Introduction to Python

Jon Macey
jmacey@bournemouth.ac.uk
http://nccastaff.bournemouth.ac.uk/jmacey/

Python

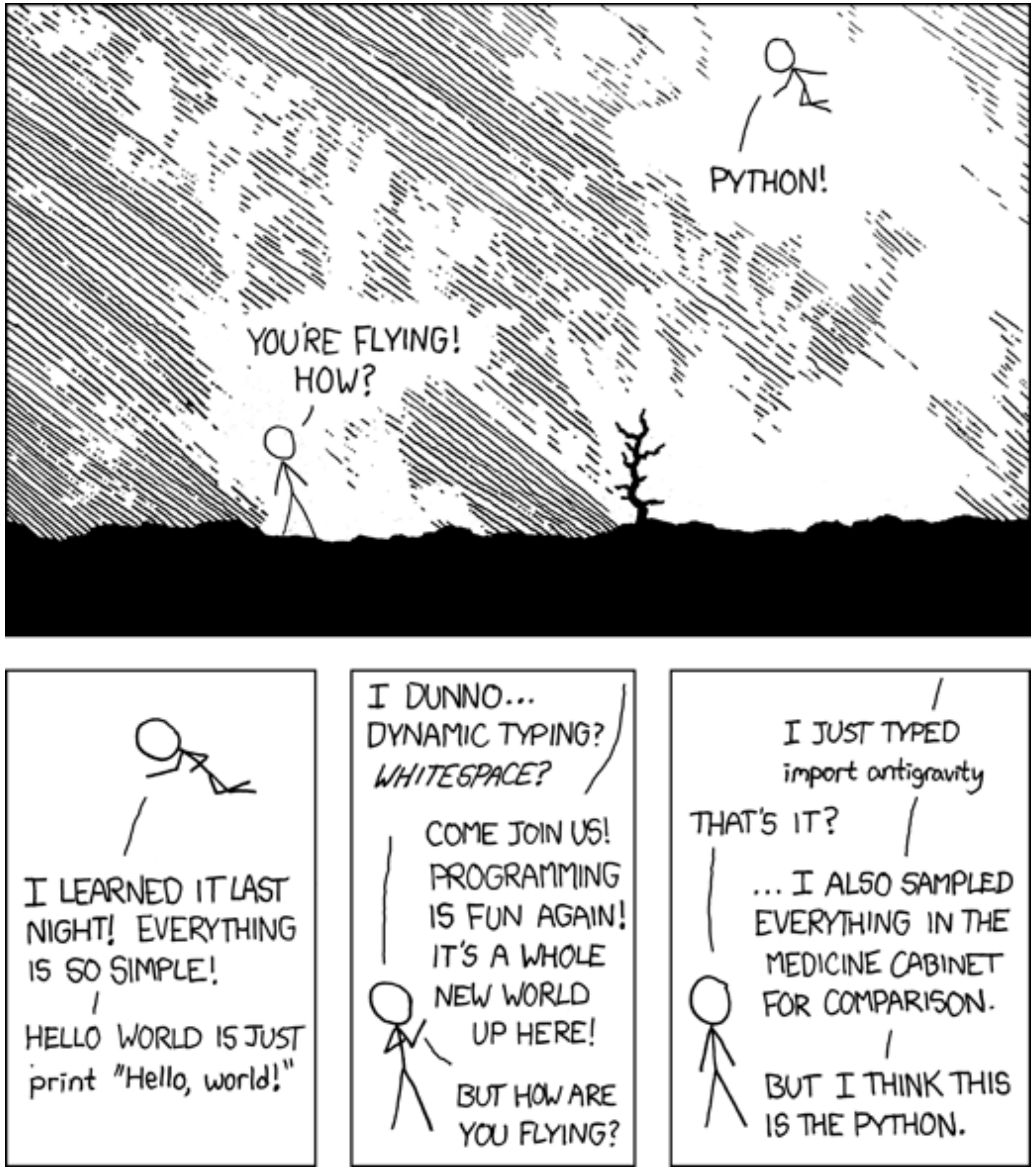
- python is a very flexible programming language, it can be used in a number of different ways.
- We can also write complex programs which run standalone, and if written correctly can run on all operating systems

import this

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
- External packages and ideas

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

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.py

```
#!/usr/bin/python

a=1
b=2.0
c=35L
d=24+3j

print type(a)
print type(b)
print type(c)
print type(d)
```

```
<type 'int'>
<type 'float'>
<type 'long'>
<type 'complex'>
```

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 len-1 at the end
- The plus (+) sign is the string concatenation operator, and the asterisk (*) is the repetition operator.

strings.py

```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"
```

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

String1.py

```python
#!/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(".exr")
```

```
Pass.0001.exr
The string has 13 elements
P
a
s
s
.
0
0
0
1
.
e
x
r
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

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

FormatString.py

```python
#!/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
```

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 sting data is accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to len-1.
- The + operator concatenates and the * duplicates

list.py

```
#!/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

tuple.py

```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+=moreData
```

```
./tuple.py
(123, 'hello', 2.4500000000000002, (3+2j))
hello
(2.4500000000000002, (3+2j))
hello world
Traceback (most recent call last):
  File "./tuple.py", line 13, in <module>
    data+="more"
TypeError: can only concatenate tuple (not "str") to tuple
```

more on slice operators

slice.py

```
#!/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]␣",a[:-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

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

Dictionary={
            "red":[1.0,0.0,0.0],
            "green":[0.0,1.0,0.0],
            "blue":[0.0,0.0,1.0],
            "white":[1.0,1.0,1.0],
            "black":[0.0,0.0,0.0]
            }

print Dictionary.get("red")
print Dictionary.get("white")
print Dictionary.get("purple")
```

