Introduction to Python

Jon Macey jmacey@bournemouth.ac.uk

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 stand alone, and if written correctly can run on all operating systems

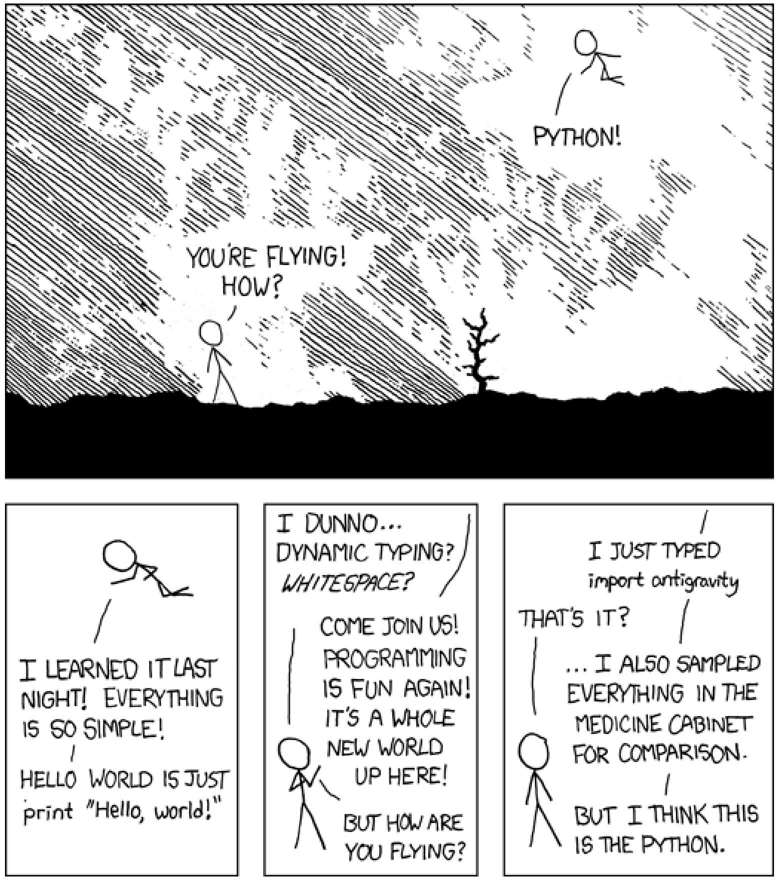
Hello World

```
print 'Hello World!'

```

import this

```python
import this

```

import antigravity


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

- python easter eggs

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

```
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)

numbers

```python
#!/usr/bin/python
a=1
b=2.0
c=35L
d=24+3j

```

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.

Strings

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

str = 'Hello python'

# Prints complete string
print str
```

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

Lists

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

data = [123,"hello",2.45,3+2J]
moreData=["  ","world"]

print data
```

Tuples

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

Tuples

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

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

print data
```

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]
```

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

Python Dictionaries

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

colours={
          "red" : [1,0,0],
          "green" : [0,1,0],
          "blue" : [0,0,1],
```

Type Conversion

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

Function	Description
`int(x ,base)`	Converts x to an integer. base specifies the base if x is a string
`long(x,base)`	Converts x to an long int. base specifies the base if x is a string.
`float(x)`	Converts x to an float.
`complex(real,img)`	Generate a complex number
`str(x)`	Converts x to a string

representation

Type Conversion

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

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

```

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

Membership

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

data = (123,"hello",2.45,3+2J)
numbers=[1,2,3,4,5]
print "world" in data
print "text" not in numbers
```

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

Sequences

```python
import turtle
wn = turtle.Screen()
turtle = turtle.Turtle()

turtle.forward(100)
turtle.left(90)
```

Sequences

```python
#!/usr/bin/python
import turtle
wn = turtle.Screen()
turtle = turtle.Turtle()


```

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

function demo 1

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


def multiReturn(_data) :
    a=_data*1
    b=_data*2
```

function demo 2

```
#!/usr/bin/python


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

```

Selection

- selections allow us to make choices
- most programming languages have 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

Selection

