

Create a Function

- We'll now create a function called Adder that:
  - Accepts a list(or tuple) as a single argument
  - Sums each of the items in the list,
  - Returns the final sum.
- Save the file with the name Adder.py
- Try calling the function with a list of numbers:

```
sum1=Adder([1,2,3,4,5])
print sum1
```

- Then try it with a list of characters:

```
sum2=Adder(['a','b','c','d','e'])
print sum2
```

- Then with a string:

```
sum3=Adder("abcdefg")
print sum3
```

Function Adder

```
def Adder(list):
    '''This function will add up the elements in
  the list and return the final value'''
```

Function Adder

```
def Adder(list):
    '''This function will add up the elements in
   the list and return the final value'''
     # init the sum to the first value
```

Function Adder

```
def Adder(list):
    '''This function will add up the elements in
  the list and return the final value'''
    # init the sum to the first value
    sum=list[0]
```

Function Adder

```
def Adder(list):
    '''This function will add up the elements in
  the list and return the final value'''
    # init the sum to the first value
    sum=list[0]
    # for each element in the list,
    # skip the first one since we
    # initialized the sum with it
```

Function Adder

```
def Adder(list):
    '''This function will add up the elements in
  the list and return the final value'''
    # init the sum to the first value
    sum=list[0]
    # for each element in the list,
    # skip the first one since we
    # initialized the sum with it
    for item in list[1:]:
```

Function Adder

```
def Adder(list):
    '''This function will add up the elements in
  the list and return the final value'''
    # init the sum to the first value
    sum=list[0]
    # for each element in the list,
    # skip the first one since we
    # initialized the sum with it
    for item in list[1:]:
       # add the element to the sum
```

Function Adder

```
def Adder(list):
    '''This function will add up the elements in
  the list and return the final value'''
    # init the sum to the first value
    sum=list[0]
    # for each element in the list,
    # skip the first one since we
    # initialized the sum with it
    for item in list[1:]:
       # add the element to the sum
       sum = sum + item
```

Function Adder

```
def Adder(list):
    '''This function will add up the elements in
  the list and return the final value'''
    # init the sum to the first value
    sum=list[0]
    # for each element in the list,
    # skip the first one since we
    # initialized the sum with it
    for item in list[1:]:
       # add the element to the sum
       sum = sum + item
    # return the sum
```

Function Adder

```
def Adder(list):
    '''This function will add up the elements in
  the list and return the final value'''
    # init the sum to the first value
    sum=list[0]
    # for each element in the list,
    # skip the first one since we
    # initialized the sum with it
    for item in list[1:]:
       # add the element to the sum
       sum = sum + item
    # return the sum
    return sum
```

Function Adder

```
def Adder(list):
    '''This function will add up the elements in the
   list and return the final value'''
    # init the sum to the first value
    sum=list[0]
    # for each element in the list,
    # skip the first one since we
    # initialized the sum with it
    for item in list[1:]:
       # add the element to the sum
       sum = sum + item
    # return the sum
    return sum
sum1=Adder([1,2,3,4,5])
print sum1
print Adder(['a','b','c','d','e'])
print Adder("abcdefg")
```

Default Function Arguments

- Function arguments can have specified default values
  - def myFun(arg1,arg2=1,arg3="default",arg4=None):
- When calling the function you do not need to specify the argument if the default is correct.
- All the following will have the same effect
  - myFun(0)
  - myFun(0,1)
  - myFun(0,1,"default")
  - myFun(0,1,"default",None)

Using Keyword Arguments

- You can also pass arguments to a function call by specifying: keyword=value
- All non-keyword arguments must appear before any keyword arguments

```
def myFun(arg1,arg2=1,arg3='default',arg4=None):
```

- The following are valid

```
myFun(0,arg3='notdefault',arg2=6)
myFun(0,arg4=100)
```

- The following are invalid

```
# non-keyword arg follows a keyword argument
myFun(arg3='newdefault',2)
#did not specify the non-keyword arg
myFun(arg4=[1,2,3])
```

Using Keyword Arguments

```
>>> def myFun(arg1,arg2=1,arg3='default',arg4=None):
...         print "args are",arg1,arg2,arg3,arg4
...
>>> myFun(2)
args are 2 1 default None
>>> myFun(arg3=1,2)
SyntaxError: non-keyword arg after keyword arg
>>> myFun(3,arg3=1,arg2="hello")
args are 3 hello 1 None
```

More on Keyword Arguments

- You can use the keyword arguments to get a dynamically variable number of acceptable arguments
- If the last argument in the function definition begins with ** then any remaining keyword arguments are placed in a dictionary

```
>>> def myFun(arg1,arg2=None,**kwargs):
...     print arg1,arg2
...     for key in kwargs.keys():       print
  key,kwargs[key]
...
>>> myFun(1,arg2="a",arg3="b",arg4="c")
1 a
arg3 b
arg4 c
```

Programming Models

- Monolithic:
  - Old style programming
  - Everything in one file and one routine
  - Good for small, simple programs
- Procedural:
  - Most common style of programming
  - May have lots of globally accessible data
  - Data and Procedures are separated
  - Procedures/Functions/Subroutines that act on the data
  - Good for libraries or when do not have lots of things with distinct behavior
  - Any procedure can change the data

Programming Models

- Object Oriented:
  - Newest style of programming
  - Package of data and behavior/methods that act on the data
  - Data hiding: object methods are only way to interact with the data
  - Little if any globally accessible data
  - Ideal when you have lots of things with distinct behavior

Python Modules

- We have created created a script file that defines our Adder function and then uses the function
- We would like to make this function available to other scripts
- The Python way is to create a module: a file that contains variable and/or function definitions
- Python includes a standard library with MANY modules that extend the capability and simplify tasks
- The *main* module is the collection of variables that you have access to in a script executed at the top level and when running the interpreter interactively

Python Modules

- The file name of a module is the module name with ".py" appended.
  - myModule is contained in the file myModule.py
- Within a module, the variable __name__ contains the module's name as a string
- You make the module available to python using the import statement:
  - import myModule
- You can use variables and functions in a module by prefixing them with the module name:
  - Sum=myModule.functionName()
- You can avoid having to prefix by the module name by using
  - from myModule import *
  - Sum=functionName()

Python Modules

- Modules can import other modules
- A module can contain executable statements as well as function definitions.
- These statements are intended to initialize the module.
- They are executed only the first time the module is imported somewhere

dir() function

- The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> a=1
>>> dir()
['__builtins__', '__doc__', '__name__', 'a']
>>> def myfun():
...     print "hello"
...
>>> dir()
['__builtins__', '__doc__', '__name__', 'a',
  'myfun']
>>> print __name__
__main__
```

Python Modules

- Take your Adder.py script and add the following at the end:

```
print "dir() returns",dir()
print "__name__ = ",__name__
```

Python Modules

- Run as a script:

```
./Adder.py
15
abcde
abcdefg
dir() returns ['Adder', '__builtins__',
   '__doc__', '__name__', 'sum1', 'sum2', 'sum3']
__name__ =  __main__
```

Python Modules

- Import as a module from interactive python:

```
>>> import Adder
15
abcde
abcdefg
dir() returns ['Adder', '__builtins__',
   '__doc__', '__file__', '__name__', 'sum1',
   'sum2', 'sum3']
__name__ =  Adder
```

Python Modules

- Notice that our test statements were executed even when the module was imported.
- Notice that the __name__ variable changed from ‘main’ to ‘Adder’
- We can test to see if the file is run as a script by checking to see if the variable __name == ‘main’
- Use the condition to have test code skipped when used as a module

```
if __name__ == "__main__":
    #test code that is not executed when imported
```

Python Modules

- Copy Adder.py to AdderMod.py and add the test:

```
if __name__ == "__main__":
    sum1=Adder([1,2,3,4,5])
    print sum1
    sum2=Adder(['a','b','c','d','e'])
    print sum2
    sum3=Adder("abcdefg")
    print sum3
```

- Run as a script, then interactive python and import

```
>>> import AdderMod
>>> print AdderMod.Adder([1,2,3,4,5])
15
```

- Notice that now the test code is not executed on import

More on dir() function

- dir(objName) will return the names associated the variable/object called objName

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'EOFError',
    'Ellipsis', 'EnvironmentError', 'Exception', 'FloatingPointError',
    'IOError', 'ImportError', 'IndexError', 'KeyError',
    'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError',
    'None', 'NotImplementedError', 'OSError', 'OverflowError',
    'RuntimeError', 'StandardError', 'SyntaxError', 'SystemError',
    'SystemExit', 'TypeError', 'ValueError', 'ZeroDivisionError', '_',
    '__debug__', '__doc__', '__import__', '__name__', 'abs', 'apply', 'buffer',
    'callable', 'chr', 'cmp', 'coerce', 'compile', 'complex', 'delattr', 'dir',
    'divmod', 'eval', 'execfile', 'exit', 'filter', 'float', 'getattr', 'globals',
    'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'intern', 'isinstance', 'issubclass',
    'len', 'list', 'locals', 'long', 'map', 'max', 'min', 'oct', 'open', 'ord', 'pow',
    'quit', 'range', 'raw_input', 'reduce', 'reload', 'repr', 'round', 'setattr',
    'slice', 'str', 'tuple', 'type', 'vars', 'xrange']
```

Python Modules

- Examine some of the names after importing

```
>>> import AdderMod
>>> dir() #list of names in main
['AdderMod', '__builtins__', '__doc__',
  '__name__']
>>> print AdderMod
<module 'AdderMod' from 'AdderMod.pyc'>
>>> dir(AdderMod) #list of names in the module
['Adder', '__builtins__', '__doc__', '__file__',
  '__name__']
>>> print AdderMod.Adder.__doc__ #docstring
This function will add up the elements in the
  list and return the final value
```

Look at Namespace

- We have seen that with Python you do not declare variable names/types ahead of time
- The variable/names spring into existence when they are assigned a value
- We have seen that different portions of the code know about different names.
  - We have seen that the main, a module, and a function have different sets of names: I.e. their own namespace
- This prevents confusion and unintended side effects where the name in one place is the same in another place
  - Consider two modules or functions that used the same name for a variable
  - If these two names referenced the same variable then changing a value in one place would also change it in the other place
- This ability to have different namespaces is also called scope
  - The scope of a variable is where it is known and can be accessed

Look at Namespace

- The module (including the main or top level module) defines a global namespace for the module
- Each CALL to a function defines a new local namespace
  - Names assigned a value in a function are put in the local namespace/scope
- You can assign a value to a global variable by first declaring it to be global:

```
global varName
```

- All other names are global or built-in
  - Names that are ***not assigned a value*** in the function are assumed to be globals (in the enclosing modules namespace) or built-in (names predefined in python)

Look at Namespace

- When python looks for a variable name it follows the following three rules:
  - 1. Python searches for names in the three scopes/namespaces:
    - Local, then Global, then Built-in
    - Remember: LGB rule
  - 2. By default name assignments create or change names in the local scope
  - 3. The global declaration tells python to look for the name in the scope of the enclosing module

Example of Namespace

```
#!/usr/bin/env python
# ExamineScope.py
# the global scope for the module
g=1

def examine(a):
    # now in the local scope for the function
    # assign a value, creates the name in the local scope
    l=2
    # add the local variable l to the global variable g
    # store the result in the local variable m
    m = l + g
    # examine the names in the local scope
    print "local scope names",dir()
    return m
# call the function and put the return value in a global scope variable
r=examine(2)

print "The returned value is",r

print "global scope names",dir()
```

Run ExampleScope.py

```
./ExamineScope.py
local scope names ['a', 'l', 'm']
The returned value is 3
global scope names ['__builtins__', '__doc__',
  '__name__', 'examine', 'g', 'r']
```

Example 2 of Namespace

```
#!/usr/bin/env python
# ExamineScope2.py
# the global scope for the module
g=1
e=0
def examine(a):
    # now in the local scope for the function
    # assign a value, creates the name in the local scope
    l=2
    # add the local variable l to the global variable g
    # store the result in the local variable m
    m = l + g
    # tell python that we want to use the global name
    # and not create another local variable
    global e
    e=m
    # examine the names in the local scope
    print "local scope names",dir()
    return m
# call the function and put the return value in a global scope variable
r=examine(2)
print "The returned value is",r,"and the value of e is",e
print "global scope names",dir()
```

Run ExampleScope2.py

```
./ExamineScope2.py
local scope names ['a', 'l', 'm']
The returned value is 3 and the value of e is 3
global scope names ['__builtins__', '__doc__',
  '__name__', 'e', 'examine', 'g', 'r']
```

Name Qualification

- Remember python uses the LGB rule to find a variable with a given name
- You can specify where to find the name (name qualification) by specifying the object containing it:
  - **myMod.aVar** will search for **aVar** in the object **myMod**
    - Will use LGB rules to find **myMod**
- This expands to a qualification path:
  - **myMod.AnObject.aVar** will search for **AnObject** in the object **myMod** and then search it for **aVar**

Standard Python Modules

- Python comes with a library of standard modules
  - Described in The Python Library Reference at http://www.python.org/doc/current/lib/lib.html
- You can install your own or other modules
  - Set the environment variable PYTHONPATH to include the location where the various modules are described
  - sys.path is a variable in the sys module that contains the current path that is searched to find a module

```
>>> import sys
>>> print sys.path
['', '/opt/dislin/python', '.',
  '/opt/local/Python', '/opt/local/Python/PIL',
  '/opt/local/lib/', '/usr/local/lib/',
  '/usr/local/lib/vtk/python',
  '/home/noack/local/Python',
  '/home/noack/local/Python/lib/python/', '',
  '/usr/lib/python1.5/']
```

sys Module

- The sys module provides other important capabilities
- sys.argv
  - The list of command line arguments as strings
  - sys.argv[0] is name of the executable or script
- sys.exit()
  - Terminates python and returns to the command line
- sys.stdin
  - Standard input file, usually the keyboard
- sys.stdout
  - Standard output file, usually the terminal window/screen
- sys.stderr
  - Standard error file, usually the terminal window/screen

string Module

- The string module provides many useful functions that operate on strings
- We'll examine only a few methods
- You must first import the module

```
>>> import string
```

- **string.split(s)** splits the string **s** wherever it finds white space

```
>>> string.split("are we having fun")
['are', 'we', 'having', 'fun']
```

- **string.split(s,sep)** splits the string **s** wherever it finds the separator string **sep**

```
>>> string.split("1,2,3,4,5",",") #split at comma
['1', '2', '3', '4', '5']
```

string Module

- **string.join(wordlist)** joins the list of words inserting a single space between them

```
>>> string.join(["first","second","third"])
'first second third‘
```

- **string.join(wordlist,sep)** joins the list of words inserting the **sep** string between them

```
>>> string.join(["first","second","third"],’-’)
'first-second-third'
```

string Module

- **string.replace(s,old,new)** returns a string that replaces the occurrences of **old** by **new** in the string **s**

```
>>> string.replace("are we having fun","are
  we","We ARE")
'We ARE having fun'
```

- **string.lower(s)** returns a lowercase version of **s**
- **string.upper(s)** returns an uppercase version of **s**

```
>>> string.lower("MiXeD CaSeS")
'mixed cases‘
>>> string.upper("MiXeD CaSeS")
'MIXED CASES'
```

- See also the **re** module for even more advanced string operations

Input and Output using Files

- The print statement provides output via the standard output file.
- Access to other files is provided via a file object
- Create a file object with the python function open(filename,mode)
  - filename is a string containing the name of the file
  - mode is a string that specifies the processing mode:
    - ‘r’ means open the file for reading input
    - ‘w’ means open the file for writing output
    - ‘a’ means open the file appending, writing output to then end of the file
  - The open function returns a file object that provides appropriate methods: fileobject=open(name,mode)
  - Close the file with the fileobject.close() method

Input using Files

- Reading a file
  - First open the file and get a file object: f=open(name,'r')
  - Data = f.read() reads the WHOLE file and returns it as a string
  - Data = f.read(n) reads the n bytes from the file and returns it as a string
  - f.read() returns an empty string when it encounters and end of file:
  - Data = f.readline() read one line including the newline, '\n', at the end of the line
  - Lines=f.readlines() returns a list containing all the lines of data in the file, each line includes the newline
  - Lines=f.readlines(sizehint) returns a list containing lines of data in the file, but it will read sizehint number of bytes and enough more to get a complete line. Useful for reading large files with a small computer.