Create a dictionary of colour lists
“key”:[r,g,b]

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

```
[1.0, 0.0, 0.0]
[1.0, 1.0, 1.0]
None
```

note “None” returned if “key” not found

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

convert.py

```python
#!/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

membership.py

```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
print 99 in numbers
print 2 in numbers
```

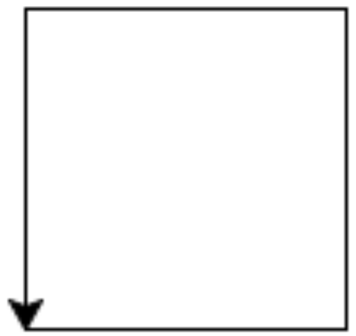
```
False
True
False
True
```

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.

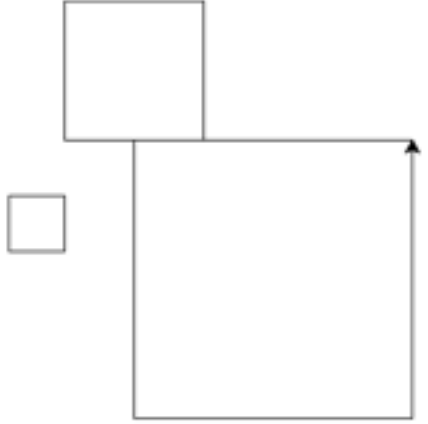
Sequences

- 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



sequence.py

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

from turtle import *

forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
done()
```

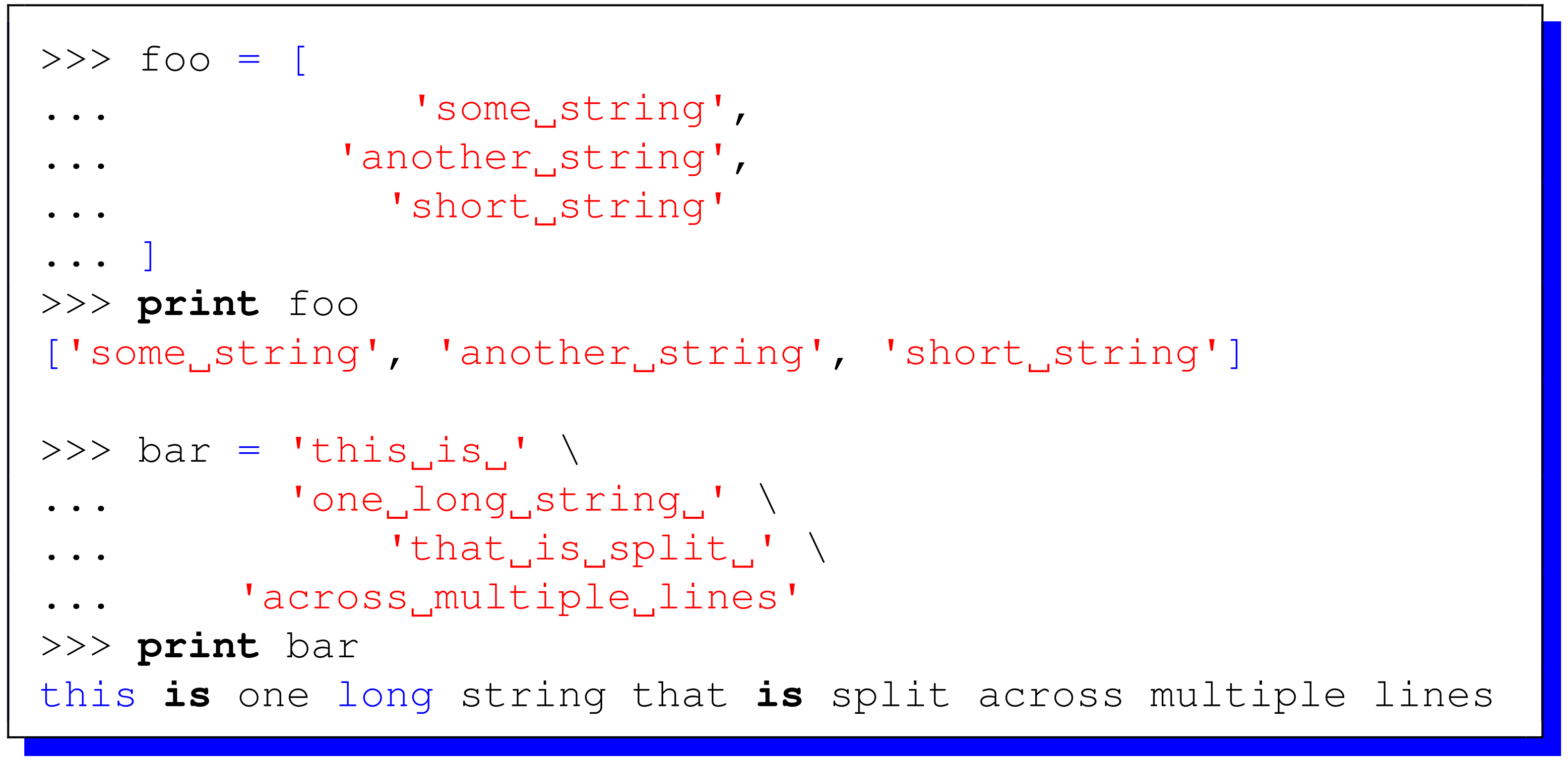


sequence2.py

```python
#!/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()
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



