Introduction to Python

Jon Macey

http://nccastaff.bournemouth.ac.uk/jmacey/Python/index.html
Python

• python is a very flexible programming language, it can be used in a number of different ways.

• Most of our animation packages allow for embedded python scripting

• We can also write complex programs which run standalone, and if written correctly can run on all operating systems
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess. There should be one and preferably only one obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
import antigravity

http://xkcd.com/353/
Lecture Series Outline

- Some basic python commands and techniques
- Interaction with the operating system
- Reading and Writing data to files
- Object Orientation in Python
- Some basic python for the major animation packages
Getting started

• At it’s simplest level python can be used as a simple command interpreter

• We type python into the console and we get a prompt which lets us enter commands

• If nothing else we can use this as a basic calculator

• It is also useful for trying simple bits of code which we wish to put into a larger system
Keywords

• The following identifiers are keywords in python and must not be used as identifiers

```plaintext
and    del    from    not    while
as     elif    global   or     with
assert else    if       pass    yield
break  except  import   print
class  exec    in       raise
continue  finally  is      return
def    for    lambda  try
```
Data Types

- Python is a dynamically typed language, this means that variable values are checked at run-time (sometimes known as “lazy binding”).

- All variables in Python hold references to objects, and these references are passed to functions by value.

- Python has 5 standard data types
  - numbers, string, list, tuple, dictionary
Numbers

- Python supports four different numerical types:
  - int (signed integers)
  - long (long integers [can also be represented in octal and hexadecimal])
  - float (floating point real values)
  - complex (complex numbers)
Strings

- Python strings are immutable
- Python allows for either pairs of single or double quotes
- Subsets of strings can be taken using the slice operator ( [ ] and [ : ] ) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end
- The plus ( + ) sign is the string concatenation operator, and the asterisk ( * ) is the repetition operator.

```python
#!/usr/bin/python
str = 'Hello_python'

# Prints complete string
print str

# Prints first character of the string
print str[0]

# Prints characters starting from 3rd to 6th
print str[2:5]

# Prints string starting from 3rd character
print str[2:]

# Prints string two times
print str * 2

# Prints concatenated string
print str + "with_added_text"
```
Lists

- A list is the most common of the Python data containers / types.
- It can hold mixed data, include lists of lists.
- A list is contained within the [] brackets and is analogous to C arrays.
- Like a string data is accessed using the slice operator ( [ ] and [ : ] ) with indexes starting at 0 in the beginning of the list and working their way to end-1.
- The + operator concatenates and the * duplicates.

```python
#!/usr/bin/python
data = [123, "hello", 2.45, 3+2j]
moreData=["", "world"]

print data
print data[1]
print data[2:]

hello=data[1]+moreData[0]+moreData[1]
print hello
```

```
./list.py
[123, 'hello', 2.4500000000000002, (3+2j)]
hello
[2.4500000000000002, (3+2j)]
hello world
```
Tuples

- A tuple can be thought of as a read only list.
- It uses parenthesis to contain the list data.

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

data = (123, "hello", 2.45, 3+2j)
moreData= ("\", "world")

print data
print data[1]
print data[2:]

hello=data[1]+moreData[0]+moreData[1]
print hello
data+="more"

TypeError: can only concatenate tuple (not "str") to tuple
```
more on slice operators

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

a=range(0,10)
print "a[::2]",a[::2]
print "a[::1]",a[::1]
print "a[1:10:2]",a[1:10:2]
print "a[1:-1:1]",a[1:-1:1]
del a[::2]
print "del a[::2]",a
print range(10)[slice(0, 5, 2)]
```

```
a[:2] [0, 2, 4, 6, 8]
a[::1] [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
a[1:10:2] [1, 3, 5, 7, 9]
a[-1:1] [0, 1, 2, 3, 4, 5, 6, 7, 8]
del a[:2] [1, 3, 5, 7, 9]
[0, 2, 4]
```
Python Dictionaries

• Python dictionaries are a powerful key / value data structure which allows the storing of different data types in the same data set

• It is similar to an associative array or hash map in other programming languages

• Many Python API’s use dictionaries to store values and variable length function parameters
Create a dictionary of colour lists “key”:[r,g,b]

Use the .get(“key”) method to find the value

Note “None” returned if “key” not found
Type Conversion

- Python allows type conversion via a number of functions, the most common are

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int(x, base)</td>
<td>Converts x to an integer. base specifies the base if x is a string.</td>
</tr>
<tr>
<td>long(x, base)</td>
<td>Converts x to a long int. base specifies the base if x is a string.</td>
</tr>
<tr>
<td>float(x)</td>
<td>Converts x to a float.</td>
</tr>
<tr>
<td>complex(real, img)</td>
<td>Generate a complex number</td>
</tr>
<tr>
<td>str(x)</td>
<td>Converts x to a string representation</td>
</tr>
</tbody>
</table>
#!/usr/bin/python

intText="12"
floatText="0.23123"
intData=123

a=int(intText)
b=float(floatText)
text=str(intData)

print a, type(a)
print b, type(b)
print text, type(text)

./convert.py
12 <type 'int'>
0.23123 <type 'float'>
123 <type 'str'>
Python Membership Operators

- There are two membership operators in python “in” and “not in”
- These can be used to test for membership in lists, tuples and strings

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

data = (123, "hello", 2.45, 3+2j)
numbers=[1,2,3,4,5]
print "world" in data False
print "text" not in numbers True
print 99 in numbers False
print 2 in numbers True
```
Programming Constructs

• Most programming tasks can be split into a combination of the following elements
  • Sequences
  • Selection
  • Iteration
  • Whenever I learn a new language I see how these are represented syntactically as this makes learning the language easier.
Sequences

• As the name suggest a sequence is a fixed set of instructions
• They are always carried out in the same order
• With the use of functions we can bundle other sequences together to make programs easier to read / maintain
• The following example shows this in action
#!/usr/bin/python
from turtle import *
def Square(_size):
    forward(_size)
    left(90)
    forward(_size)
    left(90)
    forward(_size)
    left(90)
    forward(_size)
    penup()
goto(10,20)
pendown()
Square(40)
penup()
goto(50,200)
pendown()
Square(100)
penup()
goto(300,100)
pendown()
pendown()
Square(200)
done()
Python White Space rules

- Python uses white space to delimit scope, it can use either tabs or spaces
- Mixing the two can become problematic however we can still do the following