```python
import turtle
wn = turtle.Screen()
turtle = turtle.Turtle()

type = "Triangle"

```

Python Comparison Operators

given `a=10 b=20`

Operators	Description	Example
==	equality operator returns true if values are the same	(a==b) is not true
`!=`	not equal operator	(a!=b) is true
<> (now obsolescent)	Checks if the value of two operands are equal or not	(a<>b) is true
>	Checks if the value of left operand is greater than the value of right operand	(a>b) is not true
<	Checks if the value of left operand is less than the value of right operand	(a>b) is true
>=	Checks if the value of left operand is greater than or equal to the value of right operand	(a>=b) is not true
<=	Checks if the value of left operand is less than or equal to the value of right operand	(a<=) is true

Python Logical Operators

given `a=10 b=20`

Operators	Description	Example
`and`	Logical and	a and b is true
`or`	Logical or	a or b is true
`not` (now obsolescent)	Logical not	not (a and b) is false

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

iteration

```python
import turtle
wn = turtle.Screen()
turtle = turtle.Turtle()

def Square(_size) :
    turtle.forward( size)
```

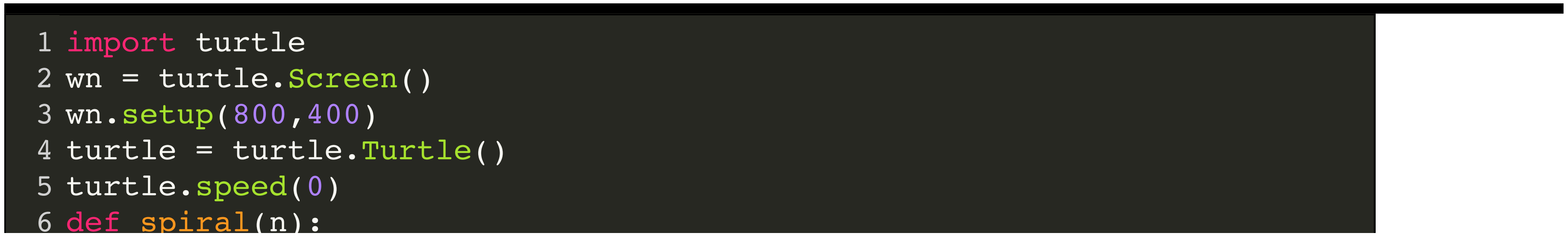
iteration

```python
import turtle
wn = turtle.Screen()
turtle = turtle.Turtle()

#code taken from http://docs.python.org/dev/library/turtle.html

```

Recursion

- Recursion occurs when a thing is defined in terms of itself or of its type
- in programming this usually done by defining a function and call the same function within itself
- obviously we will need some way of escaping this else it will go on forever
- We use this quite a lot in graphics to traverse hierarchies.

Recursion



```python
import turtle
wn = turtle.Screen()
wn.setup(800,400)
turtle = turtle.Turtle()
turtle.speed(0)
def spiral(n):
```

looping for x and 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

```
for(float y=-10.0f; y<10.0f; ++y)
{
  for(float x=-10.0f; x<10.0f; ++x)
  {
    std::cout<<x<<' '<<y<<'\n';
  }
}
```

looping for x and y

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

y=-10.0

while y<=10.0 :
    x=-10.0
```

A 'pythonic' 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

2. Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
`abs()`	`divmod()`	`input()`	`open()`	`staticmethod()`

enumerate

```
#!/usr/bin/python

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

c=list(enumerate(colours))
print c
```

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

set / frozenset

this doesn't work fully on this system best to run in the shell

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

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

lambda

```python
#!/usr/bin/python
import math
a=[1,2,3,4,5]
b=map(lambda x: x+1 , a)
print b

```

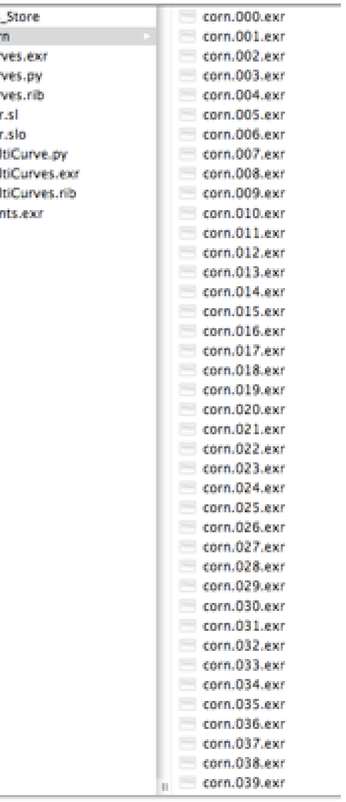
Programming

- 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
- In some cases this method is quicker than using the GUI (but does require some additional knowledge)