```
>>> 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

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
```

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

```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)
```

```
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"
```

all code after the :
is indented with a tab and
executed
if statement True

elif : is an else if

else is a “catch all”
if the others fail

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
!=	Checks if the value of two operands are equal or not	(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

a=10 and b=20

Operator	Description	Example
and	Logical and	a and b is true
or	Logical or	a or b is true
not	Logical not	not (a and b) is false

selection

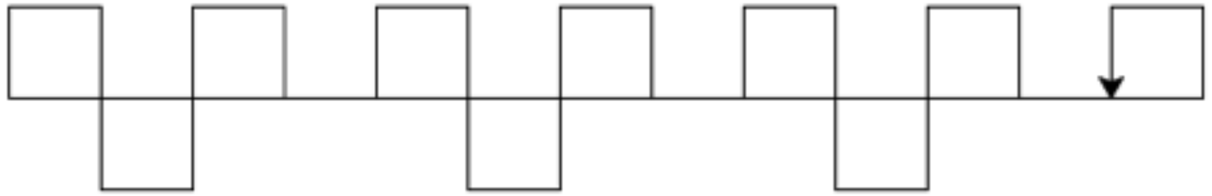
- selections can be embedded to create quite complex hierarchies of “questions”
- This can sometimes make reading code and maintenance hard especially with the python white space rules as code quite quickly becomes complex to read
- We usually prefer to put complex sequences in functions to make the 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

```
#!/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()
```

iteration.py

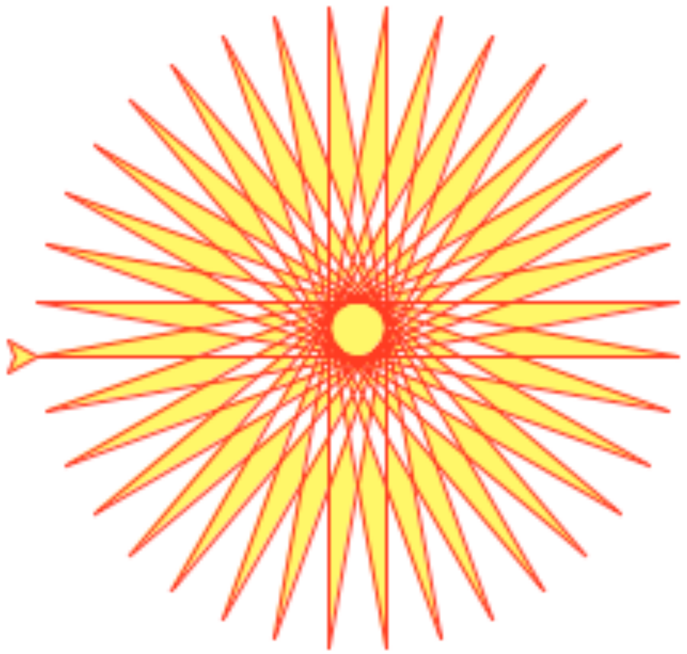


the range function
produces a list
for x in ... assigns
each list element
to x in turn

```
range(-250,250,40)
[-250, -210, -170, -130, -90, -50, -10, 30, 70, 110, 150, 190, 230]
```

iteration2.py

```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

```
for (float y=-10; y<=10.0; y+=0.5)
{
  for(float x=-10; x<=10.0; x+=0.5)
  {
    std::cout<<x<<"␣"<<y<<"\n";
  }
}
```

```
#!/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
```