```python
>>> foo = [
    ...     'some_string',
    ...     'another_string',
    ...     'short_string'
    ... ]
>>> print foo
['some_string', 'another_string', 'short_string']

>>> bar = 'this is' \ 
    ...     'one_long_string' \ 
    ...     'that_is_split' \ 
    ...     'across_multiple_lines'
>>> print bar
this is one long string that is split across multiple lines
```

- for an in depth analysis see [http://www.secnetix.de/olli/Python/block_indentation.hawk](http://www.secnetix.de/olli/Python/block_indentation.hawk)
Python functions

• In python functions are actually values, this means we can pass functions around like variables

• Python functions also allow for multiple return types (unlike C/C++) this means there is no pass by value / reference type constructs

• Functions are declared using the def keyword and uses the : to indicate the body of the function which must be indented
```python
#!/usr/bin/python

def multiReturn(_data):
    a = _data * 1
    b = _data * 2
    c = _data * 3
    return a, b, c

data = ['test', 'values']
a, b, c = multiReturn(data)
print a
print b
print c
```

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

def foo(_data):
    print "foo", _data

def bar(_data):
    print "bar", _data

functions = [foo, bar]
functions[0](12)
functions[1](12)
functions[0](99)
functions[1](88)
```

`['test', 'values']`
`['test', 'values', 'test', 'values']`
`['test', 'values', 'test', 'values', 'test', 'values']`

foo 12
bar 12
foo 99
bar 88
selection

• selections allow us to make choices
• most programming languages has at least the if else construct
• some languages have more
• The result of an if operation is a boolean (true / false) value and code is executed or not depending upon these value
• In python we use the following constructs
#!/usr/bin/python

from turtle import *

type = ""

if type == "Square":
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
    done()

elif type == "Triangle":
    forward(100)
    right(120)
    forward(100)
    right(120)
    forward(100)
    done()

else:
    print "nothing_selected"
## Python Comparison Operators

Given `a=10 b=20`

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>equality operator returns true if values are the same</td>
<td><code>(a==b)</code> is not true</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Checks if the value of two operands are equal or not</td>
<td><code>(a!=b)</code> is true</td>
</tr>
<tr>
<td><code>&lt;&gt;</code> (now obsolescent)</td>
<td>Checks if the value of two operands are equal or not</td>
<td><code>(a&lt;&gt;b)</code> is true</td>
</tr>
<tr>
<td>`&gt;</td>
<td>Checks if the value of left operand is greater than the value of right operand</td>
<td><code>(a&gt;b)</code> is not true</td>
</tr>
<tr>
<td>`&lt;</td>
<td>Checks if the value of left operand is less than the value of right operand</td>
<td><code>(a&lt;b)</code> is true</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Checks if the value of left operand is greater than or equal to the value of right operand</td>
<td><code>(a&gt;=b)</code> is not true</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Checks if the value of left operand is less than or equal to the value of right operand</td>
<td><code>(a&lt;=)</code> is true</td>
</tr>
</tbody>
</table>
# Python Logical Operators

\[ a = 10 \text{ and } b = 20 \]

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>Logical and</td>
<td>( a \text{ and } b \text{ is true} )</td>
</tr>
<tr>
<td>or</td>
<td>Logical or</td>
<td>( a \text{ or } b \text{ is true} )</td>
</tr>
<tr>
<td>not</td>
<td>Logical not</td>
<td>( \text{not } (a \text{ and } b) \text{ is false} )</td>
</tr>
</tbody>
</table>
selection

- selections can be embedded to create quite complex hierarchies of “questions”

- This can sometimes make reading code and maintenance hard especially with the python white space rules as code quite quickly becomes complex to read

- We usually prefer to put complex sequences in functions to make the code easier to read / maintain
iteration

- iteration is the ability to repeat sections of code
- Python has two main looping constructs
  - for each
  - while
- For-each loops operate on ranges of data
- While loops repeat while a condition is met
```python
#!/usr/bin/python
from turtle import *
def Square(_size):
    forward(_size)
    left(90)
    forward(_size)
    left(90)
    forward(_size)
    left(90)
    forward(_size)
    for x in range(-250, 250, 40):
        goto(x, 0)
        Square(40)
done()
```

the range function produces a list for x in ... assigns each list element to x in turn
```python
#!/usr/bin/python
# code taken from
# http://docs.python.org/dev/library/turtle.html

from turtle import *

color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

here we loop forever and use a condition to see if we are finished then break out of the loop
looping for x,y

- This example shows how we can loop from -10 in the x and y in increments of 0.5.
- In C / C++ we would use a for loop.

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

x=-10.0
y=-10.0

while y<=10.0 :
    while x<=10.0 :
        print x,y
        x+=0.5
    y+=0.5
x=-10.0
```
alternative loop

```python
#!/usr/bin/python
n = ((a, b) for a in range(0, 5) for b in range(0, 5))
for i in n:
    print i
```
## Built-in Functions