Example



```
_Store          corn.000.exr
m               corn.001.exr
ves.exr         corn.002.exr
ves.py          corn.003.exr
ves.rib         corn.004.exr
r.sl            corn.005.exr
r.slo           corn.006.exr
ltiCurve.py     corn.007.exr
ltiCurves.exr   corn.008.exr
ltiCurves.rib   corn.009.exr
nts.exr         corn.010.exr
                corn.011.exr
                corn.012.exr
                corn.013.exr
                corn.014.exr
                corn.015.exr
                corn.016.exr
                corn.017.exr
                corn.018.exr
                corn.019.exr
                corn.020.exr
                corn.021.exr
                corn.022.exr
                corn.023.exr
                corn.024.exr
                corn.025.exr
                corn.026.exr
                corn.027.exr
                corn.028.exr
                corn.029.exr
                corn.030.exr
                corn.031.exr
                corn.032.exr
                corn.033.exr
                corn.034.exr
                corn.035.exr
                corn.036.exr
                corn.037.exr
                corn.038.exr
                corn.039.exr
```

- If we wish to rename every file
opposite in a GUI we would hav
file and type the new name

- Some Operating Systems allo
of GUI tasks but this is still time

- The answer in most cases is to
program or to write a script

- Most scripting languages let u
underlying os commands to do

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

Shell Commands

Command's Purpose	MS-DOS	Linux	Basic Linux Example
Copies files	copy	cp	cp thisfile.txt /home/thisdirectory
Moves files	move	mv	mv thisfile.txt /home/thisdirectory
List files	dir	ls	ls
Clears screen	cls	cls	clear
Deletes files	del	rm	rm thisFile.txt
Finds a string of text in a file	find	grep	grep ImageName *.txt
Creates a directory	mkdir	mkdir	mkdir images

Shell Commands

Command's Purpose	MS-DOS	Linux	Basic Linux Example
View a file (in shell)	more	less	more text.txt (can use less as well)
Renames a file	ren	mv	mv this.txt that.txt
Displays your location in the file system	chdir	pwd	pwd
Changes directories with a specified path (absolute path)	cd pathname	cd pathname	cd /directory/directory
Changes directories with a relative path	cd ..	cd ..	cd ../images/

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

```
echo $HOME
echo %HOMEPATH%

/Users/jmacey
\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 se 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

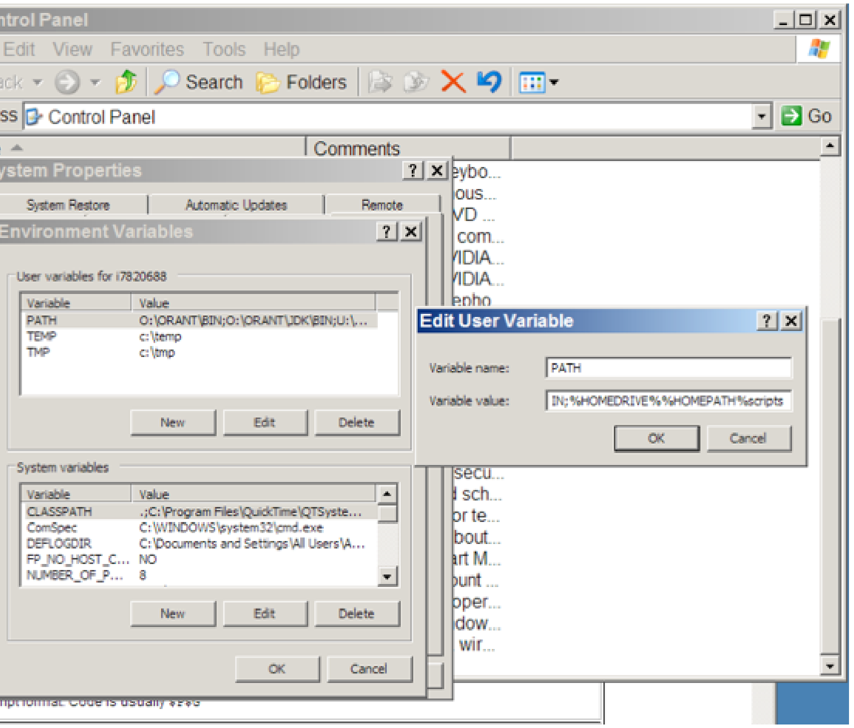
```
export PATH=$PATH:$HOME/scripts
```

- if you re-open the shell this will be made permanent
- Now any program placed in this directory may be found and executed

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

Windows Environment Variables



User variables for i7820688

Variable	Value
PATH	O:\ORANT\BIN;O:\ORANT\JDK\BIN;U:\...
TEMP	c:\temp
TMP	c:\tmp

Variable name: PATH

Variable value: IN;%HOMEDRIVE%%HOMEPATH%scripts

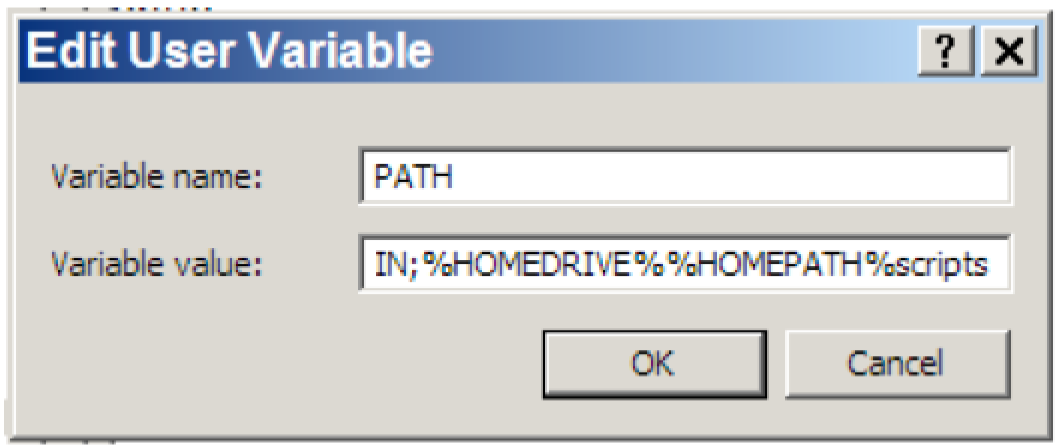
System variables

Variable	Value
CLASSPATH	.;C:\Program Files\QuickTime\QTSyste...
ComSpec	C:\WINDOWS\system32\cmd.exe
DEFLOGDIR	C:\Documents and Settings\All Users\A...
FP_NO_HOST_C...	NO
NUMBER_OF_P...	8

- Select the system
called Path

- Click on the edit b
the following dialo
displayed

Windows Environment Variables



Edit User Variable

Variable name: PATH

Variable value: IN;%HOMEDRIVE%%HOMEPATH%scripts

- At the end of the Variable value line add the following