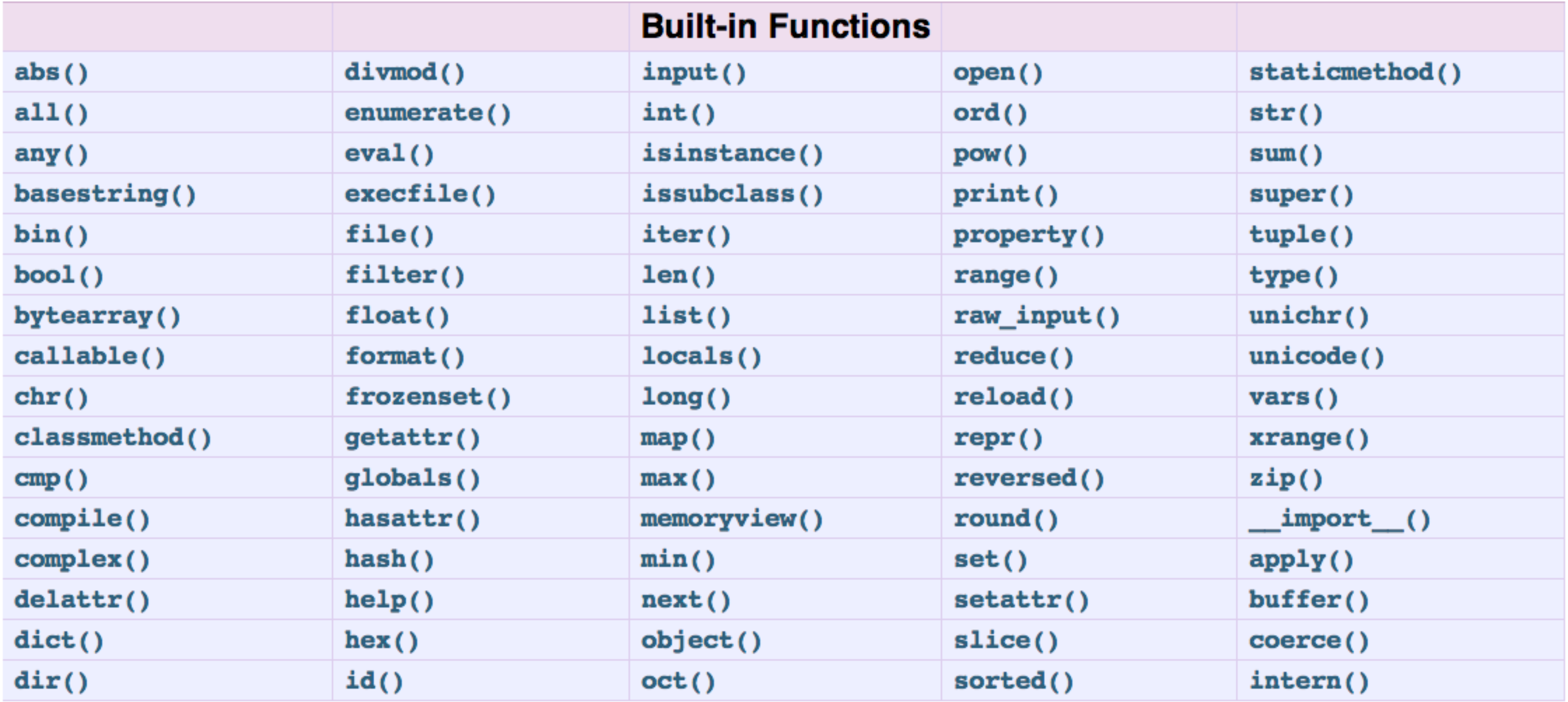
alternative loop

Loop2.py

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

```
(0, 0)
(0, 1)
(0, 2)
(0, 3)
(0, 4)
(1, 0)
(1, 1)
(1, 2)
(1, 3)
(1, 4)
(2, 0)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
(3, 4)
(4, 0)
(4, 1)
(4, 2)
(4, 3)
(4, 4)
```

Built in functions



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

enumerate

enumerate.py

```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

```python
#!/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␣a␣|␣b",a | b
print "intersection␣a␣&␣b" ,a & b
print "subset␣false␣a<b",a < b
print "difference␣a-b",a - b
print "Symmetric␣diff␣a^b",a ^ b
```

```
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
set([0, 1, 2, 3, 4])
a= set([1, 2, 3, 4])
b= set([3, 4, 5, 6])
union a | b set([1, 2, 3, 4, 5, 6])
intersection a & b set([3, 4])
subset false a<b False
difference a-b set([1, 2])
Symmetric diff a^b set([1, 2, 5, 6])
```

lambda

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

To=[0,0,0]
From=[0,8,4]
print To,From

direction = map(lambda x,y : x-y , To,From)
print direction
# get the length
len = math.sqrt(sum(map(lambda x : x*x , direction )))
print len
# divide by length
normal= map(lambda x : x/len , direction)
print normal
```

```
./lambda.py
[2, 3, 4, 5, 6]
[0, 0, 0] [0, 8, 4]
[0, -8, -4]
8.94427191
[0.0, -0.894427190999916, -0.447213595499958]
```

This example shows the inherent instability of floating point calculations

Modules

- A module is a file containing Python definitions and statements
- The file name is the module name with the suffix .py appended
- Within a module, the module's name (as a string) is available as the value of the global variable __name__
- The following example shows this (from the python documentation)

```python
# Fibonacci numbers module

def fib(n):
  a, b = 0, 1
  while b < n:
    print b,
    a, b = b, a+b

def fib2(n):
  result = []
  a, b = 0, 1
  while b < n:
    result.append(b)
    a, b = b, a+b
  return result
```

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Using Local Names

- If we are going to use a module / function often we can give it a local name.
- For example

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Modules

- A module can contain executable statements as well as function definitions. These statements are intended to initialize the module.
- Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module.
- Modules can import other modules.
    - **Note** For efficiency reasons, each module is only imported once per interpreter session.
    - Therefore, if you change your modules, you must restart the interpreter or, if it's just one module you want to test interactively, use reload(), e.g. reload(modulename).

import

- There is a variant of the import statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

- This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).
- There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Executing modules as scripts

- When you run a Python module with   python fibo.py <arguments>
- the code in the module will be executed, just as if you imported it, but with the __name__ set to "__main__".
- This means we can modify the script to execute stand alone by adding the following code

```
if __name__ == "__main__":
  import sys
  if len(sys.argv) > 1 :
    fib(int(sys.argv[1]))
```

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

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

def main(argv=None):
  print "in␣main␣function"


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

The Module Search Path

- When a module is imported (or attempted to be) the following paths are searched
    - the directory containing the input script (or the current directory).
    - PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
    - the installation-dependent default.
- After initialisation, Python programs can modify sys.path.
- The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path.