<table>
<thead>
<tr>
<th>abs()</th>
<th>divmod()</th>
<th>input()</th>
<th>open()</th>
<th>staticmethod()</th>
</tr>
</thead>
<tbody>
<tr>
<td>all()</td>
<td>enumerate()</td>
<td>int()</td>
<td>ord()</td>
<td>str()</td>
</tr>
<tr>
<td>any()</td>
<td>eval()</td>
<td>isinstance()</td>
<td>pow()</td>
<td>sum()</td>
</tr>
<tr>
<td>basestring()</td>
<td>execfile()</td>
<td>issubclass()</td>
<td>print()</td>
<td>super()</td>
</tr>
<tr>
<td>bin()</td>
<td>file()</td>
<td>iter()</td>
<td>property()</td>
<td>tuple()</td>
</tr>
<tr>
<td>bool()</td>
<td>filter()</td>
<td>len()</td>
<td>range()</td>
<td>type()</td>
</tr>
<tr>
<td>bytearray()</td>
<td>float()</td>
<td>list()</td>
<td>raw_input()</td>
<td>unichr()</td>
</tr>
<tr>
<td>callable()</td>
<td>format()</td>
<td>locals()</td>
<td>reduce()</td>
<td>unicode()</td>
</tr>
<tr>
<td>chr()</td>
<td>frozenset()</td>
<td>long()</td>
<td>reload()</td>
<td>vars()</td>
</tr>
<tr>
<td>classmethod()</td>
<td>getattr()</td>
<td>map()</td>
<td>repr()</td>
<td>xrange()</td>
</tr>
<tr>
<td>cmp()</td>
<td>globals()</td>
<td>max()</td>
<td>reversed()</td>
<td>zip()</td>
</tr>
<tr>
<td>compile()</td>
<td>hasattr()</td>
<td>memoryview()</td>
<td>round()</td>
<td><strong>import</strong>()</td>
</tr>
<tr>
<td>complex()</td>
<td>hash()</td>
<td>min()</td>
<td>set()</td>
<td>apply()</td>
</tr>
<tr>
<td>delattr()</td>
<td>help()</td>
<td>next()</td>
<td>setattr()</td>
<td>buffer()</td>
</tr>
<tr>
<td>dict()</td>
<td>hex()</td>
<td>object()</td>
<td>slice()</td>
<td>coerce()</td>
</tr>
<tr>
<td>dir()</td>
<td>id()</td>
<td>oct()</td>
<td>sorted()</td>
<td>intern()</td>
</tr>
</tbody>
</table>
enumerate

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

colours=['red','green','blue','black','white']

c=list(enumerate(colours))
print c

c=list(enumerate(colours,start=2))
print c

[(0, 'red'), (1, 'green'), (2, 'blue'), (3, 'black'), (4, 'white')]
[(2, 'red'), (3, 'green'), (4, 'blue'), (5, 'black'), (6, 'white')]
```
set / frozenset

- A set object is an unordered collection of immutable values.
- Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.
- Sets may be added to, frozen sets may not, however both types may be compared against each other.
#!/usr/bin/python

a=range(0,5)
a*=2
print a
b=set(a)
print b

a =set([1,2,3,4])
b =set([3,4,5,6])
print "a=",a
print "b=",b
print "union\n_a\n|\nb",a | b
print "intersection\n_a\n&\nb" ,a & b
print "subset\nfalse\n_a<b",a < b
print "difference\na-b",a - b
print "Symmetric\ndiff\na^b",a ^ b
This example shows the inherent instability of floating point calculations.
Programming

• There are many more constructs and techniques we need to apply to create complex programs
• but for now the basic techniques illustrated will be used in most of our code
• For the rest of the lecture we are going to look at how we can execute our own scripts within the different Operating systems we use
A trip back in time

- Early electronic computing (pre 80’s) didn’t have the GUIs we have today.
- This meant that all interactions with the computer were done with typing into a terminal.
- Most modern operating systems still have the option to do this
- And in some cases this method is quicker than using the GUI (but does require some additional knowledge)
Example

• If we wish to rename every file in the tree opposite in a GUI we would have to click on every file and type the new name.

• Some Operating Systems allow the automation of GUI tasks but this is still time consuming.

• The answer in most cases is to use another GUI program or to write a script.

• Most scripting languages let us access the underlying os commands to do this.
The Shell

- In windows we can access the command prompt (shell) by typing cmd in the start menu.
- In linux we can open a shell by clicking on the shell icon (but if you a real linux user there will be one open all the time!)
- We can then start typing commands, however windows and Unix have different commands for the same action.
<table>
<thead>
<tr>
<th>Command's Purpose</th>
<th>MS-DOS</th>
<th>Linux</th>
<th>Basic Linux Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copies files</td>
<td>copy</td>
<td>cp</td>
<td>cp thisfile.txt /home/thisdirectory</td>
</tr>
<tr>
<td>Moves files</td>
<td>move</td>
<td>mv</td>
<td>mv thisfile.txt /home/thisdirectory</td>
</tr>
<tr>
<td>Lists files</td>
<td>dir</td>
<td>ls</td>
<td>ls</td>
</tr>
<tr>
<td>Clears screen</td>
<td>cls</td>
<td>clear</td>
<td>clear</td>
</tr>
<tr>
<td>Deletes files</td>
<td>del</td>
<td>rm</td>
<td>rm thisfile.txt</td>
</tr>
<tr>
<td>Finds a string of text in a file</td>
<td>find</td>
<td>grep</td>
<td>grep this word or phrase thisfile.txt</td>
</tr>
<tr>
<td>Creates a directory</td>
<td>mkdir</td>
<td>mkdir</td>
<td>mkdir directory</td>
</tr>
<tr>
<td>View a file</td>
<td>more</td>
<td>less[d]</td>
<td>less thisfile.txt</td>
</tr>
<tr>
<td>Renames a file</td>
<td>ren</td>
<td>mv</td>
<td>mv thisfile.txt thatfile.txt[e]</td>
</tr>
<tr>
<td>Displays your location in the file system</td>
<td>chdir</td>
<td>pwd</td>
<td>pwd</td>
</tr>
<tr>
<td>Changes directories with a specified path (absolute path)</td>
<td>cd pathname</td>
<td>cd pathname</td>
<td>cd /directory/directory</td>
</tr>
<tr>
<td>Changes directories with a relative path</td>
<td>cd ..</td>
<td>cd ..</td>
<td>cd ..</td>
</tr>
</tbody>
</table>
Environment Variables

- When we open a shell we are placed in our home directory
- This place is stored in an Environment variable called
  - $HOME on unix and mac
  - %HOMEPATH% on windows

```
1  echo $HOME
2  echo %HOMEPATH%
3
4  /Users/jmacey
5  \Users\jmacey
```
Environment Variables

- Environment variables are global system variables available to all processes (i.e. programs)
- Most operating systems have a number of default values set which programs can query to set the way things operate.
- Users can also set their own environment variables to customise how things work.
- It is not uncommon for software packages to install their own environment variables when the program is installed.
Environment Variables

- The PATH environment variable allows us to set a directory where the OS will look for scripts and programs.
- We can add a local directory to our system which contains user scripts which can be executed by the user.
- The configuration is different for both Windows and Unix.
Unix Environment variables

• The default shell used in the linux studios is the bash shell (Bourne again Shell)
• To set environment variable in this shell we use a file called .bashrc which is hidden in the home directory
• if you type gedit ~/.bashrc you can access it
• if you re-open the shell this will be made permanent
• Now any program placed in this directory may be found and executed

1 export PATH=$PATH:$HOME/scripts
Windows Environment Variables

• Setting environment variables in windows is different from Unix as we have to use the GUI

• In our studios we can access them from the control panel and students have admin rights to do so

• The following panels show the way to do this
Select the system variable called Path

Click on the edit button and the following dialog will be displayed
• At the end of the Variable value line add the following