```
;%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
cd %HOMEPATH%
mkdir scripts

// linux
cd
mkdir scripts
```

testing

```
#!/usr/bin/python

print 'this is working'
```

- 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 777 hello.py
```

- now from any directory you should be able to type hello.py to run the script

os.environment

- These need to be run on a machine and not online

```
#!/usr/bin/python
import os

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

```
#!/usr/bin/python
import os

if ( os.environ.get("PROJECTDIR") == "/tmp") :
    print "project ok"
else :
    print "project not ok"
```

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

Python main

```
#!/usr/bin/python
import sys

def foo(argv=None):
    print "in foo function"
    print "my name is ",__name__



if __name__ == "__main__":
    sys.exit(foo())
```

```
#!/usr/bin/python
import sys
import foo

def main(argv=None):
    print "in main function"
    print __name__
    foo.foo()


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

Command Line arguments

- When a program is executed form 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

arguments

```python
#!/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())
```

```
./arg.py hello this is a command line -f -a 1 2 3
hello
this
is
a
command
line
-f
-a
1
2
3
```

getopt

- 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 (know 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)

getopt

```
#!/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\n\n"
        usage()
```

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

demo

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

# declare a string

File = "Pass.0001.exr"

```

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

Format Specifiers

Format String	Meaning	Data Type
`%d`	Integer Decimal	int
`%o`	Octal Decimal	int
`%x`	Hexadecimal	int
`%f`	Floating Point (Decimal Notation)	float
`%e`	Floating Point (1.E notation)	float
`%c`	First Character or argument is printed	char
`%s`	Argument is taken to be a string	string
`%r`	convert argument to python object	any python type

format

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

# declare a string

Name="BeautyPass.%04d.exr"

```

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

Accessing the filesystem

```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
```

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

os.listdir()

```
#!/usr/bin/python

import os

Files=os.listdir(".")

for file in Files :
    if file.endswith(".exr") :
        print file
```