adding a path

```
import sys
sys.path.append('~/MyPythonModules')
```

dir()

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

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__egginsert', '__excepthook__', '__name__', '
    __package__', '__plen', '__stderr__', '__stdin__', '__stdout__', '
    _clear_type_cache', '_current_frames', '_getframe', '_mercurial', '
    api_version', 'argv', 'builtin_module_names', 'byteorder', 'call_tracing', '
    callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear', '
    exc_info', 'exc_type', 'excepthook', 'exec_prefix', 'executable', 'exit', '
    flags', 'float_info', 'float_repr_style', 'getcheckinterval', '
    getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding', 'getprofile',
     'getrecursionlimit', 'getrefcount', 'getsizeof', 'gettrace', 'hexversion', '
    long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path'
    , 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', '
    py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile', '
    setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion', '
    version', 'version_info', 'warnoptions']
>>>
```

help([object])

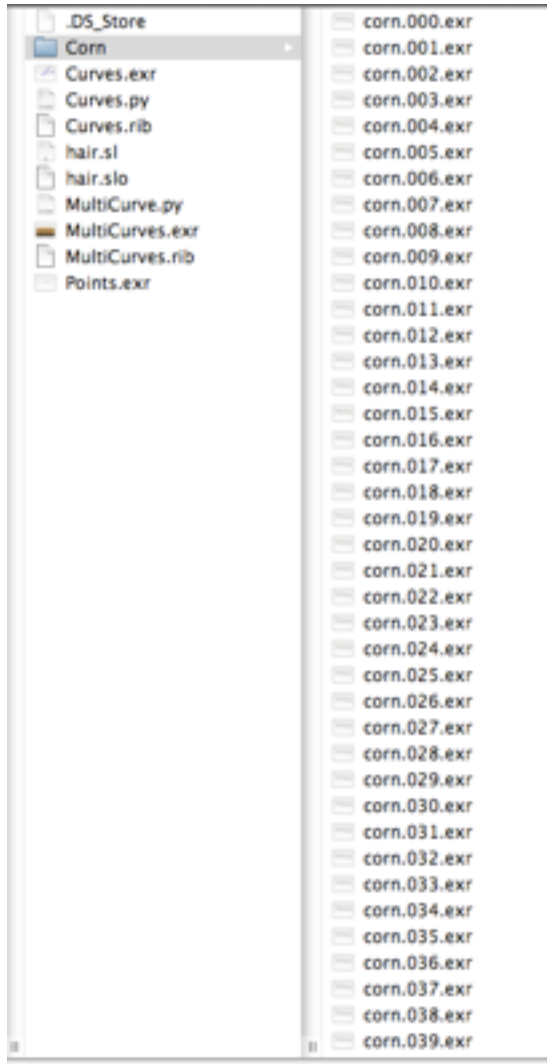
- get python help on object
- object must be imported, use q to exit and usual man page paging system for control.

```
help(min)
Help on built-in function min in module __builtin__:

min(...)
    min(iterable[, key=func]) -> value
    min(a, b, c, ...[, key=func]) -> value