Input using Files

```
>>> f=open('Adder.py')
>>> data=f.read()
>>> f.read()
''
>>>f.close()
>>>print data

>>> f=open('Adder.py')
>>> line=f.readline()
>>> line
'#!/usr/bin/env python\012'
>>> print line
#!/usr/bin/env python

>>>
```

Input using Files

```
>>> f=open('Adder.py')
>>> lines=[] # create a list to hold the lines
>>> for i in range(3): # read only three lines
...     # append each line to the list of lines
...     lines.append(f.readline())
...
>>> print lines
['#!/usr/bin/env python\012', '\012',
 'def Adder(list):\012']
```

Output using Files

- Open the file in write or append mode
- Write to a file using the write() method
- f.write(string) writes the contents of string to the file

```
>>># open the file for writing
...f=open("MyFirstFile.txt","w")
>>># write a string to the file
...# include newline at the end of the line
...f.write("I love Python!\n")
>>>f.close()
>>>
23: cat MyFirstFile.txt
I love Python!
```

Copy a File

```
>>> #open the input file
... fin=open("Adder.py","r")
>>> #read the whole file
... contents=fin.read()
>>> fin.close()
>>> #open the output file
... fout=open("Copy","w")
>>> #write the file
>>> fout.write(contents)
>>> fout.close()
```

Another useful module: urllib

- The urllib module provides the capability to fetch data/pages from a web server
- **urlopen(url)** returns a file like object with many of the same methods
  - **u.read(), u.readline(), u.readlines(),u.close()**
  - **url** should start with a scheme identifier http: or ftp:
    - I.e. http://www.python.org
    - If **url** does not start with ftp: or http: a local file is opened

```
import urllib
# open a connection to the webserver for a specific URL
# urlopen returns a file-like object
connection=urllib.urlopen("http://www.arl.hpc.mil")
# read all the data returned by the web server
page=connection.read()# get a string with all the data
connection.close()
```

Example using urllib

- Create a script to get a web page and save the page as a local file
  - getpage.py
- Change the script to get use a url specified on the command line along with the destination file
  - copypageto.py “http://www.python.org” page.html
  - Remember sys.argv is a list of the command line arguments
  - Sys.argv[0] is the command name

getpage.py

```
#!/usr/bin/env python
#getpage.py

import urllib

# open a connection to the webserver for a specific URL
# urlopen returns a file-like object
connection=urllib.urlopen("http://www.arl.hpc.mil")
# read all the data returned by the web server
page=connection.read()
connection.close()

#open an output file
out=open("webpage.html","w")
out.write(page)
out.close()
```

copypageto.py

```
#!/usr/bin/env python
#copypageto.py

import sys
import urllib

# open a connection to the webserver for a specific URL
# urlopen returns a file-like object
connection=urllib.urlopen(sys.argv[1] )
# read all the data returned by the web server
page=connection.read()
connection.close()

#open an output file
out=open(sys.argv[2] ,"w")
out.write(page)
out.close()
```

Exception Handling

- Exceptions are a high level program flow control method
- The typical use is for error handling: to catch unusual conditions and do something appropriate
- The basic syntax is:

try:

    <statements1>

except:

    <statements2>

- The code block indicated by **<statements1>** are executed and if an exception occurs the code in **<statements2>** is executed
- The default exception handler will terminate the program
- There is additional capability not discussed here. Consult a Python book/documentation on all the details

Example Exception Handling

```
>>> a=0
>>> b=1/a
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo
```

- Now use a try to catch the exception

```
>>> a=0
>>> try:
...     b=1/a
... except:
...     print "Division by",a,"failed"
...
Division by 0 failed
```

- The try/except construct let us handle the event gracefully without terminating the program

Raising an Exception

- You can create an exception event with the **raise** statement
  - **raise <name>** where **<name>** is a string, or other appropriate object
  - For example

```
>>> raise "my first error ever"
Traceback (innermost last):
  File "<stdin>", line 1, in ?
my first error ever
>>> try:
...     raise "my first error ever"
... except:
...     print "caught it!"
...
caught it!
```

What is an Object?

- An entity or building block that has attributes and behavior
- Attributes are the data that is specific to the object
- Behavior is what an object can do
  - Called methods
  - Define the interface to the object
  - Invoking/calling the method of an object is called sending it a message

What is a Class?

- The class specifies the template or design of the object
  - What are the attributes that an object of this type should have?
  - What are the methods that an object of this type should have?
- Think of the class specification as a cookie cutter
- Used to create an object instance
  - We instantiate (create an instance of) an object to get an individual/distinct object

Object Oriented Programming Concepts

- Encapsulation
  - Package of data and behavior/methods that act on the data
- Polymorphism
  - Different objects have the same interface/method name
- Inheritance
  - Common definition of data and behavior is shared
- Composition
  - An object is composed of a collection of other objects

Encapsulation

- Package of data and behavior/methods that act on the data
- Data hiding
  - Object methods are only way to interact with the data
- Interface should be fixed
  - The implementation can be changed without affecting the other objects

Polymorphism

- Literally means many shapes
- Different objects have the same interface/method name
  - Respond to the same message
  - Act appropriate for the particular type of object
- For example:
  - Have a Circle object, Square object
    - Each object has a **DrawMe()** method that will draw the shape.
    - Same method name/interface but the method for each object does the appropriate thing.

Inheritance

- Common definition of data and behavior is shared
- IS-A relationship
- Subclass/derived class inherits data and behavior from a Superclass/base/parent Class
- Promotes code reuse
- For example:
  - Have a Shape class
  - Circle and Square inherit from Shape class
- Refactoring
  - Take an existing set of classes and find commonalities between them that define a superclass
  - Create/modify the superclass and change the code to reuse the new code in the superclass.

Composition

- An object that is composed of a collection of other objects
- HAS-A relationship
- For example
  - Car Class
    - Contains an Engine, 4 Wheels, a SteeringWheel, etc.

Python Objects

- Most everything within python is an object
  - Built-in types like integer and list are not
    - Cannot be used as a base class
- Access an objects attributes and methods with:
  - Object.attribute
  - Object.method()
- Already have seen examples of this:
  - ComplexNumber.real()
  - ComplexNumber.imag()
  - dictionary.keys()

Python Classes

- The python class is specified by the class keyword, a name for the class, optional parenthesized list of classes to inherit from, a colon, and a block of code
- The simplest form of a class definition is

class ClassName:

<block of code>

- The block of code defines a template that specifies
  - How an object of this type should look(What are its attributes)
  - How it should behave(what are its methods)
- An instance of the class is created by calling the class name without any arguments:
  - anInstance=ClassName()

Simplest Python Class

- The simplest class definition is

class passClass:

    pass

- The class does not define any default attributes or methods.
- You can use it as a structure to hold attributes and methods that you define at run time

```
>>> class passClass: pass
...
>>> a=passClass()
>>> a.day='mon'
>>> a.date='08/11/2003'
>>> def afun(arg): print 'entered afun',arg
...
>>> afun(1)
entered afun 1
>>> a.fun=afun
>>> a.fun(2)
entered afun 2
```

Python Class Attributes

- Class attributes (variables that are shared by all instances) are specified before any method definitions
- Reference them as object.attribute or className.attribute

```
>>> class myClass:
...     a=1.0
...     text="myClass is simple"
...
>>> anInstance=myClass()
>>> print anInstance
<__main__.myClass instance at 80d98c0>
>>> print anInstance.a
1.0
>>> print anInstance.text
myClass is simple
>>> print myClass.text
myClass is simple
```

Python Class Methods

- Class method definitions follow the class attributes
- Class methods again are specified by def keyword, method name, list of arguments, a colon and followed by a block of code
- Remember to properly indent
- The argument list must contain at least one variable
  - The first argument will be the object instance that by convention is called self
  - You are not required to call it self as the name self has no special meaning to Python
  - If you do not follow the convention your code may be less readable by other Python programmers
- The methods can change the values of class variables
  - self.var=value

Calling Class Methods

- As we have seen in other examples you call the method associated with an object with the syntax of: object.method()
- Remember that the first argument of a class method is the object instance
- You can call the class method in two ways:
  - Unbound class method
    - You specify the class (not an object instance) and the method
    - You must explicitly pass the instance argument
  - Bound instance methods
    - You specify the object instance and the method
    - Python packages the instance with the function in the instance class
    - You do not pass the object instance, Python adds it for you

Calling Class Methods

- Unbound class method

  anInstance=myClass() # create an instance

  – Call the method by specifying the class and passing the instance as the first argument.

  myClass.myMethod(anInstance)

- Bound class method: instance and method are packaged by python

  – Object instance is implicitly added to the call as the first argument

  anInstance=myClass() # create an instance

  anInstance.myMethod() # the first argument will be anInstance

- The unbound method call lets us call any class method as long as the object instance is appropriate

Python Class Attributes

- Variables that are unique to the instance of the object can be created or changed:
  - Within a method definition:
    - self.var=value
  - Outside the class definition by specifying the object instance:
    - anInstance=myClass()
    - anInstance.var='the value'

Python and Private Attributes

- Python object attributes are not private
  - You can access them at any place where you have an object instance

    a=myClass()

    a.var=value
  - You should exercise care in not directly accessing object attributes
  - OO approach is to access the attributes through methods

    a=myClass()

    a.setVar(value)

    val=a.getVar()
- Python has limited support of private attributes via name mangling
  - Any name of the form __name (at least two leading underscores, at most one trailing underscore) is now textually replaced with _classname__name, where classname is the current class name with leading underscore(s) stripped

Python and Private Attributes

```
#!/usr/bin/env python
class Private:
    __p=1
    def printP(self):
        print self.__p

a=Private()
a.printP() # access via a method
print a._Private__p # can still access the
  mangled name
print a.__p # cannot access the private var
  directly
```

Python Class Attributes

```
#!/usr/bin/env python2
class myClass:
    a=1.0
    text="myClass is simple"
    def myMethod(self):
        self.b=2.0
        print "You called myMethod and the
  instance is",self
anInstance=myClass()
print anInstance
print anInstance.a
print anInstance.text
anInstance.myMethod()
anInstance.c='a new instance variable'
print anInstance.b,anInstance.c
```

Object Constructor

- It is often useful to have the object initialized with specific attributes when it it created
  - In many OO languages a routine called a constructor is called when the object is created
- Python invokes a default constructor routine (that does not initialize any instance variables) when the object is create
- You can override this default behavior by defining a class method called __init__(self)

def __init__(self):

<block of code>

- You can also pass arguments to be used in setting instance variables

Object Constructor

```
#!/usr/bin/env python2
# myClassConstructor.py
class myClass:
    def __init__(self,color,fill=None):
        self.color=color
        self.fill=fill

instance1=myClass('red')
instance2=myClass('blue','solid')
print instance1.color,instance1.fill
print instance2.color,instance2.fill
```

Class Inheritance

- Inheritance provides for code reuse
  - Attributes and methods that are common between two different classes can be placed into a base class and shared
- Another view is to start with a base class(superclass) and create a derived class(subclass) that is a modification or extension of the base class
  - The base class can provide a template for derived classes
- In python the base classes are specified as a parenthesized list of base classes to the class name:

class derivedClass(baseClass):

<block of code>

- For multiple inheritance you specify the set of base classes

class derivedClass(baseClass1, baseClass2, baseClass3):

<block of code>

Inheritance and Attribute Resolution

- Python will try to resolve attribute and method by
  - Looking first in the derived class
  - Looking in the base class
    - If the base class is derived it will recursively search any of its base classes
  - For multiple inheritance cases it uses a depth first, left-to-right search path
    - Fully search the first base class
    - If not found then fully search the second base class
    - Etc.
    - Example: **class derivedClass(baseClass1, baseClass2, baseClass3):**
      - **Search baseClass1 (including any classes it inherits from)**
      - **If not found the search baseClass2, etc.**

Derived Classes Extend Base Classes

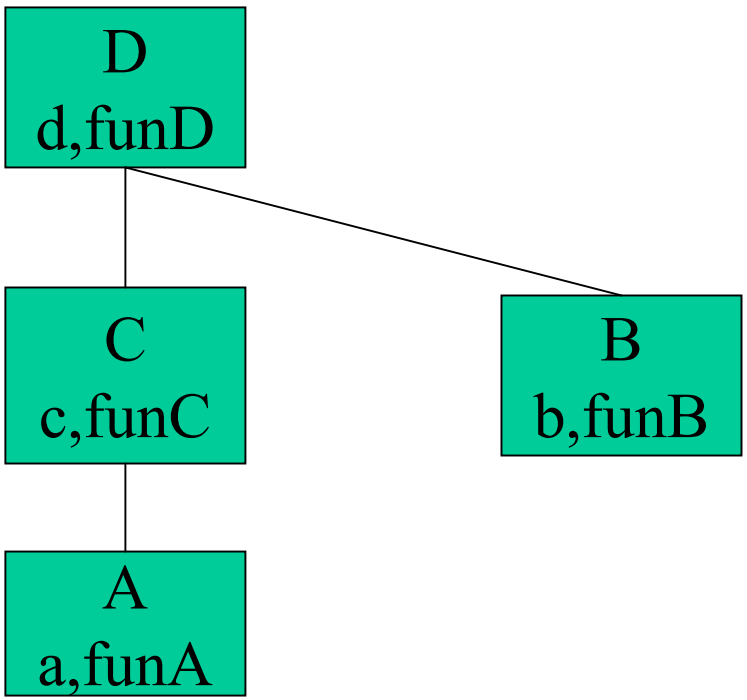
- First consider the case where a derived class will extend the base class
- The derived class adds new attributes and methods
- The base class attributes and methods are unchanged and available

Inheritance Example

```
#!/usr/bin/env python2
class a:
    a=1.0
    def funA(self):  print "You called funA",self
class b:
    b=2.0
    def funB(self):  print "You called funB",self
class c(a):
    c=3.0
    def funC(self):  print "You called funC",self
class d(b,c):
    d=4.0
    def funD(self):  print "You called funD",self

Dinstance=d()
print Dinstance.a,Dinstance.b,Dinstance.c,Dinstance.d
Dinstance.funA()
Dinstance.funB()
Dinstance.funC()
Dinstance.funD()
```

Inheritance Tree

- Let's examine the inheritance tree for the example



Override Inherited Methods

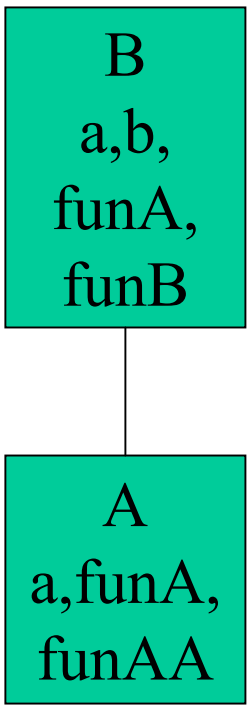
- A derived class can have attributes and/or methods that have the same name as those in a base class
- This allows for the derived class to override the behavior or attributes of the base class
  - The derived class is very similar to the base class but needs different behavior
  - The derived class has a method with the same name as the base class but does something different
- The derived class can still access the base class method by using an unbound method call

Inheritance and Override Example

```
#!/usr/bin/env python2
class a:
    a=1.0
    def funA(self):   print "You called a.funA",self
    def funAA(self):  print "You called a.funAA",self
class b(a):
    a=2
    b=3.0
    def funB(self):  print "You called b.funB",self
    def funA(self):  print "You called b.funA",self

Binstance=b()
print Binstance.a,Binstance.b
Binstance.funA()
Binstance.funB()

Binstance.funAA()
a.funA(Binstance) # unbound method call
```

Inheritance Tree

- Let's examine the inheritance tree for the example



Overloading Operators

- Classes can implement special methods to allow objects to respond to the usual operations: +, -, /, *, <, > , etc.
- Overloading in Python is achieved by providing specially named class methods
  - Already have overloaded the constructor __init__()
  - All overload methods have names that start and end with two underscores.

Common Methods for Numeric Objects

- The following methods can be defined to emulate numeric objects.
  - Methods corresponding to operations that are not supported by the particular kind of number implemented should be left undefined.

__add__(self, other)	x+y
__sub__(self, other)	x-y
__mul__(self, other)	x*y
__div__(self, other)	x/y
__abs__(self)	abs(x)
__neg__(self)	-x
__pos__(self)	+x

Methods for Objects of Different Types

- Special procedures are required when the two objects are not of the same type
  - Example: multiply a constant by a vector
- The following methods are called when the left object is not of the same type as the right
  - y+x, y-x, y*x, y/x and y does not implement the regular method

__radd__(self, other)	y+x
__rsub__(self, other)	y-x
__rmul__(self, other)	y*x
__rdiv__(self, other)	y/x

Overloading Operators: Example 1

- Class V partially implements a Vector class
  - Stores x,y,z as attributes
  - Methods for adding and multiplying (dot product) of two vectors

```
#!/usr/bin/env python2
class V:
    def __init__(self,x=0,y=0,z=0):
        self.x = x; self.y = y; self.z = z
    def __add__(self,other):
        new=V(self.x+other.x,self.y+other.y,self.z+other.z)
        return new
    def __mul__(self,other):
        new=V(self.x*other.x,self.y*other.y,self.z*other.z)
        return new
    def __repr__(self):
        return "(%f,%f,%f)"% (self.x,self.y,self.z)
v1=V(1.0,1.0,1.0)
v2=V(2.0,0.0,0.0)
print v1,"+",v2,"=",v1+v2
print v1,"*",v2,"=",v1*v2
print v1+2.0 # V + a float won't work
```

Overloading Operators: Example 1

```
./override_operators.py
(1.000000,1.000000,1.000000) +
   (2.000000,0.000000,0.000000) =
   (3.000000,1.000000,1.000000)
(1.000000,1.000000,1.000000) *
   (2.000000,0.000000,0.000000) =
   (2.000000,0.000000,0.000000)
Traceback (most recent call last):
  File "./override_operators.py", line 19, in ?
    print v1+2.0
  File "./override_operators.py", line 6, in __add__
    new=V(self.x+other.x,self.y+other.y,self.z+other.z)
AttributeError: 'float' object has no attribute 'x'
```

Overloading Operators: Example 2, Page 1

- Change Class V Methods to also allow adding and multiplying of a scalar and a vector

```
#!/usr/bin/env python2
class V:
    def __init__(self,x=0,y=0,z=0):
        self.x = x; self.y = y; self.z = z
    def __add__(self,other):
        new=V()
        if isinstance(other,V):
            new=V(self.x+other.x,self.y+other.y,
                  self.z+other.z)
        else:
            new=V(self.x+other  ,self.y+other  ,self.z+other)
        return new
    def __radd__(self,other):
        new=V()
        print "radd",other,"+",self
        new=V(self.x+other  ,self.y+other  ,self.z+other)
        return new
```