Rename.py

- 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

Rename.py

```
#!/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]
```

shutil

- 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

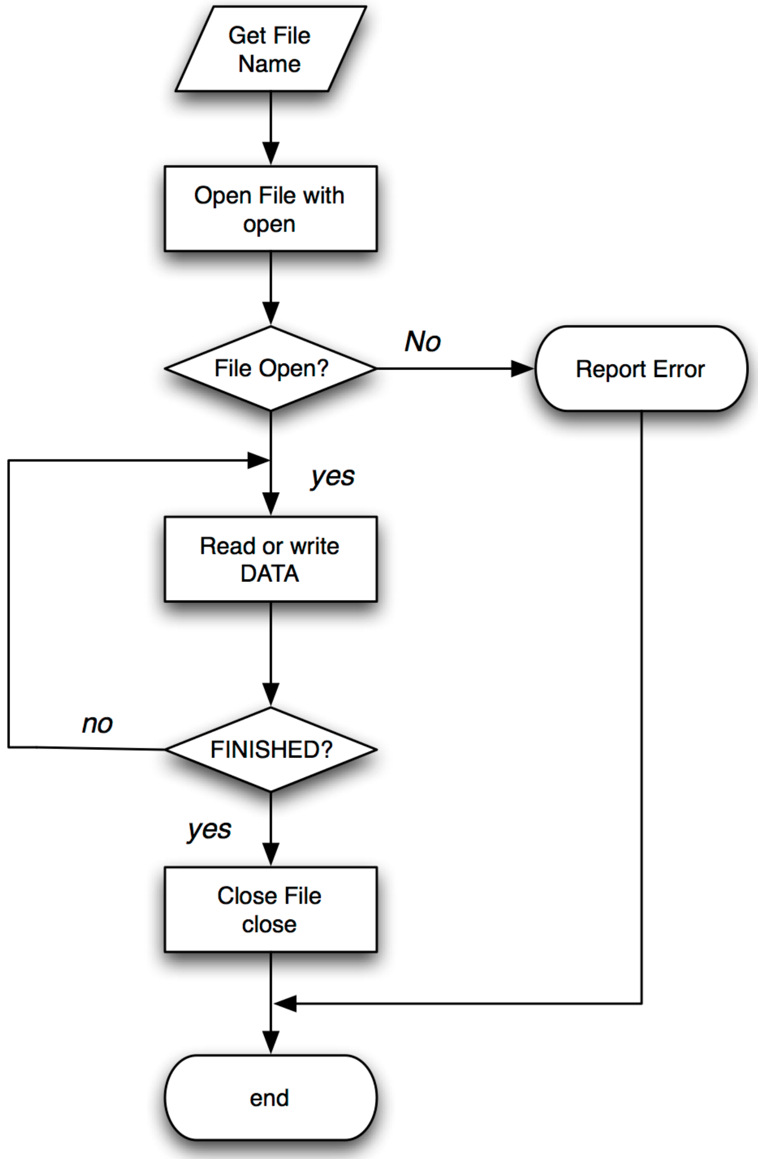
repad.py

```
#!/usr/bin/python
from os import *
from os.path import *
import shutil

import os, commands, getopt, sys

def usage() :
    print "**********************************************************
    print "repad.py re-number file sequences"
    print "Version 1.0 by jmacey@bmth.ac.uk"
    print "**********************************************************
    print "At present it only works for files of the format Name.###.
    print "The script will process all files it finds in the current
    print "If only certain files are to be processed use the -f Filte
    print "\nOptions :\n"
```

Files

- 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.

Files


Get File Name
Open File with open
File Open?
No
Report Error
yes
Read or write DATA
no
FINISHED?
yes
Close File close
end

Stream I/O

- 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

```
# open a file for reading
FILE=open('test.txt','r')

# open a file for writing
FILE=open('text.txt',''w')
```

- The open function takes two parameters
  - The fist is a String for the name of the file to open
  - The 2nd is the open mode 'r' for reading from a file 'w' for writing to a file

The close method

```
FILE.close()
```

- 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

Example

```
#!/usr/bin/python

import os
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)) :
```

write file

```python
#!/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 :
```

output

```
Point -8.079503 -5.887453 0.477799
Point -8.509921 -1.826855 6.271168
Point 5.899356 9.357611 6.468166
Point 8.883614 -7.286649 -7.365122
Point -6.063683 -4.825969 -3.024902
Point -0.119126 5.620598 5.814827
Point 8.060026 2.640244 -4.197079
Point 8.952118 1.571210 8.069305
Point -9.708913 5.454307 2.763587
Point -2.809199 -5.292178 3.994426
Point 2.788986 4.434073 8.763425
```

Reading the data

- 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

ReadData

```python
#!/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 :
```

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

```
class ClassName :
  <statement 1>
  .
  .
  .
  <statement N>
```

A Colour Class

```
#!/usr/bin/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)
```

Colour Test

```
#!/usr/bin/python

from Colour import *


red=Colour()
red.r=1.0
red.debugprint()
```

```
./ColourTest.py
[1.000000,0.000000,0.000000,1.000000]
```

`__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.