    With a single iterable argument, return its smallest item.
    With two or more arguments, return the smallest argument.
```

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



.DS_Store
Corn
Curves.exr
Curves.py
Curves.rib
hair.sl
hair.slo
MultiCurve.py
MultiCurves.exr
MultiCurves.rib
Points.exr

corn.000.exr
corn.001.exr
corn.002.exr
corn.003.exr
corn.004.exr
corn.005.exr
corn.006.exr
corn.007.exr
corn.008.exr
corn.009.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 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

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
Lists files	dir	ls	ls
Clears screen	cls	clear	clear
Deletes files	del	rm	rm thisfile.txt
Finds a string of text in a file	find	grep	grep this word or phrase thisfile.txt
Creates a directory	mkdir	mkdir	mkdir directory
View a file	more	less[d]	less thisfile.txt
Renames a file	ren	mv	mv thisfile.txt thatfile.txt[e]
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 ..

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

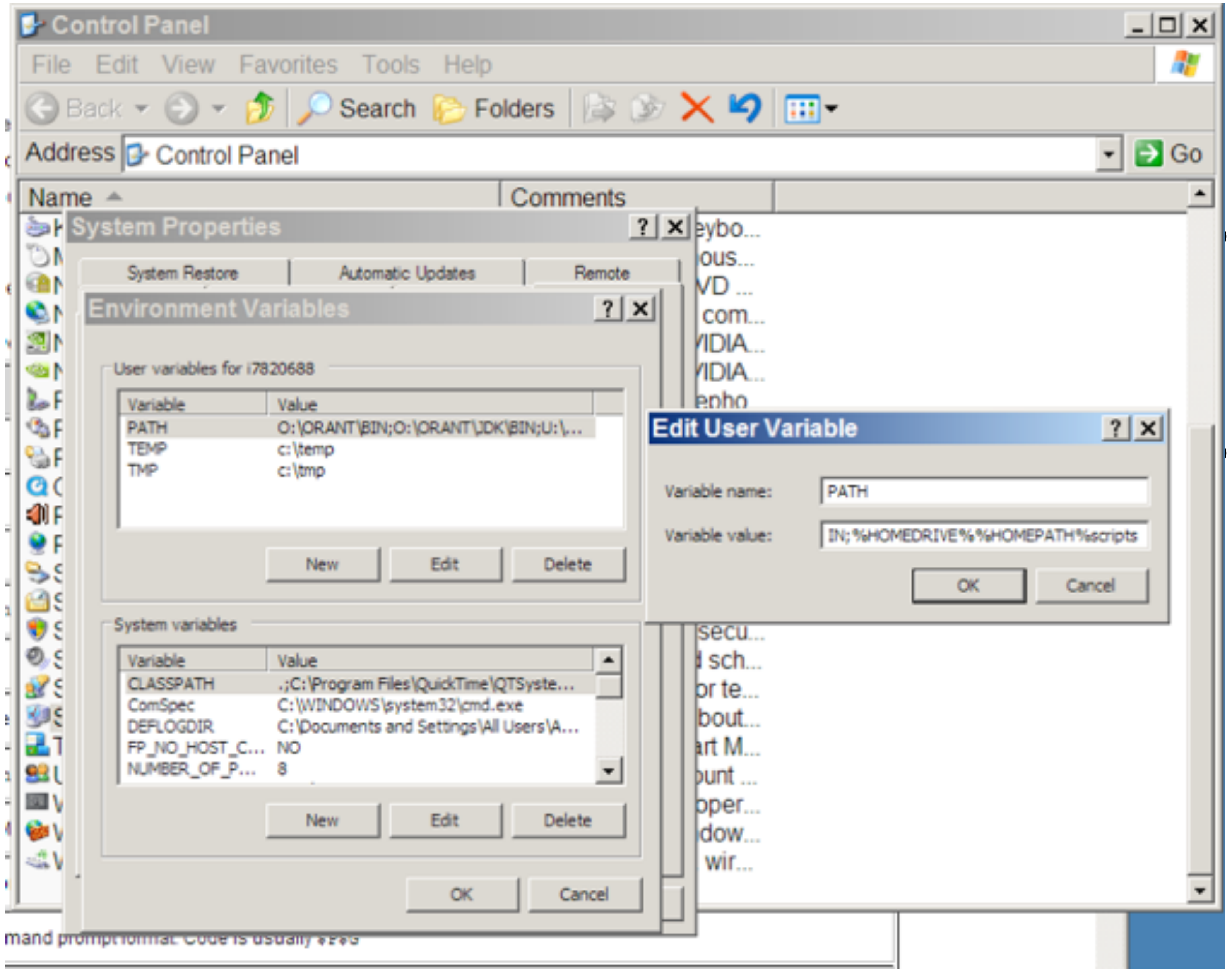
- The default shell used in the linux studios is the bash shell (Bourne again Shell)
- To set environment variable in this shell we use a file called .bashrc which is hidden in the home directory
- if you type gedit ~/.bashrc you can access it

```
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



User variables for i7820688

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

System variables

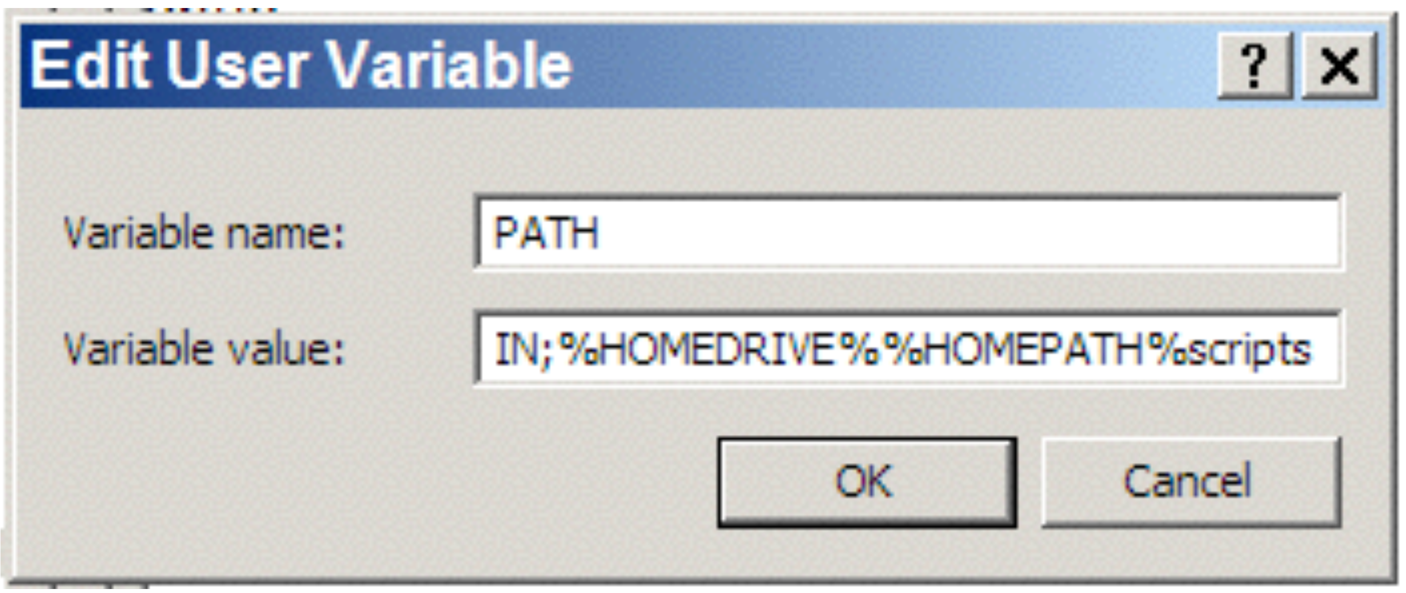
Variable	Value
CLASSPATH	.;C:\Program Files\QuickTime\QTSyste...
ComSpec	C:\WINDOWS\system32\cmd.exe
DEFLOGDIR	C:\Documents and Settings\All Users\A...
FP_NO_HOST_C...	NO
NUMBER_OF_P...	8