Overloading Operators: Example 2, Page 2

```
    def __mul__(self,other):
        new=V()
        if isinstance(other,V):
            new=V(self.x*other.x,self.y*other.y,
                  self.z*other.z)
        else:
            new=V(self.x*other  ,self.y*other  ,self.z*other)
        return new
    def __repr__(self):
        return "(%f,%f,%f)"% (self.x,self.y,self.z)
v1=V(1.0,1.0,1.0)
v2=V(2.0,0.0,0.0)

print v1,"+",v2,"=",v1+v2
print v1,"*",v2,"=",v1*v2

print 2+v1  # will use __radd__()
print v1+10
print v1*4
```

Overloading Operators: Example 2

```
./override_operators2.py
(1.000000,1.000000,1.000000) +
   (2.000000,0.000000,0.000000) =
   (3.000000,1.000000,1.000000)
(1.000000,1.000000,1.000000) *
   (2.000000,0.000000,0.000000) =
   (2.000000,0.000000,0.000000)
radd 2 + (1.000000,1.000000,1.000000)
(3.000000,3.000000,3.000000)
(11.000000,11.000000,11.000000)
(4.000000,4.000000,4.000000)
```

Container Objects

- We can create a container object by implementing special methods
  - A container is either a list (integer key) or a dictionary (arbitrary key)
  - Allows us to access items in our object with **obj[key]** syntax rather than **obj.list[key]**
  - Python calls the following methods for the respective access
    - **__getitem__(self,key)** for **self[key]**
    - **__setitem __(self,key,value)** for **self[key]= value**
    - **__len __(self)** for **len(self)**
    - A complete implementation will require more methods

Container Objects: Example

```
#!/usr/bin/env python2
class ListContainer:
    def __init__(self,n=0):
        if n > 0:
            self.list=[0]*n
        else:
            self.list=[]
    def __getitem__(self,key):
        return self.list[key]
    def __setitem__(self,key,value):
        self.list[key]=value
    def append(self,value):
        self.list.append(value)
c=ListContainer()
c.append(1); c.append(2)
print "c[0]=",c[0]
for item in c:
    print item
c[1]=10
print "c[1]=",c[1]
```

Container Objects: Example

```
./container.py
c[0]= 1
1
2
c[1]= 10
```

Creating a GUI

- A graphical user interface (GUI) can simplify the interaction with a user for many tasks
  - There are many cases where it can make it more difficult to perform a task
- People are very comfortable with the graphical interface
- There are many toolkits available that can be used to create a GUI
- Using a toolkit that runs on multiple platforms preserves your investment and does not lock you into a proprietary platform

GUI Toolkits

- A partial list of toolkits that have support for Python
  - Tkinter, included with Python, based upon TCL/TK
    - http://www.tcl.tk/
  - WXWINDOWS, a C++ library, wxPython
    - http:// www.wxwindows.org
    - http:// www.wxpython.org
  - Fox , a C++ library, FXPy
    - http://www.fox-toolkit.org
    - http://fxpy.sourceforge.net
  - Qt , a C++ library, PyQt (not free for Windows)
- We will look at Tkinter because:
  - It is included with Python (eliminates having to install another package)
  - It is fairly simple to use

General Comments on GUI Programming

- A GUI is intended to provide a lot of flexibility in its interaction with the user
  - Can push this button, or that one, move a slider, or a scrollbar, etc.
- The interaction can come from many different GUI components and in any sequence
- This requires an asynchronous/event driven approach
- The programmer specifies:
  - What components are used, where they are placed, etc.
  - What should happen when an event occurs, ie. When button A is pressed call function ButtonPressed()
- The toolkit is responsible for drawing the GUI and handling the even loop.
  - The control of the execution is passed to the toolkit

First GUI

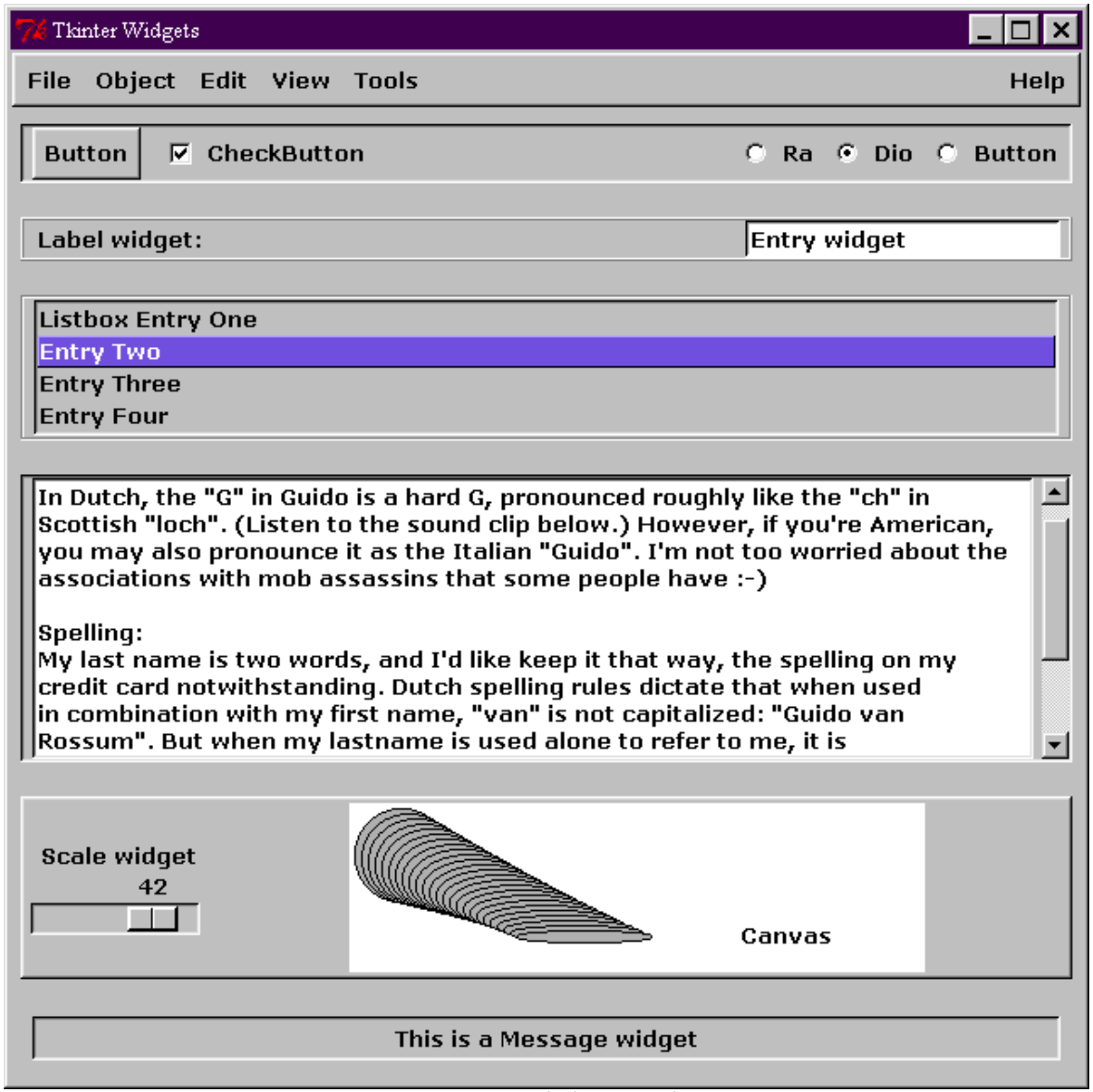
- Import the Tkinter module
- Create a root window to hold everything
- Pass control to the toolkit

```
>>> import Tkinter
>>> root=Tkinter.Tk()
>>> root.mainloop()
```

Tkinter Widgets

- Tkinter provides a number of widgets to create a GUI
  - Button
  - RadioButton
  - CheckButton
  - MenuButton
  - Text
  - Entry
  - Label
  - ListBox
  - Menu
  - ScrollBar
  - Message
  - MenuButton
  - Scale

Tkinter Widgets



Tkinter Organizational Widgets

- Tkinter provides a number of widgets to control the GUI layout
  - Root or TopLevel
    - Creates a new window
  - Frame
    - A container for other widgets
  - Layout managers
    - Grid
      - Specify the location as row,column
    - Pack
      - Specify the side of the available space where the widget should be placed
      - Like packing a suitcase
    - Place
      - Specify the location in window coordinates or as a fraction of the space available

Tkinter Widgets: Parent/Child Relationship

- A tree structure is used to organize the widget relationships
- Each widget will be created as a child of a specified parent widget
- The root widget does not have a parent
  - This is the top of the tree
  - Created with:
    - root=Tk()
- The Toplevel widget creates a new window that is a child of the root window

```
>>> from Tkinter import *
>>>
>>> root=Tk()
>>> root.title('Root Window')
''
>>> newtop=Toplevel(root)
>>> newtop.title('New TopLevel Window')
''
>>> root.mainloop()
```

Tkinter Widgets: Label

- Label class creates a widget that will display text or images
  - Text can be multiple lines
  - Must specify the parent widget
  - Wlabel=Label(root,text='This is the label string')

```
#!/usr/bin/env python2
from Tkinter import *

root=Tk()
wlabel=Label(root,text="Hello there")
wlabel.grid(row=0)

root.mainloop()
```

Tkinter Widgets: Entry

- Entry class creates a one-line text box where the text string can be edited:
  - Must specify the parent widget
  - Wentry=Entry(root)
- Wentry.get() method returns the text as a string
- Wentry.insert(index,string) inserts the contents of string before the character specified by index

```
#!/usr/bin/env python2
from Tkinter import *
root=Tk()
wentry=Entry(root)
wentry.grid(row=0)
wentry.insert(0,"Hello there")
contents=wentry.get()
print "The entry widget contained:", contents

root.mainloop()
```

Tkinter Widgets: Button

- Button class creates a button widget that displays a text string, a bitmap, or image
  - Must specify the parent widget
  - Wbutton=Button(root)
- It can be displayed in three different ways:
  - Raised, sunken, or flat
- It will call a callback routine when invoked (press left mouse button with cursor over the button

```
#!/usr/bin/env python2
from Tkinter import *

root=Tk()
wbutton=Button(root,text="Push Me")
wbutton.grid(row=0)

root.mainloop()
```

Tkinter Widgets: Button Callback

- A function can be specified to be invoked or called when the button is pressed
  - Wbutton=Button(root,text='Push me',command=functionName)

```
#!/usr/bin/env python2
from Tkinter import *

def buttonCallback():
    print 'You pushed me!'

root=Tk()
wbutton=Button(root,text="Push Me",
  command=buttonCallback)
wbutton.grid(row=0)

root.mainloop
```

Tkinter Widgets: Button Callback

- Notice that the callback function did not have any arguments
- You can access/change global variables

```
#!/usr/bin/env python2
from Tkinter import *
count=0
def buttonCallback():
    global count
    count=count+1
    print 'You pushed me',count,'times!'
root=Tk()
wbutton=Button(root,text="Push Me",
  command=buttonCallback)
wbutton.grid(row=0)
root.mainloop()
```

Tkinter Widgets: Button Callback

- Notice that the callback function did not have any arguments
- You can access/change global variables

```
#!/usr/bin/env python2
from Tkinter import *
count=0
def buttonCallback():
    global count
    count=count+1
    print 'You pushed me',count,'times!'
root=Tk()
wbutton=Button(root,text="Push Me",
  command=buttonCallback)
wbutton.grid(row=0)
root.mainloop()
```

Button Callback: OO approach

- Remember that Python packages an instance and its method when the method is called
- This allows us access to an instance variable
- Callback will have an argument that is the instance variable: self

```
#!/usr/bin/env python2
from Tkinter import *
class MyApp:
    def __init__(self,root):
        self.wbutton=Button(root,text="Push Me",
   command=self.buttonCallback)
        self.wbutton.grid(row=0)
        self.count=0
    def buttonCallback(self):
        self.count=self.count+1
        print "You pushed me",self.count,"times!",self
root=Tk()
app=MyApp(root)
root.mainloop()
```

Button Callback Using Inheritance

- Now lets create a class that is derived from the Button class
- We'll add more than one button
  - Each button will be a separate instance and hence have private attributes

Button Callback Using Inheritance

```
#!/usr/bin/env python2
from Tkinter import *
class MyButton(Button):
    def __init__(self,root,text="change me", maxtimes=10):
        # we have to explicitly call the contructor for the
   base class
        self.wbutton=Button.__init__(self,root,text=text,
   command=self.buttonCallback)
        self.text=text; self.count=0
        self.maxtimes=maxtimes
    def buttonCallback(self):
        self.count=self.count+1
        if self.count > self.maxtimes: print "Ouch!"
        else:
            print "You pushed",
   self.text,self.count,"times!",self
root=Tk()
button1=MyButton(root,text="Button1"); button1.grid(row=0)
button2=MyButton(root,text="Button2",maxtimes=5)
button2.grid(row=1)
root.mainloop()
```

Connecting Button with Entry

- A complete GUI will have interaction between the different components
- We'll demonstrate this with a Button that retrieves the contents of an Entry widget
- Design decisions:
  - MyButton class will inherits from Button
  - Each button will have separate instance of the MyButton object
  - Each button will have an attribute that is the associated Entry widget
  - Pressing the button will get the text from the associated Entry widget

Connecting Button with Entry Button: Page1

```
#!/usr/bin/env python2
from Tkinter import *

class MyButton(Button):
    def __init__(self,root,text="change me"):
        # we have to explicitly call the contructor for
   the base class

   self.wbutton=Button.__init__(self,root,text=text,
command=self.buttonCallback)
        self.text=text
        self.wentry=None
    def buttonCallback(self):
        contents=self.wentry.get()
        print "The entry widget for button",self.text,"
   contained:", contents
```

Connecting Button with Entry Button: Page2

```
root=Tk()
button1=MyButton(root,text="Button1")
button2=MyButton(root,text="Button2")

# save the entry widget as an instance variable
button1.wentry=Entry(root)
button2.wentry=Entry(root)

# place each widget
button1.grid(row=0); button1.wentry.grid(row=0,col=1)
button2.grid(row=1); button2.wentry.grid(row=1,col=1)

root.mainloop()
```

re Module for Advanced String Handling

- String module can search based upon a fixed string
- Regular expressions are search strings that form vast number of possible search strings
  - Describe search strings composed of patterns of characters
  - Probably familiar with wild card/characters from Unix: * ?
- Regular expressions are strings that contain regular text and special character sequences.
- The special characters provide the power to generalize the search to many possible search strings
- See also
  - "Mastering Regular Expressions" by Jeffrey Fiedl, ISBN 0596002890, O'Reilly and Associates
  - http://www.python.org/doc/current/lib/module-re.html
  - http://www.amk.ca/python/howto/regex/
- **`import re`** to make the module available

re Module Functions

- Re module contains main functions for working with strings/text: searching, substitution, splitting, etc.
- The re.search() and re.match() methods return a MatchObject that have methods of their own
- The re.compile() method compiles the regular expression and returns a RegexObject with a set of methods
  - Speed up repeated use by compiling the regular expression

re Module Functions

- search(pattern,string)
  - Searches string for the first match of pattern
  - Returns MatchObject if match is found, **None** if not
- search(pattern,string,**flags**)
  - Same but **flags** is a bitwise or of flags that control the behavior
    - For example **flags=re.I** changes to non-case-sensitive matching
- match(pattern,string,**flags**) is similar but checks for zero or more occurrences at the beginning of the string

```
>>> print re.search("a","A")
None
>>> print re.search("a","a")
<re.MatchObject instance at 80f1a90>
>>> print re.search("a","A",re.I)
<re.MatchObject instance at 80e7b70>
```

Regular Expression Special Characters

- “text” will match the literal string “text”
  - re.search(“Hello”,line)
- Wild Character
  - “.” matches any character except newline
    - re.search(“He..o”,line) will match “Hello”, “HeLLo”, “HeAAo”, etc.
- Position Anchors
  - “^” matches the start of a string
    - re.search(“^Hello”,line) will match “Hello” but not “no Hello”
  - “$” matches the end of a string
    - re.search(“Hello$”,line) will match “Hello” but not “Hello there”

Regular Expression Special Characters

- Repeat Count
  - "*" matches **zero** or more repetitions of the preceding expression and matches as many repetitions as possible (Greedy match).
    - re.search("Hel*o",line) matches "Heo", "Helo", "Hellllllo"
  - "+" matches **one** or more repetitions of the preceding expression and matches as many repetitions as possible (Greedy match).
    - re.search("Hel+o",line) matches "Helo", "Hello", "Hellllllo"
  - "?" matches **zero** or **one** repetitions of the preceding expression
    - re.search("Hel?o",line) matches "Helo", "Hello", "Hellllllo"

```
>>> import re
>>> print re.search("Hel*o","Hellllo")
<re.MatchObject instance at 80edf88>
>>> print re.search("Hel?o","Hellllo")
None
>>> print re.search("Hel?o","Heo")
<re.MatchObject instance at 80e7af8>
>>> print re.search("Hel+o","Heo")
None
```

Regular Expression Special Characters

- Repeat Count: Non-Greedy
  - To make matches non-greedy (match as few repetitions as possible) add a "?" to the repeat character
  - "*?" matches **zero** or more repetitions of the preceding expression
    - re.search("Hel*?o",line) matches "Heo", "Helo", "Hellllllo"
  - "+" matches **one** or more repetitions of the preceding expression
    - re.search("Hel+o",line) matches "Helo", "Hello", "Hellllllo"
  - "?" matches **zero** or **one** repetitions of the preceding expression
    - re.search("Hel?o",line) matches "Helo", "Hello", "Hellllllo"

```
>>> import re
>>> print re.search("Hel*o","Hellllo")
<re.MatchObject instance at 80edf88>
>>> print re.search("Hel?o","Hellllo")
None
>>> print re.search("Hel?o","Heo")
<re.MatchObject instance at 80e7af8>
>>> print re.search("Hel+o","Heo")
None
```

Regular Expression Characters Sets

- A set of characters that should be matched are specified inside []
  - [abcd] will match any **one** of the characters in the set
    - [0-9] is the set of numeric digits
    - [a-zA-Z] is the set of lower and upper case non-numeric characters
  - Prefix the set by "^" to match the characters not in the set
    - [^0-9] will match anything that is not a numeric digit

```
>>> import re
>>> print re.search("[0-9]","a0b")
<re.MatchObject instance at 80e96c0>
>>> print re.search("[0-9]","ab")
None
>>> print re.search("[^0-9]","ab")
<re.MatchObject instance at 80d9290>
```

Regular Expression as Raw Strings

- The regular expression strings will frequently have embedded special characters and the backslash “\”
- A raw string hides the special meaning of the special characters from the python interpreter and passes the string to the re module for processing
- Denote a raw string by prefixing the string with ***r***
  - For example:
    - r”a raw string”
    - r”\w.”