```
1 ;%HOMEDRIVE%%HOMEPATH%scripts
```

• The ; is a separator for the different values
The scripts directory

• Now we have told the system to look in the scripts directory for any scripts to run we need to create this directory

• To do this in the console we do the following where the mkdir command makes a directory

```
// Windows

3 cd %HOMEPATH%
4 mkdir scripts

// Unix

6 cd
7 mkdir scripts
```
Type the above in an editor (or your choice) and save it in the scripts directory as hello.py

In unix issue the following command in the same directory

```
chmod 755 hello.py
```

now from any directory you should be able to type hello.py to run the script
#!/usr/bin/python
import os

for env in os.environ:
    print "Variable=%s\nValue=%s" % (env, os.environ.get(env))

print os.environ.get("PATH")
The main function

• The main function is a special function for most programming languages
• It is the first function to be executed and is the entry point for most programs
• The main function is usually passed a set of global system variables called arguments
• These are available through the life of the program and are a good way of passing values to a program
#!/usr/bin/python
import sys

def main(argv=None):
    print "in_main_function"

if __name__ == "__main__":
    sys.exit(main())
Command Line args

- When a program is executed from the command line the whole line typed is passed to the program using the variable argv
- argv is a text string array split based on white space
- The following program show how we can print these values out
#!/usr/bin/python

import sys

def main(argv=None):
    if argv is None:
        argv = sys.argv
    for args in argv:
        print(args)

if __name__ == "__main__":
    sys.exit(main())
The getopt function is used to process a list of arguments in the form

- `-l` or `-vfx` will be split into `-v -f -x`
- `-f` [optional argument]
- `--help` (known as a long option)

The programmer passes a list of these options and the getopt function will split them (any additional command line values will be ignored)
processing arguments

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

import getopt, sys

def usage()
    print "to use the program pass -l for long mode"
    print "-f [name] for file mode"

class Usage(Exception):
    def __init__(self, msg):
        self.msg = msg

print "Unknown Option"

def main(argv=None):
    if argv is None:
        argv = sys.argv
    try:
        try:
            opts, args = getopt.getopt(argv[1:], "hlf:", ["help","long","file="])
        except getopt.error, msg:
            raise Usage(msg)
        except Usage, err:
            print >>sys.stderr, err.msg
            print >>sys.stderr, "for help use --help"
            return 2
        for opt, arg in opts:
            if opt in ("-l", "--long"):
                print "long mode"
            elif opt in ("-h", "--help"):
                usage()
                return
            elif opt in ("-f", "--file"):
                print "File Mode name passed",arg
                print "Now in Main Program"
        if __name__ == "__main__":
            sys.exit(main())
    except getopt.error, msg:
        raise Usage(msg)
```

The string data type

- Python has a built in string data type which allows us to manipulate text
- Python has the ability to handle both ASCII and Unicode string.
- For all the examples we are going to work with we will be using only ASCII strings
- The following example shows some basic string manipulation
#!/usr/bin/python

# declare a string
File = "Pass.0001.exr"

print File
print "The string has %d elements" %(len(File))

# we can treat a string like a list
for i in range(0,len(File)) :
    print File[i]

# we can find the index of a particular element
print File.find(".ex")

# we can split the string based on a character
StringList = File.split(".")

print StringList;

# we can replace elements
File=File.replace("Pass","BeautyPass")

print File

# see if file starts with a particular string
print File.startswith("BeautyPass")

# see if file ends with a particular string
print File.endswith(".ex")

Pass.0001.exr
The string has 13 elements
F
a
s
s
.
0
9
0
.
1
0
1
9

['Pass', '0001', 'exr']
BeautyPass.0001.exr
True

True
Format Specifiers

• In the previous example we used the % format specifier to add to a text string the numeric value for the length.

• This is similar to the C syntax for printing values.

• The table on the next page shows the available specifiers
<table>
<thead>
<tr>
<th>Format String</th>
<th>Meaning</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>Integer Decimal</td>
<td>int,</td>
</tr>
<tr>
<td>%o</td>
<td>Octal Decimal</td>
<td>int</td>
</tr>
<tr>
<td>%x</td>
<td>Hexadecimal</td>
<td>int</td>
</tr>
<tr>
<td>%f</td>
<td>Floating Point (Decimal Notation)</td>
<td>float</td>
</tr>
<tr>
<td>%e</td>
<td>Floating Point (1.E notation)</td>
<td>float</td>
</tr>
<tr>
<td>%c</td>
<td>First Character or argument is printed</td>
<td>char</td>
</tr>
<tr>
<td>%s</td>
<td>Argument is taken to be a string</td>
<td>string</td>
</tr>
<tr>
<td>%r</td>
<td>convert argument to python object</td>
<td>any python type</td>
</tr>
</tbody>
</table>
#!/usr/bin/python

# declare a string
Name="BeautyPass.%04d.exr"
# add the index value
for i in range(0,4) :
    print Name %(i)

Name="Pass"
frame=2
Ext="tiff"
# build a new string from components
FullName ="%s.%04d.%s" %(Name,frame,Ext)
print FullName

BeautyPass.0000.exr
BeautyPass.0001.exr
BeautyPass.0002.exr
BeautyPass.0003.exr
Pass.0002.tiff
Accessing the Filesystem

- The python os module contains a number of functions which allow us to access the file system
- This module allows us to create files and directories
- Change directories
- List the contents of a directory
- and much more besides
```python
#!/usr/bin/python

import os

# get our current directory
CWD = os.getcwd()

print CWD

# make a directory
os.mkdir("TestDir")

# change to the new directory
os.chdir("TestDir")

NewDir = os.getcwd()

print NewDir

print os.listdir(CWD)

# change back to CWD
os.chdir(CWD)

# remove the dir we made
os.rmdir("TestDir")

print os.listdir(CWD)
```
Listing Files in a directory