Variable name: PATH

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

Select the system variable called Path

Click on the edit button and the following dialog will be displayed



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

// Unix
cd
mkdir scripts
```

testing

```
#!/bin/python
print "This␣is␣working"
```

- Type the above in an editor (or your choice) and save it in the scripts directory a 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

os.environment

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

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

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

print os.environ.get("PATH")
```

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)
```

```
/Users/jmacey/teaching/Python/PythonLectures/Code/Lecture2
/Users/jmacey/teaching/Python/PythonLectures/Code/Lecture2/TestDir
['FormatString.py', 'OS.py', 'String1.py', 'TestDir']
['FormatString.py', 'OS.py', 'String1.py']
```

Listing Files in a directory

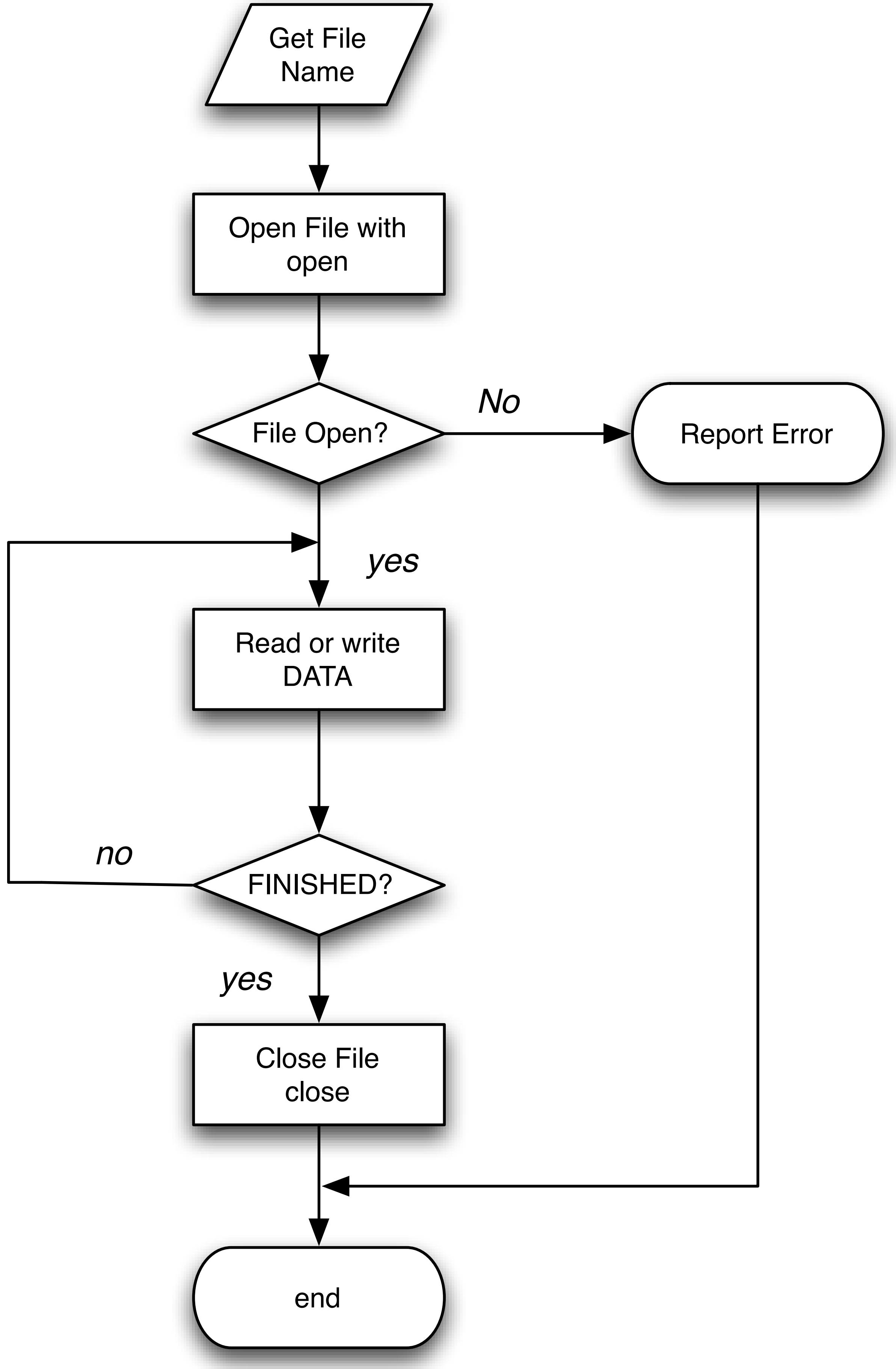
- The os.listdir() function will return a list of all the files in the current directory
- If we need to identify only a certain type of file we need search the string for the type we are looking for
- The following example identifies only exr files based on the .exr extension

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

Files=os.listdir(".")

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

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.

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

```
# open a file for reading
FILE=open(FileName,"r")