Regular Expression Groups

- The substring matched by a regular expression can be saved, numbered/named, and used later
  - Enclose the regular expression in parentheses (), reuse the substring with *number* where *number* is from 1-99
  - ([0-9.]*)
  - Name the group with (?P<name>pattern), reuse the substring with (?P=name)

```
>>> a=re.search(r"([0-9.]*)","1234 ab 1.3")
>>> print a.groups()
('1234',)
>>> print a.group()
1234
>>> a=re.search(r"(?P<digits>[0-9.]+)","1234 ab 1235")
>>> print a.groups()
('1234',)
>>> print a.group("digits")
1234
```

Reusing Regular Expression Groups

```
>>> a=re.search(r"(?P<digits>[0-9.]+).*(?P=digits)",
   "1234 ab 1235")
```

- Have a raw string pattern with a group named “digits”
  - Pattern is “[0-9]+” to match one or more digits.
- Whole pattern will look for a set of digits, zero or more of any character, followed by the same set of digits
- The digits substring that matched is:

```
>>> a.group("digits")
'123'
```

- The whole substring that matched is:

```
>>> a.group()
'1234 ab 123‘
```

Other re Module Functions

- split(pattern,string [,maxsplit=0])
  - Splits string at the occurrences of the pattern
  - Returns the list of substrings.
  - Optional argument maxsplit limits the number of splits

```
>>> print re.split(r"[0-9]","a0b1c2d3e")
['a', 'b', 'c', 'd', 'e']
```

- findall(pattern,string)
  - Returns a list of all non-overlapping matches of the pattern
  - Include empty matches

```
>>> re.findall(r"([0-9.]*)","1234 ab 1.3")
['1234', '', '', '', '', '1.3', '']
```

Other re Module Functions

- sub(pattern,replacement,string [,maxnumber=0])
  - Replaces the leftmost non-overlapping occurrences of pattern in string
  - Returns the new string.
  - Optional argument maxnumber limits the number of substitutions
  - Replacement can be a function
    - Argument is a MatchObject and should return the replacement string

```
>>> re.sub(r"([0-9.]+)","<DELETED>","1234 ab 1.3")
'<DELETED> ab <DELETED>‘
```

- subn(pattern,replacement,string [,maxnumber=0])
  - Identical to sub() but returns a tuple containing the new string and the number of occurrences

```
>>> re.subn(r"([0-9.]+)","<DELETED>","1234 ab 1.3")
('<DELETED> ab <DELETED>', 2)
```

Using Compiled Regular Expressions

- Compiling a regular expression can speed up repeated use
- reo=re.compile(pattern [,flags])
  - A RegexObject is returned
  - RegexObject has many methods and attributes available
  - Do not pass the pattern to the method calls
- reo.split(string [,maxsplit])
  - Do not specify the pattern as an argument
  - Behavior is identical to the split() function
- reo.findall(string)
  - Identical to the findall() function
- reo.sub(repl,string [,maxnumber=0])
  - Identical to the sub() function
- reo.subn(repl,string [,maxnumber=0])
  - Identical to the subn() function

Using Compiled Regular Expressions

```
>>> reo=re.compile("([0-9.]+)")
>>> print reo
<re.RegexObject instance at 80eb720>
>>> reo.subn(r"<DELETED>","1234 ab 1.3")
('<DELETED> ab <DELETED>', 2)
>>> reo.search(r"1234 ab 1.3")
<re.MatchObject instance at 80d7c38>
```

Parallel Programming with Threads

- There are two common parallel programming paradigms
  - Parallel threads of execution on shared memory machines
    - Cannot have threads across nodes in a distributed memory cluster
  - Message passing for distributed (and/or shared) memory machines
    - pyMPI is an interface to MPI for Python
      - http://sourceforge.net/projects/pympi
      - Good way to learn/experiment with MPI
      - Can build large scale parallel computations with Python as the driver/executive
- This section will focus on programming with threads

Why use Parallel Threads?

- The use of parallel execution to speed up a computation is common for HPC users
- More common need is to maintain responsiveness of one part of the program while another is busy.
  - Allow the user to interact with the GUI while some time consuming task takes place
    - Time consuming computation is run in a separate thread
  - Wait for/read input while other tasks/GUI are active
    - Separate reader thread(s) waits for/processes input
      - A thread is assigned to each input stream
      - Maintain response time for each reader
  - GUI runs in main thread, other tasks are run in separate threads

All Threads Share Access to Process Memory

- Don't have to pass messages to transfer memory
- Each thread can read/write to the same memory location
  - Can cause problems if two threads read/write to same location at the same time
    - Both threads read same value, modify it, write it back: Value will be from the last one to write
  - User must lock access to memory that may have conflicts
- All GUI execution must be done in the main thread
  - Other threads cannot make call to GUI components
  - If GUI interactions are required (for example: new data to be displayed) must inform the main thread

Threads and the Python Interpreter

- Remember Python is an interpreted language
  - Compiled to bytecodes that are interpreted
- Python schedules and switches between threads
  - Uses a global lock that allows only as single thread of execution at a time
  - Switching can only occur between execution of individual byte codes
- Care must be exercised if using an extension module written in another language
  - Long running calculation in another programming language may limit effectiveness
    - Will block execution of all other threads unless specifically written to interact with the threaded Python interpreter
- Must be careful with main thread exit
  - Other threads may be killed immediately

Python Has Two Thread Modules

- The thread module provides low-level functions for working with threads
  - Very simple interface/few methods
  - Similar interface as Unix pthreads library
- The threading module provides higher-level, object oriented support for working with threads
  - Built on top of the thread module
  - Has some other useful functions
- Choice depends upon your programming style

What is executed in a Thread?

- In both modules the thread executes a worker function that the user supplies
- A tuple of arguments and an optional dictionary of keyword arguments are passed to the worker function
  - Function and arguments are passed to call to start the thread
  - The two thread modules call your worker function with the supplied arguments

Thread Module

- import thread to make module available
- Call **thread.start_new_thread(worker,args [,kwargs])** to start a thread that will execute the function **worker**
  - **args** is a tuple of arguments to be passed to the worker function
  - **kwargs** is an optional dictionary with keyword arguments to be passed to the function
- Call **thread.get_ident()** to get an integer thread identifier of the current thread

```
def worker(arg1,arg2,arg3):
   pass
thread.start_new_thread(worker,(1,2,3))
```

Thread Example 1

```
#!/usr/bin/env python2
#threads-1.py

import thread,time

def worker(arg1,arg2=None,arg3="a"):
    id=thread.get_ident()
    print "I am thread",id
    print id,arg1,arg2,arg3
    time.sleep(1)

nthreads = 4
for t in range(nthreads):
    thread.start_new_thread(worker,(t,-1,3))
    thread.start_new_thread(worker,(t,"Using default arg"))

#give the threads time to run before exiting
time.sleep(10)
```

Thread Example 2

```
#!/usr/bin/env python2
#threads-2.py

import thread,time

def worker(arg1,arg2,arg3,**kwargs):
    id=thread.get_ident()
    print "I am thread",id
    print id,arg1,arg2,arg3
    time.sleep(1)
    for key in kwargs.keys():
        print id,arg1,key,kwargs[key]

nthreads = 4
for t in range(nthreads):
    thread.start_new_thread(worker,(t,2,3),
                            {"a":10,"b":20,"c":30})

time.sleep(10)
```

Thread Locking

- A big advantage of threaded programs is that the memory space is shared
- Attempts to update shared data can cause a race condition where the results are inconsistent
- Must provide mutually exclusive access to the shared data to make program consistent and deterministic
- Thread must acquire a mutex lock associated with the data
  - First allocate the mutex lock object
    - **Lock=thread.allocate_lock()**
    - **Lock=threading.Lock()**
  - Acquire the lock
    - **Lock.acquire()**
- Lock must be released when finished
  - Failure to release will cause a deadlock where other threads are waiting for the lock
    - **Lock.release()**

Minimize Time Holding a Lock

- Want to minimize the time spent holding a lock
  - Other threads may be waiting for lock
- Can have multiple mutex lock objects to provide finer granularity of locking
  - Allocate a mutex lock object for each section of critical code/variable

```
Lock_A=threading.Lock()
Lock_B=threading.Lock()

Lock_A.acquire()
#change variable A
#save value for my computations
Lock_A.release()
#other code
Lock_B.acquire()
#change variable B
#save value for my computations
Lock_B.release()
```

Example: Threads and Race Condition; pg 1

```
#!/usr/bin/env python2
#race-1.py
import thread,time,random
# global counter variable that each thread will update
counter=0

def worker(sleep_time,increment):
    global counter
    # save the current value
    oldvalue=counter
    # sleep to simulate doing something
    time.sleep(sleep_time)
    # now update the global variable
    counter=oldvalue + increment
    id=thread.get_ident()
    # output values for the thread
    print "I am thread",id,counter, \
  "oldvalue=",oldvalue,sleep_time
```

Example: Threads and Race Condition; pg 2

```
nthreads = 4
for t in range(nthreads):
  sleep_time=random.randrange(0,4)

  thread.start_new_thread(worker,(sleep_time,t+1))

#give the threads time to run before exiting
time.sleep(10)

print "final value is",counter
```

Example: Threads and Race Condition; pg 1a

```
#!/usr/bin/env python2
#race-1a.py
import time,random
from threading import *
# global counter variable that each thread will update
counter=0
def worker(sleep_time,increment):
    global counter
    # save the current value
    oldvalue=counter
    # sleep to simulate doing something
    time.sleep(sleep_time)
    # now update the global variable
    counter=oldvalue + increment
    id=currentThread()
    # output values for the thread
    print "I am thread",id,counter,"oldvalue=",oldvalue,
  sleep_time
```

Example: Threads and Race Condition; pg 2a

```
nthreads = 4
threads=[0]*nthreads
# create the thread objects
for t in range(nthreads):
  sleep_time=random.randrange(1,4)
  threads[t]=Thread(target=worker,args=(sleep_time,t+1))

#start them running
for t in range(nthreads):   threads[t].start()

#wait for each thread to finish
for t in range(nthreads):
   print "Joining", threads[t]; threads[t].join()

print "final value is",counter
```

Example: Threads and Race Condition; pg 2a

```
$ ./race-1a.py
Joining <Thread(Thread-1, started)>
I am thread <Thread(Thread-2, started)> 2 oldvalue= 0 1
I am thread <Thread(Thread-3, started)> 3 oldvalue= 0 1
I am thread <Thread(Thread-1, started)> 1 oldvalue= 0 2
Joining <Thread(Thread-2, stopped)>
Joining <Thread(Thread-3, stopped)>
Joining <Thread(Thread-4, started)>
I am thread <Thread(Thread-4, started)> 4 oldvalue= 0 2
final value is 4

$ ./race-1a.py
Joining <Thread(Thread-1, started)>
I am thread <Thread(Thread-3, started)> 3 oldvalue= 0 1
I am thread <Thread(Thread-4, started)> 4 oldvalue= 0 1
I am thread <Thread(Thread-1, started)> 1 oldvalue= 0 3
Joining <Thread(Thread-2, started)>
I am thread <Thread(Thread-2, started)> 2 oldvalue= 0 3
Joining <Thread(Thread-3, stopped)>
Joining <Thread(Thread-4, stopped)>
final value is 2
```

Example: Using Mutex Locks; pg 1a

```
#!/usr/bin/env python2
#race+lock-1a.py
import time,random
from threading import *
#global variable and a lock
counter=0
counter_lock=Lock()
def worker(sleep_time,increment):
    global counter,counter_lock
    #acquire the lock before we get the data
    counter_lock.acquire()
    oldvalue=counter
    time.sleep(sleep_time)
    counter=oldvalue + increment
    #release the lock now that we are done
    counter_lock.release()
    id=currentThread()
    print "I am thread",id,counter,"oldvalue=",oldvalue,
  sleep_time
```

Example: Using Mutex Locks; pg 2a

```
nthreads = 4
threads=[0]*nthreads
# create the thread objects
for t in range(nthreads):
  sleep_time=random.randrange(1,4)
  threads[t]=Thread(target=worker,
                    args=(sleep_time,t+1))

#start them running
for t in range(nthreads):   threads[t].start()

#wait for each thread to finish
for t in range(nthreads):  print "Joining", threads[t];
   threads[t].join()

print "final value is",counter
```

Example: Using Mutex Locks; pg 2a

```
$ ./race+lock-1a.py
Joining <Thread(Thread-1, started)>
I am thread <Thread(Thread-1, started)> 1 oldvalue= 0 3
Joining <Thread(Thread-2, started)>
I am thread <Thread(Thread-2, started)> 3 oldvalue= 1 2
Joining <Thread(Thread-3, started)>
I am thread <Thread(Thread-3, started)> 6 oldvalue= 3 1
Joining <Thread(Thread-4, started)>
I am thread <Thread(Thread-4, started)> 10 oldvalue= 6 2
final value is 10
```

Subclassing From Thread

- The treading module provides an object oriented interface to the thread control
- You can subclass from the Thread class to maintain the OO style
- Can only override the constructor, **__init__(),** and **run()** functions
  - Must call the base class constructor if overriding the constructor
- Advantages
  - We can create a custom constructor with the arguments that are more understandable than a tuple
  - Can subclass from other classes to have access to other methods

Subclassing From Thread: Example 2; pg 1

```
#!/usr/bin/env python2
# threads-1b.py

import time,random
from threading import *

# global variable and a lock
counter=0
counter_lock=Lock()
```

Subclassing From Thread: Example 2; pg 2

```
class myThread(Thread):
    def __init__(self,id,increment,sleep_time):
        self.id = id
        self.increment = increment
        self.sleep_time = sleep_time
        # must call base class constructor
        Thread.__init__(self)

    def run(self):
        global counter,counter_lock
        # acquire the lock before we get the data
        counter_lock.acquire()
        oldvalue=counter
        time.sleep(self.sleep_time)
        counter=oldvalue + self.increment
        # release the lock now that we are done
        counter_lock.release()
        id=currentThread()
        print "I am thread",id,counter,"oldvalue=",oldvalue,
   self.sleep_time
```

Subclassing From Thread: Example 2; pg 3

```
nthreads = 4
threads=[0]*nthreads
# create the thread objects
for t in range(nthreads):
    sleep_time=random.randrange(1,4)
    threads[t]=myThread(t,t+1,sleep_time)
#start them running
for t in range(nthreads):   threads[t].start()

#wait for each thread to finish
for t in range(nthreads):  threads[t].join()

print "final value is",counter
```

Example: GUI and Threads

- Next example will utilize threads to perform work for a Tkinter GUI.
- Main thread will perform all GUI calls
- Worker threads do not interact with the GUI
- Will use mutex locks to coordinate access to critical data
- Program design:
  - GUI will consist of an entry field and a button
  - User enters a comma separated list of stock symbols
  - A separate thread is spawned to get a quote for each one.
  - When the data is available a popup window will display the data along with a dismiss button
    - Use the Tkinter after function to check for completed quotes as a background work process
  - Use Python classes to package data with methods instead of using global variables

Mixin Class for Background Timer

- We will create a Background Timer class that uses the Tkinter after() function to execute a function in the background
  - Mixin Class: a class that is designed to add functionality/methods to another class through inheritence. Not usually instantiated by itself

```
class BGtimer:
    def StartTimer(self,msecs,WorkProc):
        self._msecs = msecs
        self.WorkProc=WorkProc
        # submit it to the event queue for the first time
        self._submit()
    def StopTimer(self):
        self.root.after_cancel(self.timer_id)
        self.WorkProc = None
        self._msecs = 0
        self.timer_id=None
    def _submit(self):
        self.timer_id=self.root.after(self._msecs,self._run)
    def _run(self):
        if self.WorkProc:
            self.WorkProc()
            # resubmit the after event so that it continues
            self._submit()
```

Use a Python Class to Package the Data

```
#!/usr/bin/env python
# getquotes+threads.py
import sys
import string
import urllib
from Tkinter import *
from threading import *
from BGtimer import *

class GetQuoteApp(BGtimer):
    def __init__(self,root):
        self.root = root
        self.threads={}
        self.quotes={}
        self.quotes_lock=Lock()
        # Create GUI Widgets
        self.wsymbol  = Entry(self.root)
        wquote_button = Button(self.root,text="Quote")
        wquote_button.bind("<Button-1>",self.GetDetailsCB)
        self.wsymbol.grid(row=0)
        wquote_button.grid(row=1)
```