- The os.listdir() function will return a list of all the files in the current directory.
- If we need to identify only a certain type of file we need search the string for the type we are looking for.
- The following example identifies only .exr files based on the .exr extension.
#!/usr/bin/python

import os

Files=os.listdir(".")

for file in Files :
    if file.endswith(".exr") :
        print file
• The following script uses the previous examples to search for files in the current directory beginning with “name”

• It will then rename the files with the name passed in with the 2nd argument
#!/usr/bin/python

import os
import shutil
import sys

def Usage():
    print "Rename OldName NewName"

def main(argv=None):
    # check to see if we have enough arguments
    if len(sys.argv) != 3:
        Usage()
    else:
        # get the old and new file names
        OldName=sys.argv[1]
        NewName=sys.argv[2]
        # get the files in the current directory
        Files=os.listdir("."")
        # for each file
        for file in Files:
            # see if it starts with the old name
            if file.startswith(OldName):
                # make a copy of the old file name
                oldfile=file;
                # now we break down the string so we can
                # build the new file name
                file=file.split(".")
                file[0]=NewName
                file="%s.%s.%s" %(file[0],file[1],file[2])
                # finally we rename the file (using move)
                shutil.move(oldfile,file)

if __name__ == "__main__":
    sys.exit(main())
The shutil module offers a number of high-level operations on files and collections of files.

As different operating systems use different commands this is a good way of doing operating system independent operations.

This allows us to write scripts which will work on all operating systems.
A More Advanced example

- The following example allows us to reformat files structured like `Name.xxx.ext`
- It has the option to resize the padding `.xxx.` values to any user specified length (default 4)
- To filter file names so only certain files are converted
- To rename the file as part of the conversion
#!/usr/bin/python
from os import *
from os.path import *
import shutil
import os, commands, getopt

def usage():
    print "******
    print 'repad_py re-number_file_sequences'
print "Version 1.0 by macay@bnch.ac.uk"
print "This is only works for files of the format Name.###.ext"
print "The script will process all files if it finds in the current directory."
print "It only certain files are not processed using the '-filter' option."
print "Option -v:"
print "-v will give the output example 00000001."
print "-v filter: only process files containing the text filter."
print "-v rename (newname): rename the file as well"
print "This works only on the whole file name so for example "
print "rename.py -v test Tiff would search for files containing the text Tiff"
print "-h --help print this help"

class Usage(Exception):
    def __init__(self, msg):
        self.msg = msg
    print "Unknown Option"
    usage()

    def main(argv=None):
        if argv is None:
            argv = sys.argv
        # process the command line options
        try:
            try:
                opts, args = getopt.getopt(argv[1:], "hp:f:i:" )
            except getopt.error, msg:
                raise Usage(msg)
            raise Usage, err:
        print >>sys.stderr, err.msg
        print >>sys.stderr, "For help use -h" 
        return 2
        # default file pad is 4 i.e. .0001.
        PAD=4
        # by default try and process all files in the directory
        FILTER=False
        # by default dont rename the file as well
        RENAME=False
        RenameString=[]
        # the string to contain the filter text
        FiltString=[]
for opt, arg in opts:
    if opt in ('-p', '---pad'):
        PAD=int(arg)
    elif opt in ('-h', '--help'):
        usage()
        return
    # find the filter string
    elif opt in ('-f', '--filter'):
        FILTER=True
        FiltString=arg
    elif opt in ('-r', '--rename'):
        RENAME=True
        RenameString=arg

# ok this is cool (try doing it in C++) we create a string containing our format specifier
# i.e. %0d (using PAD as the variable to specify the numeric value)
# this is then used later to pass the value we want for the number of the file
# this works as Python evaluates as it goes [interprets] the string, also note to use a % in the
# string we need to use %% (a la C)
str='%%%0d' % (PAD)

# get all the files in the current directory
FileNames=listdir('.')

# now loop through all the files
ConvCount=0
for Files in FileNames :
    # split the file name into sections
    name=Files.split('.')

    # if filter option has been set see if it is in the string
    if FILTER==True :
        if Files.rfind(FiltString) == -1 :
            continue

    # if we have 3 elements to the filename (not the best check as it could be wrong)
    if len(name) ==3 :
        if RENAME == True :
            name[2]=RenameString

            #build the new file name with the padding
            outputname=name[0]+".*"+str(int(name[1]))+".*name[2]

            # this should be portable over different os's, but basically calls the move command
            shutil.move(Files, outputname)
            ConvCount+=1
            print("Files Converted", ConvCount)
        else:
            print(name[0]+".*"+name[1]+".*name[2])

if __name__=='__main__':
    sys.exit(main())
One of the simplest way of communicating between different packages and different programs is by the use of text files.

Reading and writing files in python is very simple and allows us to very quickly output elements from one software package to another in an easily readable hence debuggable way.
File processing: order of operations
Stream IO

• When a file is opened a file descriptor is returned and this file descriptor is used for each subsequent I/O operation, when any operation is carried out on the file descriptor its is known as a stream.

• When a stream is opened the stream object created is used for all subsequent file operations not the file name.
The open function

- The open function takes two parameters
- The first one is a String for the name of the file to open
- The second one is the open mode “r” for reading from a file “w” for writing to a file

```
1  # open a file for reading
2  FILE=open(FileName,"r")
3  
4  # open a file for writing
5  FILE=open(FileName,"w")
```
The close function

Once a file has been finished with it must be closed.

This is especially important if we are writing to a file as the OS may be storing these values in memory.