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

self

- 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

Colour Private

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

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) :
```

Private Test

```
#!/usr/bin/python

from ColourPrivate import *


red=ColourPrivate()
red.__r=1.0
print red.getR()
red.debugprint()
red.setR(1.0)
print red.getR()
```

```
./ColourPTest.py
0.0
[0.000000,0.000000,0.000000,1.000000]
1.0
```

Attribute Access

```
#!/usr/bin/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 lik
        return "[%r,%r]" %(self.x,self.y)

    def __getattr__(self,name) :
        print "the attrib %r doesn't exist" %(name)
```

```
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
99
trying to delete 'w'
```

`__del__`

- `__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

del test

```
#!/usr/bin/python

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

    def __del__(self) :
        print "deleted"
```

```
python
>>> from del import *
>>> d=DelTest()
init
>>> d=1
deleted
>>>
```

vec3 class

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

```
class Vec3 :
    ''' a simple Vec3 class for basic 3D calculations etc'''
    def __init__(self,x=0.0,y=0.0,z=0.0) :
        self.x=x
        self.y=y
        self.z=z


    def __str__(self) :
        ''' this method will return our data when doing something lik
        return "[%f,%f,%f]" %(self.x,self.y,self.z)

    def __eq__(self,rhs) :
        ''' equality test'''
        return self.x == rhs.x and self.y == rhs.y and self.z == rhs.
```

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

Comparison Operators

```
# equality operator ==
__eq__(self,rhs)
# inequality operator !=
__ne__(self,rhs)
# less than operator <
__lt__(self,rhs)
# greater than operator >=
__gt__(self,rhs)
# less or equal than operator <=
__le__(self,rhs)
# greater than or equal operator >=
__ge__(self,rhs)
```

`__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.

Numeric Operators

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

```
__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
```

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.

Reflected Operators

```
__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
```

Augmented Assignment

- These are the += style operators

```
__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
```

Class Representation

- There are quite a few other special class methods that can be used if required

```
__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.

Custom Containers

```
__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

```
class MyContainer :
    ''' a very simple container class '''
    def __init__(self,data=None) :
        if data is None :
            self.data=[]
        else :
            self.data=data

    def __str__(self) :
        ''' method to print out the container contents'''
        return ','.join(map(str, self.data))

    def __len__(self) :
        ''' return the length of the data'''
        return len(self.data)
```

Test

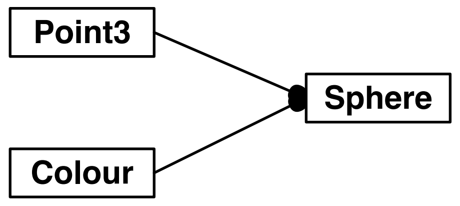
```
  #!/usr/bin/python

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
```

output

```
1,2,3,4,5,string,c lengthofcis 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
```

Composition and Aggregation

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



Colour.py

```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,%f]' %(self.r,self.g,self.b,self.a)
```

Point3.py

```python
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,%d]' %(self.x,self.y,self.z)
```

Sphere.py

```
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)
```

Test

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

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")
```

```
./SphereTest.py
Sphere Sphere1
Radius 2
Colour [1.000000,0.000000,0.000000,1.000000]
Position  [3.000000,0.000000,0]
Sphere New
Radius 12
Colour [1.000000,1.000000,1.000000,1.000000]
```

```
Position  [3.000000,4.000000,5]

Sphere Sphere2
Radius 2
Colour [1.000000,0.000000,1.000000,1.000000]
Position  [3.000000,0.000000,2]
```

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

example

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

class Parent(object):

    def foo(self):
        print "foo called self=%s" %(self)
```

over ride

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

class Parent(object):

    def foo(self):
        print "foo called self=%s" %(self)
```

over ride constructor

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

class Parent(object):

    def __init__(self,a) :
        self.a=a
```

References

- http://vt100.net/docs/tp83/chapter5.html
- http://www.artima.com/weblogs/viewpost.jsp?thread=4829
- http://www.tutorialspoint.com/python/python_variable_types.ht

References

- http://en.wikipedia.org/wiki/Environment_variable
- 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
- http://www.rafekettler.com/magicmethods.html