Button Callback Starts Threads

```
def GetDetailsCB(self,arg):
    stocklist=self.wsymbol.get()
    list=string.split(stocklist,",")
    if not list:
        # empty entry field. Nothing to do
        return
    # get the lock because we're adding to the self.threads array
    self.quotes_lock.acquire()
    # create the thread objects and start them running
    for sym in list:
        # skip the thread if we already started it
        if self.threads.has_key(sym) and self.threads[sym].isAlive():
            continue
        thread=Thread(target=self.GetQuote,args=(sym,))
        # save it so we can see if it is alive
        self.threads[sym]=thread
        # start them running
        thread.start()
    # release the lock
    self.quotes_lock.release()

  # start our background timer to process the results from the threads
   self.StartTimer(100,self.CreateQuotePopup_from_list)
```

Thread Work Function to Get Quote

```
def GetQuote(self,sym):
    print "Getting Quote for",sym
    baseurl = "http://quote.yahoo.com/d?f=snl1d1t1c1p2va2bapomwerr1dyj1x&s=";
    url = baseurl + sym

    try:
       f=urllib.urlopen(url)
       data = f.readlines()
       f.close()
    except:
       return

    fields=string.split(data[0],",")

    self.quotes_lock.acquire()
    self.quotes[sym]=fields
    self.quotes_lock.release()
```

Thread Work Function to Get Quote

```
def CreateQuotePopup_from_list(self):
      """Create a popup with the data for each stock that is in the list"""
       # get the lock and see if we have any data
       self.quotes_lock.acquire()
       if self.quotes:
           # have data. get it and remove it from the list
           quotes_lcl={}
           for key in self.quotes.keys():
               quotes_lcl[key]=self.quotes[key]
               # remove the quote and thread from the main list
               del(self.quotes[key])
               del(self.threads[key])
           # now create a popup for each quote
           for key in quotes_lcl.keys():
               self.CreateQuotePopup(key,quotes_lcl[key])
       # release the lock
       self.quotes_lock.release()
       if self.quotes:
           # resubmit ourself if there are more quotes
           self.StartTimer(100,self.CreateQuotePopup_from_list)
       elif not self.threads:
           # no more quotes/threads so stop the workproc
           self.StopTimer()
```

Function to Create Popup to Display Quote

```
def CreateQuotePopup(self,sym,fields):
        # if we failed to get the quote, do nothing
        if not fields:
            print "Could not get a quote for",sym
            return

        print "Creating Popup for",sym,fields
        # new toplevel window
        root=Toplevel()
        root.title("Details for %s" % sym)
        tw = Text(root,height =20, width = 39)
        tw.pack();
        text = ("Symbol", "Name", "Price", "Date", "Time", "Change",
                "Percent. Change", "Volume", "Average Volume",
                "Bid", "Ask", "Previous", "Open", "Day Range",
                "52 week range", "Earnings/Share", "Price/Earnings", "Dividend",
                "Dividend Yield", "Market Capital");
        for i in range(len(text)):
            line = "%-16s %s\n" % (text[i], fields[i])
            tw.insert('end',  line)

        b=Button(root,text = 'Dismiss', command=root.destroy )
        b.pack(side = 'bottom')
```

Main Routine

```
if __name__ == "__main__":

    root = Tk()

    app= GetQuoteApp(root)

    root.mainloop()
```

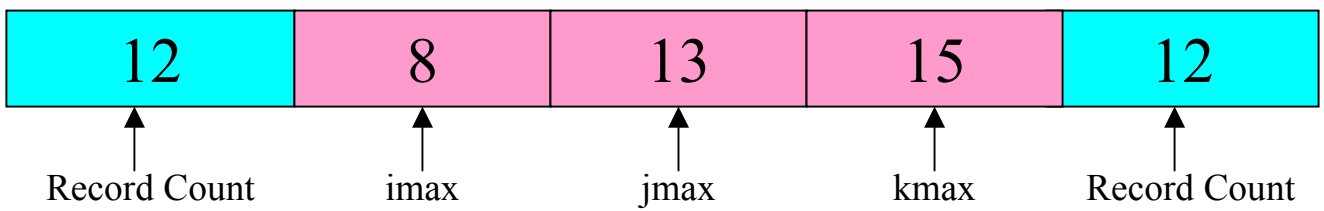
Reading/Writing Fortran Unformatted Files

- The Fortran unformatted write statement, **`write(1)`**, creates a binary record in the file
- The record has a header and trailer that contains the number of bytes in the record

  For example

```
Integer*4 imax,jmax,kmax
write(1)  imax,jmax,kmax
```

  will create actually write the following information to the file if **`imax=8, jmax=13, kmax=15`**:



Reading/Writing Fortran Unformatted Files

- To read/write Fortran unformatted files the record header must be included
- The record count can be used when reading the file to:
  - Determine if the byte order of data in the file is little or big endian
  - Determine the word size of data: 4 or 8 byte integers or floats
  - Skip over unneeded information
- We will next discuss reading/writing binary/unformatted files in Python
  - Use Fortran unformatted PLOT3D grid files as the example
    - PLOT3D grid file format for multiple grids with iblank:

```
READ(1) NGRID
READ(1) (JD(IG),KD(IG),LD(IG),IG=1,NGRID)
DO IG = 1,NGRID
 READ(1) (((X(J,K,L),J=1,JD),K=1,KD),L=1,LD),
&        (((Y(J,K,L),J=1,JD),K=1,KD),L=1,LD),
&        (((Z(J,K,L),J=1,JD),K=1,KD),L=1,LD),
&        (((IBLANK(J,K,L),J=1,JD),K=1,KD),L=1,LD)
EndDo
```

Python Read/Write Methods

- The Python read method returns a string or list of bytes

  file=open(name,”r”)

  data=file.read()

  – data is a string

- The Python write method takes as an argument a string

  file=open(name,”w”)

  file.write(“this is a string”)

Struct Module

- The Python struct module provides methods to convert data to/from numeric data and Python strings
  - **`import struct`** to make it available
  - Use the **`struct.pack(fmt,v1,v2,…)`** to convert/pack the values in **`v1, v2`**, etc. into a string using the format specified in **`fmt.`** A string is returned.
  - Use the **`struct.unpack(fmt,str)`** to convert/unpack the string **`str`** using the format specified in **`fmt.`** A tuple of the unpacked values is returned.
  - Use the **`struct.calcsize(fmt)`** to calculate the size in bytes of the structure corresponding to the format specified in **`fmt`**
  - Format string will contain a character code specifying the data type for each variable to be converted

Struct Module Format Character Codes

Format	C Type	Python Type
x	pad byte	no value
c	char	string of length 1
b	signed char	integer
B	unsigned char	integer
h	short	integer
H	unsigned short	integer
i	int	integer
I	unsigned int	long
l	long	integer
L	unsigned	long long
q	long	long long
Q	unsigned long	long long
f	float	float
d	double	float
s	char[]	string
p	char[]	string
P	void *	integer

More on Struct Module Format String

- Need a conversion code for each variable to be converted
- Can have a repeat count:

  "iiii" is the same at "4i"
- An optional first character of the format string will specify the byte ordering of the packed data
  - For example "<4i" will convert 4 integers and treat them as little-endian byte order during the conversion

Character	Byte order	Size and alignment
@	native	native
=	native	Standard = (short = 2 bytes, int,long,float = 4 bytes, double = 8 bytes)
<	little-endian	Standard
>	big-endian	Standard

Using Struct Module To Read PLOT3D Grid

- First record in the file: **READ(1) NGRID**
  - Remember the record count: one integer record has a 4 byte record count
- First read the record count:
  - Read with number of bytes specified:

```
rsize_str = file.read(4)
rsize     = struct.unpack("i",rsize_str)
```

- Read the number of grids

```
ngrids_str = file.read(4)
ngrids,    = struct.unpack("i",ngrids_str)
```

- Verify trailing record size

```
rsize_str = file.read(4)
rsize2    = struct.unpack("i",rsize_str)
if rsize2[0] != rsize[0]:
    raise "trailing record size does not match header"
```

Changes to Determine Byteorder

- First record in the file: **`READ(1) NGRID`**
  - We know that the record count should be 4
  - Convert using big and little endian to determine byte order

```
rsize_str = file.read(4)
rsize_E = struct.unpack(">i",rsize_str)
rsize_e = struct.unpack("<i",rsize_str)

# we expect rsize to be 4 as ngrids is a 32bit integer
if   rsize_E[0] == 4:
  byte_order_fmt=">"
elif rsize_e[0] == 4:
  byte_order_fmt="<"
else:
  print "wrong record size",rsize_E,rsize_e
  raise "wrong record size“
```

- Prefix subsequent formats by **`byte_order_fmt`**

Read NGRID and Determine Byteorder

```
#!/usr/bin/env python2
import sys,struct

def ReadP3DgridNgrids(file):
    rsize_str = file.read(4)
    rsize_E = struct.unpack(">i",rsize_str)
    rsize_e = struct.unpack("<i",rsize_str)
    # we expect rsize to be 4 as ngrids is a 32bit integer
    if   rsize_E[0] == 4: byte_order_fmt=">"
    elif rsize_e[0] == 4: byte_order_fmt="<"
    else:
        print "wrong record size",rsize_E,rsize_e
        raise "wrong record size"
    # read the number of grids
    ngrids_str = file.read(4)
    ngrids,     = struct.unpack(byte_order_fmt+"i",ngrids_str)
    # verify trailing record size
    rsize_str = file.read(4)
    rsize2    = struct.unpack(byte_order_fmt+"i",rsize_str)
    if rsize2[0] != 4:
        raise "trailing record size does not match header"
    return ngrids
```

OO Version; Page 1

```
#!/usr/bin/env python2

import sys,struct

class P3Dreader:
    "Class to read in a plot3d fortran unformatted multi-grid
   file"
    def __init__(self,filename):
        "Constructor opens the specified filename"
        self.filename = filename
        self.file=open(filename)
        self.byte_order_fmt=""

    def close(self):
        self.file.close()
```

OO Version; Page 2

```
def ReadNgrids(self):
    "read the number of grids in the file and determine byteorder"

    rsize_str = self.file.read(4)
    rsize_E = struct.unpack(">i",rsize_str)
    rsize_e = struct.unpack("<i",rsize_str)

    # we expect rsize to be 4 as ngrids is a 32bit integer
    if   rsize_E[0] == 4:
        self.byte_order_fmt=">"
    elif rsize_e[0] == 4:
        self.byte_order_fmt="<"
    else:
        print "wrong record size",rsize_E,rsize_e
        raise "wrong record size"

    # read the number of grids
    ngrids_str = self.file.read(4)
    ngrids,    = struct.unpack(self.byte_order_fmt+"i",ngrids_str)

    # verify trailing record size
    rsize_str = self.file.read(4)
    rsize2    = struct.unpack(self.byte_order_fmt+"i",rsize_str)

    if rsize2[0] != 4:
        raise "trailing record size does not match header"

    self.ngrids = ngrids
```

OO Version; Page 3

```
if __name__ == "__main__":

    for filename in sys.argv[1:]:
        file=P3Dreader(filename)
        file.ReadNgrids()
        file.close()
        print "%s has %d grids"%(filename,file.ngrids)
```

Read Grid Dimensions

- The second record in the Plot3d grid file contains the grid dimensions

```
READ(1) (JD(IG),KD(IG),LD(IG),IG=1,NGRID)
```

- Record size in bytes is **`NGRID*3*4`**
- Add method for P3Dreader class to read grid dimensions
  - Create a format string to unpack the data
  - Read the record and unpack the data
  - Remember a tuple containing the data is returned
  - Save the grid dimensions as object attributes

```
# Create the format with the byte order switch and a repeat count
fmt=self.byte_order_fmt+"%di"%(self.ngrids*3,)
rsize_str = self.file.read(nbytes_record)
data      = struct.unpack(fmt,rsize_str)
self.id=[0]*self.ngrids
self.jd=[0]*self.ngrids
self.kd=[0]*self.ngrids
for g in range(self.ngrids):
    self.id[g]=data[g*3]
    self.jd[g]=data[g*3+1]
    self.kd[g]=data[g*3+2]
```

Read Grid Dimensions Method

```
def ReadDimensions(self):
    "read the dimensions of each grid"
    nbytes_record=self.ngrids*3*4
    # Read the leading record size
    rsize_str = self.file.read(4)
    rsize     = struct.unpack(self.byte_order_fmt+"i",rsize_str)
    if rsize[0] != nbytes_record:
        raise  "record size does not match expected value"
    # Create the format with the byteorder switch and a repeat count
    fmt=self.byte_order_fmt+"%di"%(self.ngrids*3,)
    rsize_str = self.file.read(nbytes_record)
    data      = struct.unpack(fmt,rsize_str)
    # verify trailing record size
    rsize_str = self.file.read(4)
    rsize2    = struct.unpack(self.byte_order_fmt+"i",rsize_str)
    if rsize2[0] != nbytes_record:
        raise "trailing record size does not match header"
    self.id=[0]*self.ngrids
    self.jd=[0]*self.ngrids
    self.kd=[0]*self.ngrids
    for g in range(self.ngrids):
        self.id[g]=data[g*3]
        self.jd[g]=data[g*3+1]
        self.kd[g]=data[g*3+2]`
```

Read Grid Coordinates and IBLANK

- The next record in the Plot3d grid file contains the grid points and iblank information.
- It is repeated for each grid

```
      READ(1) (((X(J,K,L),J=1,JD),K=1,KD),L=1,LD),
     &        (((Y(J,K,L),J=1,JD),K=1,KD),L=1,LD),
     &        (((Z(J,K,L),J=1,JD),K=1,KD),L=1,LD),
     &        (((IBLANK(J,K,L),J=1,JD),K=1,KD),L=1,LD)
```

- Record size in bytes is
  - For no IBLANK information: **`JD*KD*LD*3*WordSize`**
  - With IBLANK information: **`JD*KD*LD*(3*WordSize+4)`**
  - Where the **`WordSize`** is 4 for single precision and 8 for double precision
- Add method for P3Dreader class to read grid coordinates and iblank
  - Read the reader header and determine
    - If the grid points are single or double precision
    - If IBLANK information is present
  - Create a format string to unpack the data
  - Read the record and unpack the data
  - Save the grid data as object attributes

Use Record Size to Determine Word Size

- Compute the possible sizes for the record

```
npoints=self.id[grid]*self.jd[grid]*self.kd[grid]
# calculate the possible sizes of the components of the record
single_size=npoints*3*4
double_size=npoints*3*8
iblank_size=npoints*4
```

- Compare the actual record size with the possible sizes to determine word size and if file contains IBLANK

```
have_iblank=0
if   rsize[0] == single_size:
    self.grid_wordsize=4
elif rsize[0] == double_size:
    self.grid_wordsize=8
elif rsize[0] == single_size+iblank_size:
    self.grid_wordsize=4
    have_iblank=1
elif rsize[0] == double_size+iblank_size:
    self.grid_wordsize=8
    have_iblank=1
else:
    raise "record size does not match expected value"
```

Read the Grid Coordinate Record

```
if   self.grid_wordsize == 4:
    fmt=self.byte_order_fmt+"%df"%(npoints*3,)
elif self.grid_wordsize == 8:
    fmt=self.byte_order_fmt+"%dd"%(npoints*3,)
data_str = self.file.read(npoints*3*self.grid_wordsize)
self.xyz[grid] = struct.unpack(fmt,data_str)

if have_iblank:
    # read the iblank information
    # Create the format with the byteorder switch and a repeat count
    fmt=self.byte_order_fmt+"%di"%(npoints,)
    data_str = self.file.read(npoints*4)
    self.iblank[grid] = struct.unpack(fmt,data_str)
```

Read Method

- Add a read() method to read the whole file
- Complete module is in **read_p3d_oo.py**

```
    def read(self):
        file.ReadNgrids()
        file.ReadDimensions()
        for grid in range(self.ngrids):
            self.ReadXYZIblank(grid)

if __name__ == "__main__":

    for filename in sys.argv[1:]:
        file=P3Dreader(filename)
        file.read()
        file.close()
        print "%s has %d grids"%(filename,file.ngrids)
        for g in range(file.ngrids):
            print "(%d,%d,%d)"%(file.id[g],file.jd[g],file.kd[g])
```

Two-Dimensional Graphics and Plotting

- There are MANY packages available for Python for 2D graphics
  - Low level drawing
    - Tkinter (included with Python)
    - Python Imaging Library (PIL): Image manipulation library including drawing
      - http://www.pythonware.com/products/pil/
    - Piddle
      - http://piddle.sourceforge.net
    - PyOpenGL
  - PyStripchart Widget Library: Display time-sampled data in a strip chart
    - http://jstripchart.sourceforge.net/
  - Interface to gnuplot: gnuplot.py
    - http://gnuplot-py.sourceforge.net
  - DISLIN Scientific Data Plotting Software
    - High level plotting library: Curves, polar plots, bar graphs, pie charts, 3D-color plots, surfaces, contours, maps
    - http://www.linmpi.mpg.de/dislin/
  - Chaco (Part of the SciPy project)
    - http://www.scipy.org/site_content/chaco
  - Pyscript (Create Postscript graphics)
    - http://pyscript.sourceforge.net/

Drawing With Tkinter

- Tkinter is included with Python
- Has a Canvas class for drawing, displaying images and text
  - Canvas method requires a parent widget
    - Must be drawn on the screen
  - Can save as a Postscript file
    - Requires conversion from Postscript to standard image formats
    - May need to call update() to wait for window to be sized

```
canvas.update()
canvas.postscript(file="output.ps")
```