The close function actually forces the OS to flush the file to disk and closes thing properly.
Open a file passed on the command line and print the contents

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

import os
import shutil
import sys

def Usage():
    print("ReadFile_[filename]"

def main(argv=None):
    # check to see if we have enough arguments
    if len(sys.argv) != 2:
        Usage()
    else:
        # get the old and new file names
        FileName=sys.argv[1]
        if os.path.exists(FileName):
            FILE=open(FileName,"r")
            lines=FILE.readlines()
            # now we have read the data close the file
            FILE.close()
            LineNum=0
            for line in lines:
                print("%04d\n%s\n"%(LineNum,line),
                    LineNum+=1
            if __name__ == "__main__":
                sys.exit(main())
```
#!/usr/bin/python

import os
import shutil
import sys
# import the uniform function from random
from random import uniform

def Usage():
    print "WriteData[filename]Number"

def main(argv=None):
    # check to see if we have enough arguments
    if len(sys.argv) != 3 :
        Usage()
    else:
        # get the file name to write to
        FileName=sys.argv[1]
        # convert the 2nd argument to an int
        Num=int(sys.argv[2])
        # try to open the file
        try:
            FILE=open(FileName,"w")
            # if this fails catch the error and exit
            except IOError:
                print "Error opening file",FileName
                return
        # loop and create some ranom values to write to the file
        for i in range(0,Num) :
            FILE.write("Point %f %f %f\n" %(uniform(-10,10),uniform(-10,10),uniform(-10,10)) )
        # finally close the file
        FILE.close()
if __name__ == "__main__":
    sys.exit(main())
<table>
<thead>
<tr>
<th>Point</th>
<th>8.040192</th>
<th>-0.405584</th>
<th>8.282515</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>-4.348876</td>
<td>9.117686</td>
<td>3.307612</td>
</tr>
<tr>
<td>Point</td>
<td>0.284490</td>
<td>-8.635971</td>
<td>3.291273</td>
</tr>
<tr>
<td>Point</td>
<td>0.092318</td>
<td>-9.290154</td>
<td>8.649248</td>
</tr>
<tr>
<td>Point</td>
<td>3.125148</td>
<td>-7.677539</td>
<td>-5.233937</td>
</tr>
<tr>
<td>Point</td>
<td>4.029233</td>
<td>-8.312551</td>
<td>-0.478354</td>
</tr>
<tr>
<td>Point</td>
<td>2.601833</td>
<td>8.167995</td>
<td>5.230083</td>
</tr>
<tr>
<td>Point</td>
<td>-6.664861</td>
<td>0.562662</td>
<td>2.441849</td>
</tr>
<tr>
<td>Point</td>
<td>5.003445</td>
<td>-3.522960</td>
<td>-3.876358</td>
</tr>
<tr>
<td>Point</td>
<td>-8.750782</td>
<td>-7.294186</td>
<td>1.573799</td>
</tr>
</tbody>
</table>
• The following example reads the data from the previous program and prints it out.

• As the data is stored on a per line basis we can read it in one hit and then process it
#!/usr/bin/python

import os
import shutil
import sys
# import the uniform function from random
from random import uniform

def Usage():
    print("ReadData", [filename])

def main(argv=None):
    # check to see if we have enough arguments
    if len(sys.argv) != 2:
        Usage()
    else:
        # get the file name to write to
        FileName=sys.argv[1]
        # try to open the file
        try:
            FILE=open(FileName,"r")
        # if this fails catch the error and exit
        except IOError:
            print("Error opening file", FileName)
            return
        # loop and create some ranom values to write to the file
        Lines=FILE.readlines()
        FILE.close()
        for line in Lines:
            # lets see if the line is a point
            if line.startswith("Point") :
                # now split it and convert it to a numeric value
                line=line.split()
                x=float(line[1])
                y=float(line[2])
                z=float(line[3])
                print "%.1f %f %.1f", (x,y,z)

if __name__ == "__main__":
    sys.exit(main())
Object Orientation

• Python is fully object-oriented and supports class inheritance

• Defining a class in Python is simple as with functions, there is no separate interface definition (as used in languages like c++)

• A Python class starts with the reserved word class, followed by the class name.

• Technically, that's all that's required, since a class doesn't need to inherit from any other class.
Python Classes

- Typically a Python class is a self contained .py module with all the code for that module contained within it.

- The class may also have special methods to initialise the data and setup any basic functions

```python
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```
# Class Colour

```python
class Colour:
    'a very simple colour container'

    def __init__(self, r=0.0, g=0.0, b=0.0, a=1.0):
        'constructor to set default values'
        self.r = r
        self.g = g
        self.b = b
        self.a = a

    def debugprint(self):
        'method to print out the colour data for debug'
        print '(%f,%f,%f,%f)' % (self.r, self.g, self.b, self.a)

    def mix(self, colour, t):
        '''method to mix current colour with another by t
        will catch the attribute error and pass back black if
        wrong values are passed'''
        c = Colour()
        try:
            c.r = self.r + (colour.r - self.r) * t
            c.g = self.g + (colour.g - self.g) * t
            c.b = self.b + (colour.b - self.b) * t
            c.a = self.a + (colour.a - self.a) * t
        except AttributeError, e:
            pass
        return c
```
**init**

- Is the python class initialiser, at it’s simplest level it can be thought of as a constructor but it isn’t!

- The instantiation operation (“calling” a class object) creates an empty object.

- The **init** method allows use to set an initial state.

- The actual process is the python constructor is **new**

- Python uses automatic two-phase initialisation - **new** returns a valid but (usually) unpopulated object, which then has **init** called on it automatically.
Class Methods

• The class methods are defined within the same indentation scope of the rest of the class

• There is no function overloading in Python, meaning that you can't have multiple functions with the same name but different arguments

• The last method defined with a name will be used
There are no shorthands in Python for referencing the object's members from its methods the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call.

By convention the first argument of a method is called self.

The name self has absolutely no special meaning to Python.

Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a class browser program might be written that relies upon such a convention.
encapsulation

• In python there is no private or protected encapsulation

• We can access all class attributes using the . operator

• We can also declare instance variables where ever we like in the methods (for example self.foo=10 in a method will be available once that method has been called)

• By convention it would be best to declare all instance variables (attributes) in the __init__ method
Making attributes private

• Whilst python doesn’t support private encapsulation we can fake it using name mangling

• If we declare the class attributes using __ they will be mangled and hidden from the outside of the class

• This is shown in the following example
class ColourPrivate:
    'a very simple colour container'
    def __init__(self, r=0.0, g=0.0, b=0.0, a=1.0):
        'constructor to set default values'
        self.__r = r
        self.__g = g
        self.__b = b
        self.__a = a

    def debugprint(self):
        'method to print out the colour data for debug'
        print '(%f,%f,%f,%f)' % (self.__r, self.__g, self.__b, self.__a)

    def setR(self, r):
        self.__r = r
    def getR(self):
        return self.__r

    def setG(self, g):
        self.__g = g
    def getG(self):
        return self.__g

    def setB(self, b):
        self.__b = b
    def getB(self):
        return self.__b

    def setA(self, a):
        self.__a = a
    def getA(self):
        return self.__a

# /usr/bin/python

from ColourPrivate import *
red = ColourPrivate()
red.__r = 1.0
print red.getR()
red.debugprint()
red.setR(1.0)
print red.getR()
Attribute Access

• We can use the following methods to control what happens when we try to access attributes that don’t exist

```python
class Attr:
    def __init__(self, x=1.0, y=1.0):
        self.x = x
        self.y = y
    def __str__(self):
        '''this method will return our data when doing something like print v'''
        return "[%r,%r]" %(self.x, self.y)
    def __getattr__(self, name):
        print "the attrib '%r' doesn't exist" % (name)
    def __setattr__(self, name, value):
        print "trying to set attribute '%r'=\"%r\"" % (name, value)
        self.__dict__[name] = value
    def __delattr__(self, name):
        print "trying to delete '%r'" % (name)
```

```
a=Attr(1,1)
print a
print a.w
a.w=99
del a.w
```

trying to set attribute 'x'=1
trying to set attribute 'y'=1
[1,1]
the attrib 'w' doesn't exist
None
trying to set attribute 'w'=99
trying to delete 'w'
• __del__ is analogous to the destructor

• It defines behaviour for when an object is garbage collected

• As there is no explicit delete in python it is not always called

• Be careful, however, as there is no guarantee that __del__ will be executed if the object is still alive when the interpreter exits

• __del__ can't serve as a replacement for good coding practice
#!/usr/bin/python

class DelTest:
    def __init__(self):
        'constructor to set default values'
        print "init"

    def __del__(self):
        print "deleted"

>>> from Del import *
>>> d=DelTest()
init
>>> d=1
deleted
>>>
Vec3 Class

- The following examples are going to use the following Vec3 class definition

```python
class Vec3:
    def __init__(self, x=0.0, y=0.0, z=0.0):
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        return "[%f,%f,%f]" % (self.x, self.y, self.z)
```
Comparison Operators

• `__cmp__(self, other)` is the default comparison operator

• It actually implements behavior for all of the comparison operators (<, ==, !=, etc.)

• It is however best to define your own operators using the individual operator overloads as shown in the next code segment
__eq__(self, other)
# equality operator, ==
__ne__(self, other)
# the inequality operator, !=
__lt__(self, other)
# less-than operator, <
__gt__(self, other)
# greater-than operator, >
__le__(self, other)
# less-than-or-equal-to operator, <=
__ge__(self, other)
# greater-than-or-equal-to operator, >=
```python
def __eq__(self, rhs):
    '''equality test'''
    return self.x == rhs.x and self.y == rhs.y and self.z == rhs.z

def __ne__(self, rhs):
    '''not equal test'''
    return self.x != rhs.x or self.y != rhs.y or self.z != rhs.z
```
___str___

- is used with the built in print function, we can just format the string to do what we want.

- There is also a ___repr___ method used to print a human readable presentation of an object.

```python
#!/usr/bin/python
def_vec3(v):
    return [v_1, v_2, v_3]

from Vec3 import *

v1=Vec3(1,2.0,1.0)
print v1
```
Numeric Operators

- The numeric operators are fairly easy, python supports the following operators which take a right hand side argument.

```python
__add__(self, other)
__sub__(self, other)
__mul__(self, other)
__floordiv__(self, other)
__div__(self, other)
__truediv__(self, other)  # python 3
__mod__(self, other)
__divmod__(self, other)
__pow__  # the ** operator
__lshift__(self, other)  #<<
__rshift__(self, other)  #>>
__and__(self, other)  # bitwise &
__or__(self, other)  # bitwise |
__xor__(self, other)  # ^ operator
```
```python
def __add__(self,rhs):
    ''' overloaded + operator for Vec3 = V1+V2'''
    r=Vec3()
    r.x=self.x+rhs.x
    r.y=self.y+rhs.y
    r.z=self.z+rhs.z
    return r

def __sub__(self,rhs):
    ''' overloaded - operator for Vec3 = V1-V2'''
    r=Vec3()
    r.x=self.x-rhs.x
    r.y=self.y-rhs.y
    r.z=self.z-rhs.z
    return r

def __mul__(self,rhs):
    ''' overloaded * scalar operator for Vec3 = V1*S'''
    r=Vec3()
    r.x=self.x*rhs
    r.y=self.y*rhs
    r.z=self.z*rhs
    return r
```
Reflected Operators

- In the previous examples the operators would work like this Vec3 * 2 to make operators that work the other way round we use reflected operators.

- In most cases, the result of a reflected operation is the same as its normal equivalent, so you may just end up defining __radd__ as calling __add__ and so on.
__radd__(self, other)
__rsub__(self, other)
__rmul__(self, other)
__rfloordiv__(self, other)
__rdiv__(self, other)
__rtruediv__(self, other)  # python 3
__rmod__(self, other)
__rdivmod__(self, other)
__rpow__  # the ** operator
__rlshift__(self, other)  #<<
__rrshift__(self, other)  #>>
__rand__(self, other)    # bitwise &
__ror__(self, other)     # bitwise |
__rxor__(self, other)    # ^ operator

def __rmul__(self,lhs) :
    ''' overloaded * scalar operator for Vec3 = V1*S'''
    r=Vec3()
    r.x=self.x*lhs
    r.y=self.y*lhs
    r.z=self.z*lhs
    return r
Augmented Assignment

- These are the += style operators

```python
__iadd__(self, other)
__isub__(self, other)
__imul__(self, other)
__ifloordiv__(self, other)
__idiv__(self, other)
__itruediv__(self, other)  # python 3
__imod__(self, other)
__idivmod__(self, other)
__ipow__  # the ** operator
__ilshift__(self, other)  #<<
__irshift__(self, other)  #>>
__iand__(self, other)    # bitwise &
__ior__(self, other)     # bitwise |
__ixor__(self, other)    # ^ operator
```
def __iadd__(self, rhs):
    '''overloaded +- operator for V1+=V2'''
    self.x += rhs.x
    self.y += rhs.y
    self.z += rhs.z
    return self

def __imul__(self, rhs):
    '''overloaded *= scalar operator for V1*=2'''
    self.x *= rhs
    self.y *= rhs
    self.z *= rhs
    return self
There are quite a few other special class methods that can be used if required:

```python
__unicode__(self)
__format__(self, formatstr)
__hash__(self)
__nonzero__(self)
__dir__(self)
__sizeof__(self)
```
Custom Containers

- There are a number of special class methods that allow the defining of our own containers in python.