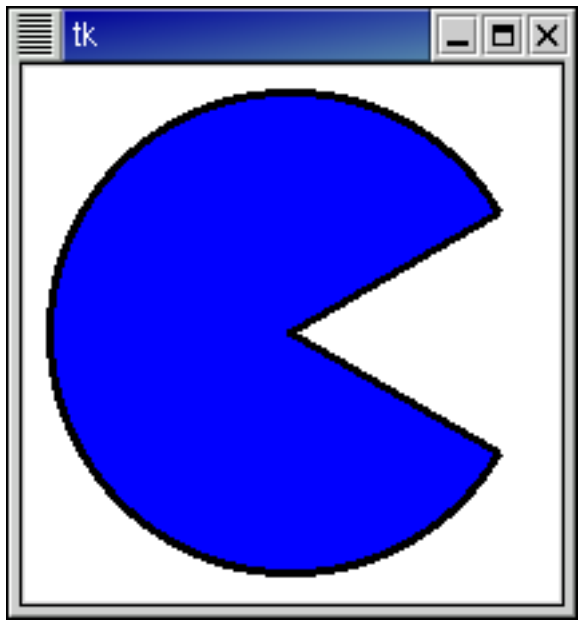
  - Various draw methods have common optional attributes:
    - fill='red', width=2, …

```
#!/usr/bin/env python2
from Tkinter import *
root=Tk()
canvas=Canvas(root,width=200,height=200,bg='white')
canvas.pack()
root.mainloop()
```

Drawing With Tkinter: Arc

- create_arc() draws an arc/oval shaped region bounded by start angle and end at start+extent angle

```
canvas.create_arc(10,10,190,190,outline='black',
                  fill='blue',
                  start=30.0,extent=300.0,width=3)
```



Drawing With Tkinter: Line

- create_line() draws a line with one or more segments
  - Screen Coordinate system:
    - Units are pixels
    - X increases from left to right
    - Y increases from TOP to BOTTOM
  - Can provide set of points as arguments or a list of coordinates where each item in the list is a tuple of the x,y pair

```
canvas.create_line(x0,y0,x1,y1,…,xn,yn,fill="red",
   outline="black",width=3)
canvas.create_line(100,250,400,250,width=2,arrow='last')
n=50
xmax = 2.0*math.pi
dx=xmax/n
XY=[0]*n
for i in range(n):
    x=i*dx
    y=math.sin(x)

    XY[i]=(100+x*300.0/xmax,150-y*100)

canvas.create_line(XY,width=2,fill='red')
```

Drawing With Tkinter: Rectangles

- create_rectangle() draws a rectangle at a specific location
  - Screen Coordinate system:
    - Units are pixels
    - X increases from left to right
    - Y increases from TOP to BOTTOM
  - Optionally specify
    - fill (color)
    - outline (specifies a color for drawing the outline)
    - width (width of the outline)

```
canvas.create_rectangle(x0,y0,x1,y1,fill="red",
   outline="black",width=3)
```

Drawing With Tkinter: Polygon

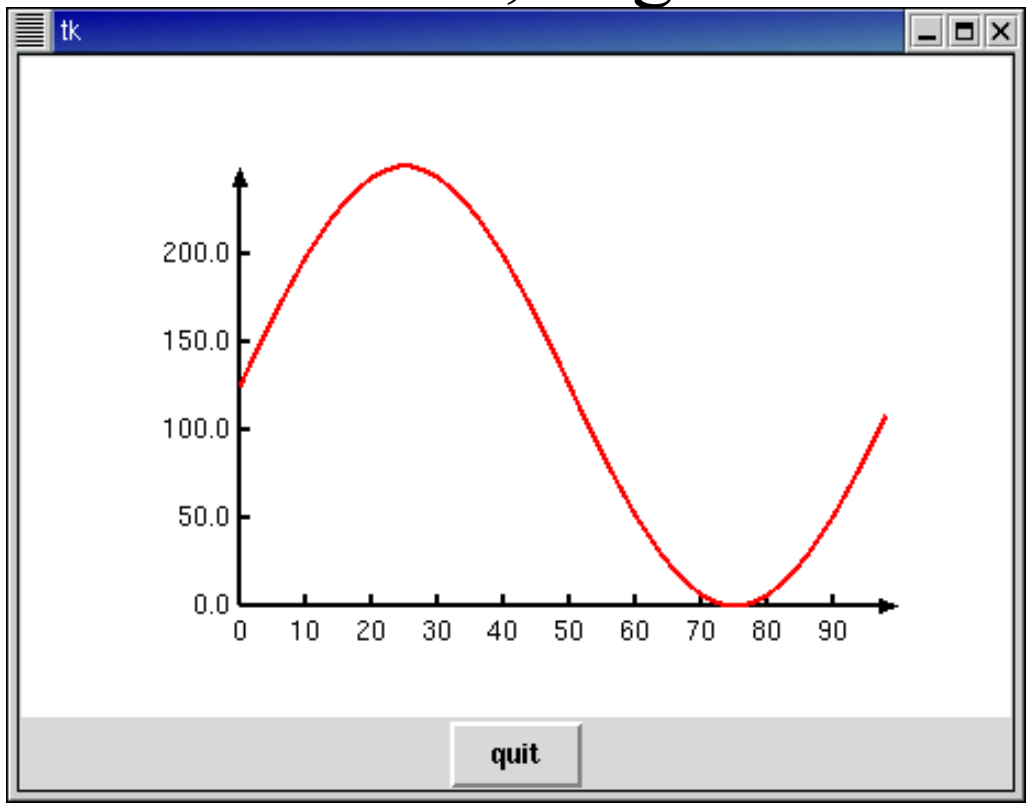
- create_polygon() draws a polygon at a specific location
  - Screen Coordinate system:
    - Units are pixels
    - X increases from left to right
    - Y increases from TOP to BOTTOM
  - Optionally specify
    - fill (color)
    - outline (specifies a color for drawing the outline)
    - width (width of the outline)
    - smooth (boolean to indicate if the boundary should be drawn as a parabolic spline)
    - splinesteps (number of line segments used to approximate split)
  - For multiple segments you can provide a list of coordinates where each item in the list is a tuple of the x,y pair

```
canvas.create_polygon(x0,y0,x1,y1,…,xn,yn,fill="red",
   outline="black",width=3)
XY=[(x0,y0),(x1,y1)]
canvas.create_polygon(XY,fill="red", outline="black",width=3)
```

Example with Lines and Text; Page 1

```
#!/usr/bin/env python2
#tkinter_line.py

import math
from Tkinter import *

root=Tk()

canvas=Canvas(root,width=450,height=300,bg='white')
canvas.pack()
Button(root,text='quit',command=root.quit).pack()
canvas.create_line(100,250,400,250,width=2,arrow='last')
canvas.create_line(100,250,100,50,width=2,arrow='last')

for i in range(10): # xticks
    x=100+i*30
    canvas.create_line(x,250,x,245,width=2)
    canvas.create_text(x,255,text='%d'%(10*i,),anchor=N)

for j in range(5): # yticks
    y=250-j*40
    canvas.create_line(100,y,105,y,width=2)
    canvas.create_text(96,y,text='%5.1f'%(50.0*j,),anchor=E)
```

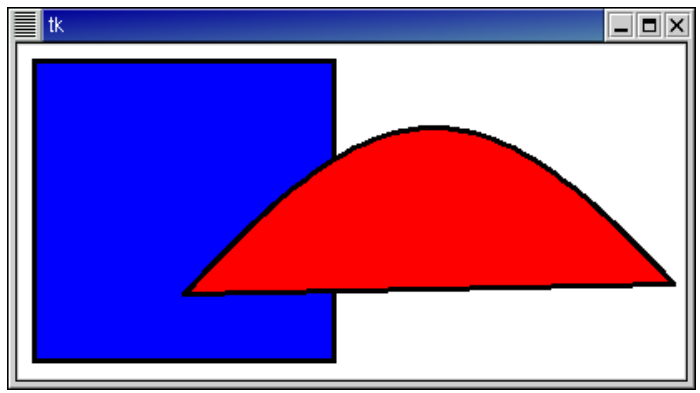
Example with Lines and Text; Page 2




```
n=50
xmax = 2.0*math.pi
dx=xmax/n
XY=[0]*n
for i in range(n):
    x=i*dx
    y=math.sin(x)

    XY[i]=(100+x*300.0/xmax,150-y*100)

canvas.create_line(XY,width=2,fill='red')
canvas.update()
canvas.postscript(file="line.ps")
root.mainloop()
```

Example with Rectangle and Polygon

```
#!/usr/bin/env python2
import math
from Tkinter import *
root=Tk()
canvas=Canvas(root,width=400,height=200,bg='white')
canvas.pack()
canvas.create_rectangle(10,10,190,190,
                        outline='black',fill='blue',
                        width=3)
n=50
xmax = math.pi
dx=xmax/n
XY=[0]*n
for i in range(n):
    x=i*dx
    y=math.sin(x)
    XY[i]=(100+x*300.0/xmax,150-y*100)

canvas.create_polygon(XY,outline='black',fill='red',width=3)
root.mainloop()
```



Python Imaging Library (PIL)

- PIL adds image processing capabilities to your Python programs
- Has extensive file format support
- Fairly powerful image processing capabilities.
- Obtain it from:

http://www.pythonware.com/products/pil/

- Free with commercial support available

A Few Possible Uses of PIL

- Image Archives
  - Ideal for image archival and batch processing applications.
  - Create thumbnails
  - Convert between file formats
  - Print images
  - Current version identifies and reads a large number of formats
  - Write support is intentionally restricted to the most commonly used interchange and presentation formats.
- Image Display
  - The current release includes Tk PhotoImage and BitmapImage interfaces
- Image Processing
  - Library contains some basic image processing functionality
  - Point operations
  - Filtering with a set of built-in convolution kernels
  - Color space conversions
  - Image resizing
  - Rotation
  - Arbitrary affine transforms
  - A histogram method produces some statistics of an image

Loading/Creating an Image Using PIL

- Load an existing image with **image.open(filename)**
- Creates a new image with **image.new(mode, size)**
- The mode of an image defines the type and depth of a pixel in the image.
  - 1 (1-bit pixels, black and white, stored as 8-bit pixels)
  - L (8-bit pixels, black and white)
  - P (8-bit pixels, mapped to any other mode using a colour palette)
  - RGB (3x8-bit pixels, true colour)
  - RGBA (4x8-bit pixels, true colour with transparency mask)
  - CMYK (4x8-bit pixels, colour separation)
  - YCbCr (3x8-bit pixels, colour video format)
  - I (32-bit integer pixels)
  - F (32-bit floating point pixels)
  - RGBX (true colour with padding)
  - RGBa (true colour with premultiplied alpha).
- Size is given as a 2-tuple specifying the horizontal and vertical size in pixels.

Loading an Image Using PIL

- Creates a new image with specified background color
  - **image.new(mode, size, color)**
- **Color** is
  - A single value for single-band images
  - A tuple for multi-band images (with one value for each band).
  - If the color argument is omitted, the image is filled with black.
  - If the color is None, the image is not initialized.

Drawing Using PIL

- The ImageDraw module provide basic graphics support for Image objects.
- It can be used to
  - Create new images
  - Annotate or retouch existing images
  - Generate graphics on the fly
- Draw methods are very similar to the Tkinter Canvas routines
- Coordinate system is x increases from left to right, y increases from top to bottom.

```
import Image
import ImageDraw
# load an image
im = Image.open("lena.pgm")
# Creates an object that can be used
# to draw in the given image.
# Note that the image will be modified in place.
draw = ImageDraw.Draw(im)
```

Drawing Using PIL: arc and pieslice

- **draw.arc(xy, start, end, options)**
  - Draws an arc (a portion of a circle outline) between the start and end angles, inside the given bounding box.
    - xy specifies the bounding box as an array of tuples
      - [(x0,y0),(x1,y1)]
    - The fill option gives the color to use for the arc.

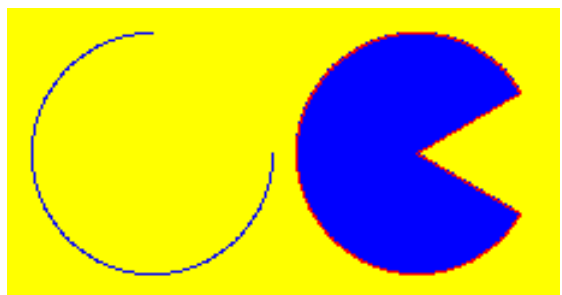
```
draw.arc([x0,y0,x1,y1],0,270,fill="blue“)
```

- **draw.pieslice(xy, options)**
  - Same as arc, but also draws straight lines between the end points and the center of the bounding box.
  - The fill option gives the color to use for the interior.
  - The outline option gives the color for the bounding curbe

```
draw.pieslice([x0,y0,x1,y1],30,330,fill="blue",
  outline='red')
```

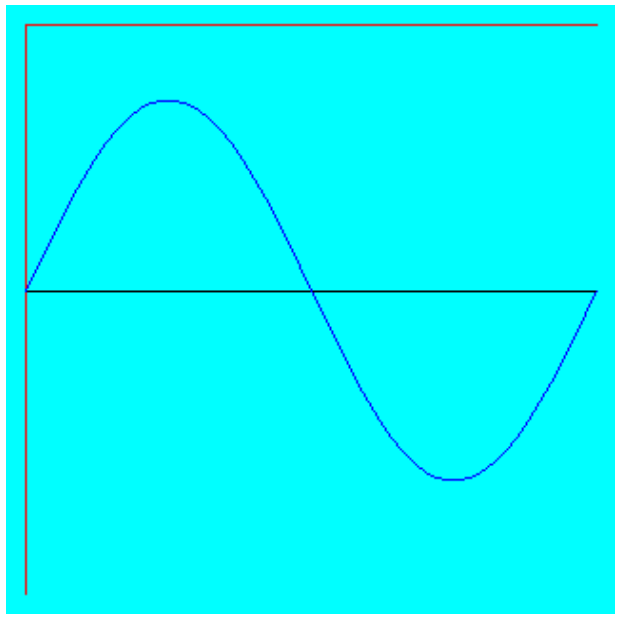
Drawing Using PIL: arc and pieslice

```
#! /usr/bin/env python2
import sys,os
sys.path.append("/home/noack/local/Python/PIL")
import Image, ImageDraw
import math,string,re

image=Image.new("RGB",(230,120),(256,256,0))
draw=ImageDraw.Draw(image)

x0=10; y0=10; x1=110; y1=110
draw.arc([x0,y0,x1,y1],0,270,fill="blue")

x0=120; x1=220
draw.pieslice([x0,y0,x1,y1],30,330,fill="blue",outline='red')
#line = [(x0,y0),(x1,y1)]
#draw.line(line,fill=0)

file=open("pil_arc.png","w")
image.save(file,"PNG")
file.close()
```



Drawing Using PIL: line

- **draw.line(xy, options)**
  - Draws a line between the coordinates in the xy list.
  - The coordinate list can be any sequence object containing either
    - 2-tuples [(x, y), ...] or numeric values [x, y, ...].
    - It should contain at least two coordinates.
  - The fill option gives the color to use for the line.
  - Coordinate system is left to right, top to bottom



Drawing Using PIL: line

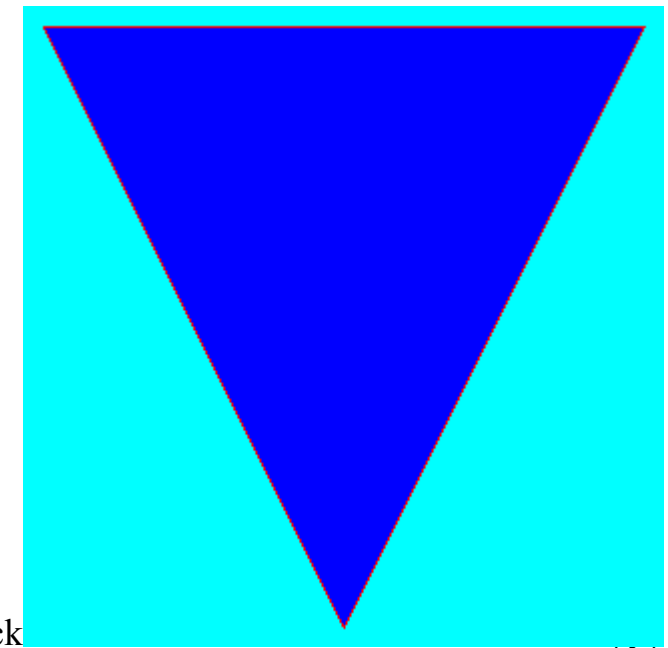
```
#! /usr/bin/env python2
import sys,os
sys.path.append("/home/noack/local/Python/PIL")
import Image, ImageDraw
import math,string,re
image=Image.new("RGB",(320,320),(0,256,256))
draw=ImageDraw.Draw(image)
xo=10;yo=10
draw.line( [(xo+300,yo),(xo,yo),(xo,yo+300)] ,fill='red')
draw.line( [xo,150,xo+300,150] ,fill='black')
n=51
xmax = 2.0*math.pi
dx=xmax/(n-1)
XY=[0]*n
for i in range(n):
    x=i*dx; y=math.sin(x)
    XY[i]=(10+x*300.0/xmax,150-y*100)
draw.line(XY,fill='blue')
file=open("pil_line.png","w")
image.save(file,"PNG")
file.close()
```

Drawing Using PIL: polygon

- **draw.polygon(xy, options)**
  - Draws a polygon bounded by the coordinates in the xy list.
  - The coordinate list can be any sequence object containing either
    - 2-tuples [(x, y), ...] or numeric values [x, y, ...].
    - It should contain at least two coordinates.
  - The fill option gives the color to use for the interior of the polygon.
  - The outline option gives the color to use for the boundary of the polygon.
  - Coordinate system is left to right, top to bottom

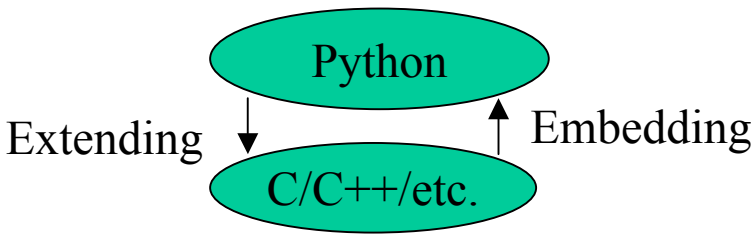
Drawing Using PIL: line

```
#! /usr/bin/env python2
import sys,os
sys.path.append("/home/noack/local/Python/PIL")
import Image, ImageDraw
import math,string,re
image=Image.new("RGB",(320,320),(0,256,256))
draw=ImageDraw.Draw(image)
draw.polygon( [10,10, 310,10, 160,310, 10,10] ,
 fill='blue',outline="red")
file=open("pil_poly.png","w")
image.save(file,"PNG")
file.close()
```



Combining Python With Other Languages

- There are two ways to use Python with other languages such as C/C++/FORTRAN.
  - Embedding
    - The other language accesses the Python interpreter
    - Use Python to provide an extension language to your program
  - Extension writing
    - Extend Python with new capabilities written in another language
      - Add new modules which access routines written in other language
    - Provides speed of compiled language



- We will discuss extension writing to provide access to routines written in another language

Interfacing Python To Another Languages

- Python objects are not directly usable in other language and vice versa
- Wrapper functions are required to translate data between the two languages
  - Python will call the wrapper function
  - Wrapper function will
    - Convert arguments from Python object to desired data type
    - Call the desired function in the other language
    - Create a Python object for any return values
    - Return the Python object to Python

- Documentation can be found at
  - http://www.python.org/doc/current/ext/ext.html
- Our extention will be imported as a module

Example Extension Module

- Have a C function to calculate the factorial of a number

```
int factorial(int n)
{
  if (n < 2) return 1;
  return (n*factorial(n-1));
}
#ifdef TESTING
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
int main(int argc,char **argv)
{
  int n;
  n=atoi(argv[1]);
  printf("Factorial of %d is %d\n",n,factorial(n));
  return 0;
}
#endif
```

Example Extension Module

- We will create a wrapper function that allows us to call the factorial function from Python
  - We will call our wrapper function **`factorial_wrap`**
  - File is called **factorial_wrap_byhand.c**
- It will be a function within a module called MyModule
  - Python code function code will be **MyModule.factorial(n)**
- First line of code should be to include "Python.h"
- Next include header file with function prototype for the factorial function: **`int factorial(int n);`**

```
#include "Python.h"
#include "factorial.h"
```

Wrapper Function Arguments

- The wrapper function
  - Returns a PyObject
  - Has two arguments:
    - **`PyObject *self`**
      - The self argument is only used when the C function implements a built-in method
      - We are defining a function, not a method, so self will always be a NULL pointer
    - **`PyObject *args`**
      - The arguments passed to the Python call to the function

```
#include "Python.h"
#include "factorial.h"
PyObject *factorial_wrap(PyObject *self,PyObject *args)
{
}
```

Extract C Data Types From Arguments

- We must extract C data types from the **args** arguments
  - **PyArg_ParseTuple** is used to convert non-keyword arguments
  - **PyArg_ParseTupleAndKeywords** is used to convert keyword arguments
  - Use a format string to determine how to convert the Python object
    - "i" for integer
    - "f" for float
    - "d" for double
    - Etc.

```
#include "Python.h"
#include "factorial.h"
PyObject *factorial_wrap(PyObject *self,PyObject *args)
{
  int n;
  if (! PyArg_ParseTuple(args,"i",&n)){return NULL;}
}
```

Call The C code

- We then call the function from our C code using the value retrieved from the argument

```
#include "Python.h"
#include "factorial.h"
PyObject *factorial_wrap(PyObject *self,PyObject *args)
{
  int n,result;
  if (! PyArg_ParseTuple(args,"i",&n) ) {return NULL;}
  /* call my function with the argument retrieved from python */
  result = factorial(n);
}
```

Return Value as PyObject

- Finally must convert the return value from our function to a PyObject and return it
  - Use a format to specify how to convert the value

```
#include "Python.h"
#include "factorial.h"
PyObject *factorial_wrap(PyObject *self,PyObject *args)
{
  int n,result;
  if (! PyArg_ParseTuple(args,"i",&n) ) {return NULL;}
  /* call my function with the argument retrieved from python */
  result = factorial(n);
  /* return the result after building the python object */
  return Py_BuildValue("i",result);
}
```

Build Method Table

- Must tell Python
  - The name of the methods as called from Python
  - The wrapper function address
  - A flag specifying the calling convention
    - **METH_VARARGS** or **METH_KEYWORDS** or bitwise OR of the two: **METH_VARARGS | METH_KEYWORDS**

```
#include "Python.h"
#include "factorial.h"
PyObject *factorial_wrap(PyObject *self,PyObject *args)
{
……
}
static PyMethodDef MyModuleMethods[]= {
  { "factorial",factorial_wrap, METH_VARARGS},
  { NULL,NULL}, /* sentinel value to terminate list */
};
```

Module Initialization Function

- Need a function for Python to call when initializing the module
  - Function is called init*MODULENAME*
  - Pass it the method table

```
#include "Python.h"
#include "factorial.h"
PyObject *factorial_wrap(PyObject *self,PyObject *args)
{
……
}
static PyMethodDef MyModuleMethods[]= {
  { "factorial",factorial_wrap, METH_VARARGS},
  { NULL,NULL}, /* sentinel value to terminate list */
};
 void initMyModule()
{
  Py_InitModule("MyModule",MyModuleMethods);
}
```

Compiling the Extension Module

- We will create a shared object file for the wrapper and the C code.
  - Allows the module to be dynamically loaded by Python interpreter
  - Creates a file called **MyModule.so**
    - Must be placed in **PYTHONPATH**
  - Makefile

```
SO=             .so
LDSHARED=     gcc -fPIC -shared
CCFLAGS_SHARED=      -fPIC
PYTHON_INC =  -I/usr/include/python2.2

CC = gcc
CFLAGS = -g -Wall $(CCFLAGS_SHARED)  $(PYTHON_INC)

OBJS = factorial.o factorial_wrap_byhand.o

MyModule$(SO):  $(OBJS)
   $(LDSHARED)  $(OBJS)   -o $@
```

Using the Extension Module

- Make it available by **`import MyModule`**

```
>>> import MyModule
>>> fact=MyModule.factorial(5)
>>> print fact
120
>>> dir(MyModule)
['__doc__', '__file__', '__name__', 'factorial']
>>>
```

Extend Python with FORTRAN Using Pyfort

- Pyfort is a tool for connecting Fortran routines (and Fortran-like C) to Python
- You write a function/subroutine prototype/specification
- Pyfort generates the wrapper function for interfacing Fortran with Python
- Requires Numerical Python array extension for arrays
- Uses Tkinter and Python megawidgets (Pmw) for GUI project file editor
  - Home for Pmw is: http://pmw.sourceforge.net
- Obtain it from http://sourceforge.net/projects/pyfortran
- Will work with several different Fortran compilers
  - g77, Portland Group (pgf77,pgf90),
  - Have a mechanism to add other compilers

Obtaining Numerical Python

- Numerical Python add fast and compact multidimensional arrays to Python
- Used by several other extension modules
  - PyOpenGL, Pyfort,…
- Obtain from:
  - http://www.pfdubois.com/numpy
  - http://sourceforge.net/projects/numpy
- Two versions
  - Numeric is the older more stable release
    - Current version is 23.1
  - Numarray is a rewrite of Numeric and will replace Numeric
    - Current version is 0.7

Installing Numerical Python

- We'll install the packages in ~/local/Python
  - They will be in ~/local/Python/lib/python
- Need to add this directory to the PYTHONPATH variable

setenv PYTHONPATH ${PYTHONPATH}:/${HOME}/local/Python/lib/python

- Extract the source in a directory

  tar xzvf Numeric-23.1.tar.gz

- Build and install in the specified directory

  cd Numeric-23.1

  python2 setup.py build

  python2 setup.py install --home=${HOME}/local/Python

Installing Pyfort

- Obtain it from http://sourceforge.net/projects/pyfortran
- Pyfort will install itself and your modules in a system directory (/usr/lib/python….) unless you specify an alternate location
- Installing it in our home does not require root privaledges
  - Add appropriate directory to the PYTHON PATH

**setenv PYTHONPATH **
${PYTHONPATH}:${HOME}/local/Python/lib/python2.2/site-packages/

  - Edit configuration.py
    - Set **prefix='/*your_home_dir*/local/Python/'**
- setup.py install --prefix=~/local/Python
  - Will install Pyfort and your modules in

~/local/Python/lib/python2.2/site-packages

- Pyfort utility is installed in: **~/local/Python/bin/**

Using Pyfort

- Pyfort takes a ”module file” that specifies the interface or function prototype
- Creates
  - A C source file that provides the glue between the Fortran code and Python
  - A text file describing the interface
- Can also have a project file to build/install/uninstall several Pyfort modules
  - Pyfort has a gui to help create the project file

Pyfort Module File

- The Pyfort ”module file” is named ***module_name*.pyf**
- Creates and installs a module called ***module_name***
- Contains the interface/prototype specification for one or more Fortran routines.
  - Describes the input and return values
  - Syntax is similar to modern Fortran
- The module file is case-insensitive.
- Routines will be called from Python using lower case.
- Python doc strings will be generated from appropriate comments

Pyfort Example: Fortran Routine

```
C This routine works only for x sorted
      function percentile(d, n, x)
          integer n
          real x(n), d
          real percentile
          if (d .lt. x(1)) then
              percentile = 0.0
              return
          elseif (d. gt. x(n)) then
              percentile = 100.0
              return
          else
              do 100 i = 1, n
                  if (x(i) .ge. d) then
                      percentile = (i*100.0/n)
                      return
                  endif
100           continue
          endif
      end
```

Example Interface Specification

```
function percentile(d, n, x)
  ! Given a sorted x(n), calculate the percentile of observation d
 integer n = size(x)
 real x(n), d, percentile
end function percentile
```

- Start with **`function`** or **`subroutine,`** function name and argument list
- Comment begins with !
  - Comment right after **`function`** or **`subroutine`** line becomes the doc string
- Specify the type and dimension of the arguments and the return value of the function
  - **`integer n = size(x)`** declares that the array size should be calculated and not passed as an argument
- Will call from Python as module.percentile(d,x)

Fortran Arguments declared intent(out)

- Arguments declared as out will not be part of the Python call list

```
subroutine mysub (x, y, n)
  integer n
  real x (n)
  real, intent (out):: y (n)
end
```

- Will call from Python as **`y = mysub(x, len (x))`**
- Can further simplify

```
subroutine mysub (x, y, n)
  integer n = size(x)
  real x (n)
  real, intent (out):: y (n)
end
```

- Will call from Python as **`y = mysub(x)`**

Example Using Pyfort

- We'll examine the demo that comes with Pyfort
- Percentile function and module interface from previous slides
- Run pyfort to create C code, compile and install

  ~/local/Python/bin/pyfort -i pyfdemo
- Create Python code to use the pyfdemo module

```
#!/usr/bin/env python2
import pyfdemo
print dir(pyfdemo)
score = 75.0
# create list of scores
n=11
scores=[0]*n
for i in range(n):
    scores[i]=(i-1)*10
print pyfdemo.percentile.__doc__
pctile=pyfdemo.percentile(score,scores)
print pctile
```

Automated Tools to Extend Python

- Hand generation of wrapper can be tedious and error prone
  - Need to update wrapper if function calls change
  - A library may contain hundreds of functions
- SWIG - generate C/C++ interface code for Python
  - http://www.swig.org
- SIP - generate C++ interface code for Python
  - http://www.riverbankcomputing.co.uk/sip
- Boost.Python - wrapping C++ classes with a Python interface
  - http://www.boots.org/libs/python/doc
- Fortran to Python interface generator: F2PY
  - http://cens.ioc.ee/project/f2py2e
- Pyrex - a language for writing Python extension modules
  - http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex
- PyInline/Weave - embed C/C++/Perl code inside your Python script
  - http://pyinline.sourceforge.net

SWIG

- Simple Wrapper Interface Generator
  - A compiler that turns ANSI C/C++ declarations into scripting languages interfaces
    - Not a full C/C++ compiler
  - Fully automated
    - Eliminates/Reduces writing of extension modules by hand
    - Produces a useable module
  - Language Neutral
    - Targets Python, Tcl, Perl, MATLAB, etc.

SWIG Interface File

- Input file contains ANSI C/C++ declarations and special SWIG directives
- SWIG directives are always preceded by a "%" to distinguish them from normal C declarations
  - Specify the module name

    %module mymodule
  - Anything between **%{** and **%}** is included verbatim into the wrapper code

    ```
    %{
    #include "myheader.h"
    %}
    ```
- Include ANSI C/C++ directives of functions/variables to be included in the module

Building Module

- Best to use a Makefile to automate the steps
  - Generate wrapper code

swig -includeall -python p3dlib_all.i

  - Compile the wrapper code and other source code
  - Generate position-independent code, suitable for use in a shared library (gcc –fpic)

CC = gcc

CFLAGS = -g -Wall –fPIC

PYTHON_INC = -I/usr/include/python2.2

$(CC) $(CFLAGS) $(PYTHON_INC) -c p3dlib_all_wrap.c -o p3dlib_all_wrap.o

  - Link into shared library
    - gcc -shared

LDSHARED = gcc –shared

$(LDSHARED) $(P3DLIB_OBJS) p3dlib_all_wrap.o -o _p3dlib_all.so

  - Copy **_p3dlib_all.so p3dlib_all.py** into directory in PYTHONPATH

Other Useful SWIG directives

- %init %{ ... %} inserts code into the module initialization function.
  - Variable initialization, call to initialize a library/device, etc.
- %inline %{ ... %} inserts code into the header section and "wraps" it.
  - Allows you to insert hand written interfaces that may not be in the library
  - Useful for returning multiple values as a tuple

```
//file: p3dlib.i
%module p3dlib
%{
#include "p3dlib.h"
%}
struct p3d_strGridIO *
p3d_LoadP3dFile(char *filename,int wordsize);
%inline %{
PyObject * p3d_GetDimsGridFromSRDR(struct p3d_strGridIO *srdr,int g)
{
return Py_BuildValue("iii",
   srdr->grids[g]->imax,srdr->grids[g]->jmax,srdr->grids[g]->kmax);
}
%}
```

SWIG directives in C/C++ Header File

- SWIG defines the preprocessor symbol SWIG
  - Put SWIG directives in C/C++ header file
  - Use #ifdef SWIG to isolate swig directives when compiling the header in C/C++ code

```
#ifdef SWIG
%module p3dlib
%{
#include "p3dlib.h"
%}
#endif /* def SWIG */
```

Using P3Dlib as Python Module

- An example of using the wrapped P3Dlib within Python
- Python code will extract edges of all the grids in a Plot3d grid file and write them as separate files

extract_edgesp3d.py: Page 1

```
#! /usr/bin/env python2
from p3dlib import *
filename="test.p3du"

# create the Structured grid ReaDeR object
# store grid as double precision
srdr = p3d_LoadP3dFile_vector_xyz(filename,8)

#get the number of zones/grids in the file
nz = p3d_GetNumGrids(srdr)
```

extract_edgesp3d.py: Page 2

```
# write each zone as a separate file
# name appends _#
# write file type is p3dwibud:
# plot3d, with iblank, fortran unformatted, double prec
ext = "p3dwibud"
format,style,mod,form,prec,endian = p3d_GetFormatFromStdExt(ext)

for i in range(nz):
    # get pointer to the zone
    zptr=p3d_GetStructuredGridFromSRDR(srdr,i)
    imax,jmax,kmax= p3d_GetGridDimTuple(zptr)

    iminedge=p3d_ExtractSubGrid(zptr,   1,   1, 1,jmax, 1,1)
    imaxedge=p3d_ExtractSubGrid(zptr,imax,imax, 1,jmax, 1,1)

    jminedge=p3d_ExtractSubGrid(zptr,1,imax,    1,   1, 1,1)
    jmaxedge=p3d_ExtractSubGrid(zptr,1,imax, jmax,jmax, 1,1)
    output_filename = filename + "_" + `i` + "." + ext
    # create the output object
    osrdr  = p3d_New_p3d_strGridIO(output_filename, format,8)
```

extract_edgesp3d.py: Page 3

```
for i in range(nz):
    ……
    osrdr  = p3d_New_p3d_strGridIO(output_filename, format,8)

    # copy edges to the output
    p3d_AddGrid2SRDR(osrdr,iminedge)
    p3d_AddGrid2SRDR(osrdr,imaxedge)
    p3d_AddGrid2SRDR(osrdr,jminedge)
    p3d_AddGrid2SRDR(osrdr,jmaxedge)

    # set the output to the desired format
    p3d_Set_Plot3d_Writer(osrdr,output_filename,format);

    # write the file (also closes it )
    p3d_WriteStructGridFile(osrdr)
```