- The first thing we need to decide is if we need a mutable or immutable container.

- For an immutable container we only need to define methods for the `len()` and access operators `[]`.

- For mutable we need to be able to set and delete items in the container.

- Finally we can create iterators if we wish as well.
__len__(self)
__getitem__(self, key)
__setitem__(self, key, value)
__delitem__(self, key)
__iter__(self)
__reversed__(self)
__contains__(self, item)
__contains__ (self, item)
__missing__(self, key)
Example

- The following example creates a simple container using a list
- We then overload the correct functions to use it

```python
class MyContainer:
    def __init__(self, data=None):
        if data is None:
            self.data = []
        else:
            self.data = data

    def __str__(self):
        return ','.join(map(str, self.data))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        return self.data[index]

    def __setitem__(self, index, value):
        self.data[index] = value

    def __delitem__(self, index):
        del self.data[index]

    def __iter__(self):
        return iter(self.data)

    def __reversed__(self):
        return MyContainer(reversed(self.data))

    def append(self, value):
        self.data.append(value)
```
#!/usr/bin/python

from MyContainer import *

c=MyContainer([1,2,3,4,5,"string","c"])
print c
print "length of c is",len(c)
c[2]="new value"
print "c[2] is",c[2]
del c[2]
print "deleted item 2",c
print "using the iterator"
for i in c :
    print i

print "using reverse iterator"
for i in reversed(c) :
    print i
c.append(999)
print c

1,2,3,4,5,string,c
length of c is    7
c[2] is new value
deleted item 2   1,2,4,5,string,c
using the iterator
1
2
4
5
string
c
using reverse iterator
c
string
5
4
2
1
1,2,4,5,string,c,999
Composition

• To build more complex classes we can use composition, we just need to import the correct module

```python
class Colour:
    # ctor to assign values
    def __init__(self, r=0, g=0, b=0, a=1):
        self.r=float(r)
        self.g=float(g)
        self.b=float(b)
        self.a=float(a)

    # debug print function to print vector values
    def __str__(self):
        return ' [%f,%f,%f]' % (self.r, self.g, self.b, self.a)

class Point3:
    # ctor to assign values
    def __init__(self, x=0.0, y=0.0, z=0.0):
        self.x=float(x)
        self.y=float(y)
        self.z=float(z)

    # debug print function to print vector values
    def __str__(self):
        return ' [%f,%f,%f]' % (self.x, self.y, self.z)
```
from Point3 import Point3
from Colour import Colour

class Sphere:
    # ctor to assign values
    def __init__(self, pos=Point3(), colour=Colour(), radius=1, name=""):  
        self.pos=pos
        self.colour=colour
        self.radius=radius
        self.name=name

    def Print(self):
        print "Sphere\%s" %(self.name)  
        print "Radius\%d" %(self.radius)  
        print "Colour",
        print self.colour
        print "Position",
        print self.pos
from Sphere import Point3, Colour, Sphere

# Pos, colour, radius, name
s1 = Sphere(Point3(3, 0, 0), Colour(1, 0, 0, 1), 2, "Sphere1")
s1.Print()

p1 = Point3(3, 4, 5)
c1 = Colour(1, 1, 1, 1)
s2 = Sphere(p1, c1, 12, "New")
s2.Print()

s3 = Sphere(Point3(3, 0, 2), Colour(1, 0, 1, 1), 2, "Sphere2")
s3.Print()
Inheritance

- In Python inheritance is generated by passing in the parent class(es) to the child class.
- This will allow all the base class functions to be accessed or override them if defined in the child.
- The first example shows a basic inheritance.
```python
#!/usr/bin/python

class Parent(object):
    
def foo(self):
        print "foo called self=", self
    
def __str__(self):
        return "Parent"

class Child(Parent):
    pass

parent = Parent()
child = Child()

parent.foo()
child.foo()
```

foo called self=Parent
foo called self=Parent
#!/usr/bin/python

class Parent(object):
    def foo(self):
        print "foo\ncalled\nself=%s" % (self)
    def bar(self):
        print "bar\ncalled\nself=%s" % (self)
    def __str__(self):
        return "Parent"

class Child(Parent):
    def foo(self):
        print "foo\ncalled\nself=%s" % (self)
    def __str__(self):
        return "Child"

parent = Parent()
child = Child()

parent.foo()
child.foo()
parent.bar()
child.bar()
#!/usr/bin/python

class Parent(object):
    def __init__(self,a):
        self.a=a

    def foo(self):
        print "foo\n\ncalled\n\nself\n\n=\n\n%"%(self,self.a)

    def __str__(self):
        return "Parent"

class Child(Parent):
    def __init__(self,a,b):
        super(Child,self).__init__(a)
        self.b=b

    def foo(self):
        print "foo\n\ncalled\n\nself\n\n=\n\n%"%(self,self.a,self.b)

    def __str__(self):
        return "Child"

##########################################

parent = Parent(2)
child = Child('test','values')

parent.foo()
child.foo()
References

• [http://vt100.net/docs/tp83/chapter5.html](http://vt100.net/docs/tp83/chapter5.html)
• [http://www.tutorialspoint.com/python/python_variable_types.htm](http://www.tutorialspoint.com/python/python_variable_types.htm)
Further Reading

- http://en.wikipedia.org/wiki/Main_function_(programming)
- http://docs.python.org/library/shutil.html
- http://www.devshed.com/c/a/Python/String-Manipulation/
- http://docs.python.org/library/string.html