F2PY: Fortran to Python Interface Generator

- The F2PY Fortran to Python interface generator provides a connection between Python and Fortran languages.
- F2PY is a Python package that facilitates creating/building Python C/API extension modules
  - Provides ability to call Fortran 77/90/95 external subroutines and Fortran 90/95 module subroutines as well as C functions
  - Provides access to Fortran 77 COMMON blocks and Fortran 90/95 module data, including allocatable arrays from Python.
  - Like SWIG but for Fortran
  - F2PY uses scipy distutils that contains support for a number of Fortran 77/90/95 compilers
- Web site is: http://cens.ioc.ee/projects/f2py2e/

F2PY: Steps to Produce Module

- Wrapping Fortran or C functions to Python using F2PY consists of three steps
  - Creating the signature file that contains descriptions/prototypes of wrappers to Fortran or C functions.
    - For Fortran routines, F2PY can create initial version of a signature file by scanning Fortran source codes
    - F2PY created signature files can be edited to optimize wrappers functions, make them smarter and more Pythonic
    - F2PY directives can be added to the Fortran source to reduce editing of the signature files
  - F2PY reads a signature file and constructs the Fortran/C/Python bindings as a Python C/API module. "
  - F2PY compiles all sources and builds an extension module containing the wrappers.
- For simple cases the steps can be combined and run with one command

Installing F2PY

- Download and extract the latest source
  - Prerequisites:
    - Python (versions 1.5.2 or later; 2.1 and up are recommended)
    - NumPy/Numeric (versions 13 or later; 20.x, 21.x, 22.x, 23.x are recommended)
    - Numarray (version 0.4.4 and up), optional, partial support.
    - Fortran compiler
- Installation:

python2 setup.py install –home=~/local/Python

  - Build and install in home directory
  - Installs
    - **~/local/Python/bin/f2py2**
    - **~/local/Python/lib/python/f2py2e/**

Simple Example: FIB1.F

```
C FILE: FIB1.F
      SUBROUTINE FIB(A,N)
C
C     CALCULATE FIRST N FIBONACCI NUMBERS
C
      INTEGER N
      REAL*8 A(N)
      DO I=1,N
         IF (I.EQ.1) THEN
            A(I) = 0.0D0
         ELSEIF (I.EQ.2) THEN
            A(I) = 1.0D0
         ELSE
            A(I) = A(I-1) + A(I-2)
         ENDIF
      ENDDO
      END
C END FILE FIB1.F
```

Simple Example: FIB1.F

- Run F2PY to create module

~/local/Python/bin/f2py2 -c fib1.f -m fib1

- Use the module. Use Numeric module to create array

```
>>> import fib1
>>> dir(fib1)
['__doc__', '__file__', '__name__', '__version__',
   'as_column_major_storage', 'fib',
   'has_column_major_storage']
>>> import Numeric
>>> a=Numeric.zeros(8,'d')
>>> fib1.fib(a)
>>> print a
[  0.   1.   1.   2.   3.   5.   8.  13.]
>>> print fib1.fib.__doc__
fib - Function signature:
  fib(a,[n])
Required arguments:
  a : input rank-1 array('d') with bounds (n)
Optional arguments:
  n := len(a) input int
```

F2PY Modes: Generate a Signature File

- F2PY has three major modes selected by command line options
  - To scan Fortran sources and generate a signature file

f2py -h *signature_filename.pyf list_fortran_files*

~/local/Python/bin/f2py2 -h junk.pyf fib1.f

```
$ cat junk.pyf
!    -*- f90 -*-
subroutine fib(a,n) ! in fib1.f
    real*8 dimension(n) :: a
    integer optional,check(len(a)>=n),depend(a) ::
  n=len(a)
end subroutine fib

! This file was auto-generated with f2py
  (version:2.37.233-1545).
! See http://cens.ioc.ee/projects/f2py2e/
```

F2PY Modes: Create module code

- To construct an extension module
  - For Fortran90 modules

  f2py ***list_fortran_files***

  - For Fortran77

  f2py -m ***module_name list_fortran_files***

  - Creates a file called **module_name**module.c

  ~/local/Python/bin/f2py2 -m fib1 fib1.f

  Creates the file **fib1module.c**

F2PY Modes: Build module

- To build an extension module

  f2py -c *list_fortran_files*

  – Creates a file called **untitled.so**

  – To specify the module name to be created use the **–m** flag

  ~/local/Python/bin/f2py2 -c -m fib1 fib1.f

  Creates the file **fib1.so**

Python Access to Fortran Common Blocks

- F2PY generates wrappers to common blocks
- In Python, the F2PY wrappers to common blocks are Fortran type objects that have (dynamic) attributes related to data members of common blocks.
- When accessed, these attributes return as Numeric array objects (multi-dimensional arrays are Fortran-contiguous) that directly link to data members in common blocks.
- Data members can be changed by direct assignment or by in-place changes to the corresponding array objects.

F2PY Directives

- The signature files may need to be changed to properly reflect the intent of each argument
- Rather than editing the signature file F2PY directives can be placed in the Fortran source code
  - An F2PY directive has the following form:
    - <comment char>f2py ...
    - Where allowed comment characters for fixed and free format Fortran codes are cC*!# and !, respectively
    - Directive is ignored by the Fortran compiler
  - Examples:
    - Cf2py intent(in) n
    - Cf2py intent(out) a

F2PY Directives

- intent(<intentspec>) arg_list: specifies the intention of the arguments in the list.
  - <intentspec> is a comma separated list of keys
    - The two most common are:
      - in: The argument is considered as an input only argument.
      - out: The argument is considered as an return variable.
  - Cf2py intent(in) a
  - Cf2py intent(out) r
- depend([<names>]) arg_list: declares that the corresponding argument depends on the values of variables in the list <names>
  - Cf2py depend(n) a
- optional: The corresponding argument is moved to the end of <optional arguments> list. A default value for an optional argument can be specified
  - Cf2py integer optional,intent(in) :: n = 13

F2PY Directives: Example FIB3.F

```
C FILE: FIB3.F
      SUBROUTINE FIB(A,N)
C
C     CALCULATE FIRST N FIBONACCI NUMBERS
C
      INTEGER N
      REAL*8 A(N)
Cf2py intent(in) n
Cf2py intent(out) a
Cf2py depend(n) a
      DO I=1,N
         IF (I.EQ.1) THEN
            A(I) = 0.0D0
         ELSEIF (I.EQ.2) THEN
            A(I) = 1.0D0
         ELSE
            A(I) = A(I-1) + A(I-2)
         ENDIF
      ENDDO
      END
C END FILE FIB3.F
```

F2PY Directives: Example FIB3.F

```
>>> import fib3
>>> y=fib3.fib(10)
>>> print y
[  0.   1.   1.   2.   3.   5.   8.  13.  21.  34.]
```

F2PY Directives: Example FTYPE.F

```
C FILE: FTYPE.F
      SUBROUTINE FOO(N)
      INTEGER N
Cf2py integer optional,intent(in) :: n = 13
      REAL A,X
      COMMON /DATA/ A,X(3)
      PRINT*, "IN FOO: N=",N," A=",A,"
     X=[",X(1),X(2),X(3),"]"
      END
C END OF FTYPE.F
```

F2PY Directives: Example FTYPE.F

```
>>> import ftype
>>> ftype.foo()
 IN FOO: N=          13 A=  0.0000000E+00 X=[  0.0000000E+00  0.0000000E+00
   0.0000000E+00]
>>> dir(ftype)
['__doc__', '__file__', '__name__', '__version__',
   'as_column_major_storage', 'data', 'foo',
   'has_column_major_storage']
# data is the common block
>>> ftype.data.x[0]=0
>>> ftype.data.x[0]=1
>>> ftype.data.x[1]=2
>>> ftype.data.x[2]=3
>>> ftype.foo(5)
 IN FOO: N=           5 A=  0.0000000E+00 X=[   1.000000       2.000000
   3.000000    ]
```

VPython

- VPython is a 3D graphics module that is exceptionally easy to use
- Can create simple 3D objects and position them in space
- VPython handles updating the 3D scene many times a second to reflect the current positions of objects
  - Simplifies animating objects
- Programmer can focus on the computational aspects and does not need to deal with display management.
- Navigation/Viewpoint can be controlled by using the mouse to zoom and rotate.

Obtaining and Installing VPython

- Prerequisites:
  - Python2.2 or higher
  - Gtk+1.2
  - Gtkglarea
  - Pkg-config
  - OpenGL
  - These could be already installed in a Redhat installation
- Web site: http://www.vpython.org/
- Extract tar archive
- Set python executable to use

  setenv PYTHON /usr/bin/python2.2
- Configure

  ./configure --prefix=${HOME}/local/Python
- **Make**

  make install

VPython Objects

- Cylinder

  cylinder(pos=(0,2,1), axis=(5,0,0), radius=1)
- Arrow: straight box-shaped shaft with an arrowhead at one end

  arrow(pos=(0,2,1), axis=(5,0,0), shaftwidth=1)
- Cone: circular cross section and tapers to a point

  cone(pos=(5,2,0), axis=(12,0,0), radius=1)
- Pyramid: rectangular cross section and tapers to a point

  pyramid(pos=(5,2,0), size=(12,6,4))
- Sphere: Position is the center of the Sphere

  sphere(pos=(1,2,1), radius=0.5)

VPython Objects

- Ring: circular, with a specified outer radius and thickness, and with its center given by the pos attribute, in plane normal to axis:

  ring(pos=(1,1,1), axis=(0,1,0), radius=0.5, thickness=0.1)
- Box: Position is the center of the box

  mybox = box(pos=(x0,y0,z0), length=L, height=H, width=W)
- Curve: Straight lines between points

  curve(pos=[(0,0),(0,1),(1,1),(1,0),(0,0)])
- Convex: Convex hull of a list of points
- Label: display text in a box
- Faces: primitive that takes a list of triangles (position, color, and normal for each vertex)
- Frame: used to group objects together to make a composite object that can be moved and rotated together

VPython Faces

- The VPython faces primitive takes a list of triangles
  - The triangle is specified by the position, color, and normal for each vertex
  - The pos, color, and normal attributes are lists containing the data
    - First three vertices are for the first triangle, etc.
    - Clockwise cyclic order of the three vertices determines the lighted side of the face
  - Need to use a frame to have motion
- The Vector object is not a display object but simplifies the 3D computations
  - **vector(x,y,z) r**eturns a vector object with the given components
  - Vectors can be added or subtracted from each other, or multiplied by an ordinary number. For example,

v1 = vector(1,2,3)

v2 = vector(10,20,30)

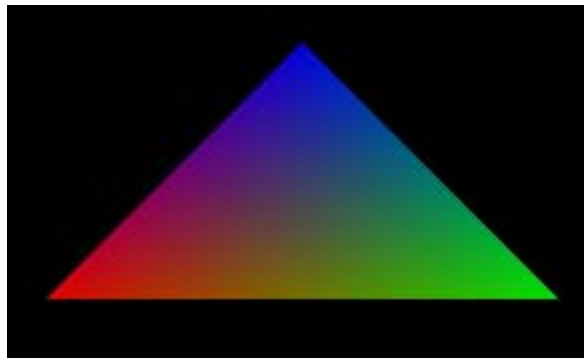
print v1+v2 # displays (11 22 33)

print 2*v1 # displays (2 4 6)

  - You can refer to individual components of a vector:

v2.x is 10, v2.y is 20, v2.z is 30

VPython Faces Example



```
#!/usr/bin/env python2
# vpython_faces.py
import sys,string,re,math
from visual import *
f = frame(axis=(1,0,0))
triangle = faces( frame = f )
# create the vertices
v1 = vector(0.0,0.0,0.0)
v2 = vector(1.0,0.0,0.0)
v3 = vector(0.5,0.5,0.0)
# calculate the normal to the triangle
normal = norm( cross(v2-v1, v3-v1) )
# add the triangle vertices to our faces object
triangle.append( pos=v1, color=(1,0,0), normal=normal )
triangle.append( pos=v2, color=(0,1,0), normal=normal )
triangle.append( pos=v3, color=(0,0,1), normal=normal )
# animate it
dt=0.01; velocity=vector(-1,-1,0)
dth=dt*2.0*math.pi/10.0
while 1:
    rate(30)
    triangle.pos = triangle.pos + velocity*dt
    # rotates about current position
    f.rotate(angle=dth)
```

VPython Example

- VPython comes with many demo programs
- We will create one to display an unstructured surface mesh and put the object in motion
  - Read a file containing the triangular surface mesh
  - Create a VPython object using faces() to display the object
  - Use a frame to animate it
- File is flex format
  - :xyz num_nodes
    - Followed by x(i) y(i) z(i) for each node

    :xyz 158279

    0.41076500E+01 0.35345000E+00 0.35345000E+00
  - :connectivity_faces num_faces num_nodes_face num_nei
    - Followed by (node_index(j,iface),j=1,num_nodes_face), (nei(j,iface),j=1,num_nei)

    :connectivity_faces 13924 3 2 1 13924

    712 711 145 1 -3

VPython Example: Page 1

```
#!/usr/bin/env python2
# vpython_gen_store.py
import sys,string,re,math
from visual import *

f = frame(axis=(1,0,0))
bomb = faces( frame = f )
```

VPython Example: Page 2

```
def FacetedTriangle(obj,v1, v2, v3, color=None):
    """add a triangle to the object"""
    try:
        normal = norm( cross(v2-v1, v3-v1) )
    except:
        normal = vector(0,0,0)
    for v in (v1,v2,v3):
        obj.append( pos=v, color=color, normal=normal )
```

VPython Example: Page 3

```
def ReadXYZ(file,n):
    print "Reading xyz"
    xyz=[0]*n
    for i in xrange(n):
        line=file.readline()
        (x,y,z)=string.split(line)
        tri = vector(float(x),float(y),float(z))
        xyz[i]=tri
    print "Done"
    return xyz
```

VPython Example: Page 4

```
def ReadConnectivityFaces(file,nfaces,nnodes_face,xyz,obj):
    print "Reading connectivity"
    if nnodes_face != 3:
        raise "nnodes_face != 3"
    for i in xrange(nfaces):
        line=file.readline()
        fields=string.split(line)
        i1=int(fields[0])-1
        i2=int(fields[1])-1
        i3=int(fields[2])-1
        FacetedTriangle(obj,xyz[i1], xyz[i2], xyz[i3],
                        color=(1,0,0))
    print "Done"
```

VPython Example: Page 5

```
file=open("gen_store.flex","r")
while 1:
    line=file.readline()
    if not line:
        break

    if line[0] == ":":
        print line
    if line[0:4] == ':xyz':
        kw,nxyz=string.split(line)
        xyz=ReadXYZ(file,int(nxyz))
    if re.match(':connectivity_faces',line):
        fields=string.split(line)
        nfaces=int(fields[1])
        nnodes_face=int(fields[2])

   ReadConnectivityFaces(file,nfaces,nnodes_face,xyz,bomb)
file.close()
print "Done"
```

VPython Example: Page 6

```
dt=0.01
velocity=vector(-1,-1,0)
dth=dt*2.0*math.pi/10.0
while 1:
    rate(30)
    bomb.pos = bomb.pos + velocity*dt
    # rotates about current position
    f.rotate(angle=dth)
```

PyOpenGL

- PyOpenGL is the Python binding to OpenGL and related API's
  - Includes support for OpenGL, GLU, GLUT, GLE, WGL, and Togl (Tk OpenGL widget)
  - Available from http://pyopengl.sourceforge.net/
  - Also available is OpenGL Context, a teaching and testing library
- Only minor differences between OpenGL and Python bindings in PyOpenGL
- Requirements
  - Python 2.2.x or greater
  - OpenGL 1.1 and GLU (included with most Linux distributions)
  - GLUT 3.7+ (included with most Linux distributions)
  - Numeric Python (numpy) v22 (or greater if building from source)

What is OpenGL?

- OpenGL is a low level 3D graphics library
  - Intended to be a software interface to the graphics hardware
    - Current consumer oriented hardware is very fast and cheap
      - Excellent performance for the price
  - Does not include any commands for windowing tasks
    - Other software must
      - Create a window to draw in
      - Handle user interactions via keyboard and pointer
  - Programmer has complete control over generation and update/redraw of the display
    - Other graphics toolkits may setup view port, update loop, etc.
    - OpenGL requires the programmer to integrate update with the user interface

Installing PyOpenGL From Source

- Assume have already installed Numeric Python
- Extract source

  tar xzvf PyOpenGL-2_0_0_44_tar.gz

  cd PyOpenGL-2.0.0.44/
- edit config/linux.cfg
  - Make sure include_dirs and library_dirs has path to X11 files.
  - Redhat 7.3 need to add /usr/X11R6/{include,lib}

  include_dirs=/usr/include:/usr/X11/include:/usr/X11R6/include

  library_dirs=/usr/lib:/usr/X11/lib:/usr/X11R6/lib
- Build and install in home directory

  python2 setup.py build

  python2 setup.py install --home=${HOME}/local/Python

PyOpenGL Demo programs

- Installation includes lots of demo scripts

  cd ~/local/Python/lib/python/OpenGL/Demo/

Advanced Capabilities

- There are MANY modules in the standard library that we have not talked about
- MANY packages are available for Python that extend the capability
  - Other 2D and 3D graphics
  - pyMPI: parallel programming using MPI
  - Statistics
  - Chemistry
  - LOTS more

Conclusion

- Hope you have gained enough knowledge to start writing python programs
- The additional details can be picked up as you begin to use it
- There are a LOT of modules that people have written and made available
  - A web search on python and most any topic will find something useful
- Call/email me if you have questions
- Good luck