# open a file for writing
FILE=open(FileName,"w")
```

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

The close function

```
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

Open a file passed on the command line and print the contents

```
#!/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␣%s" %(LineNum,line),
        LineNum+=1

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

```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 :
    # 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())
```

```
Point 8.040192 -0.405584 8.282515
Point -4.348876 9.117686 3.307612
Point 0.284490 -8.635971 3.291273
Point 0.092318 -9.290154 8.649248
Point 3.125148 -7.677539 -5.233937
Point 4.029233 -8.312551 -0.478354
Point 2.601833 8.167995 5.230083
Point -6.664861 0.562662 2.441849
Point 5.003445 -3.522960 -3.876358
Point -8.750782 -7.294186 1.573799
```

Reading the data back

- 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 numberic value
        line=line.split()
        x=float(line[1])
        y=float(line[2])
        z=float(line[3])
        print "%f␣%f␣%f" %(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.

Data Representation

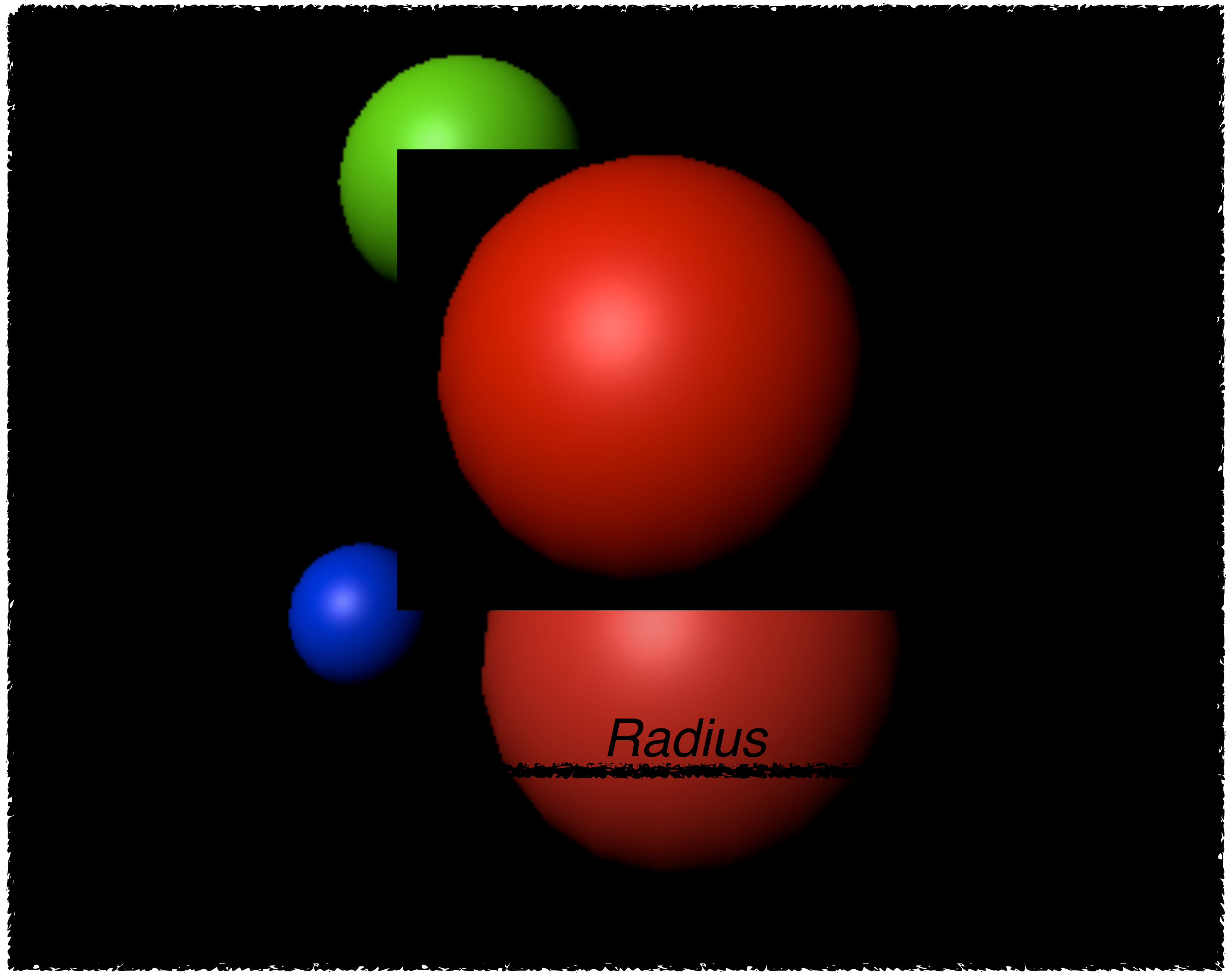
- Most programming tasks are the representation and manipulation of data
- When programming for graphics we need to think in terms of the representation of data (usually numbers)
- The visualisation of this data is usually very easy.
- However storing and manipulation this data is not.
- Usually we will create data structures (or classes) to hold the data and apply algorithms to this data to change it
- Finally we visualise it (draw to the screen)

Structures

- Often programs manipulate objects which have several different parts. In C we can create variables called structures
- This allows us to store records of data regarding a particular subject
- Each part may be of a different type
- Each record may have several components/attributes

Exercise Pt 1

- Consider the image opposite
- How can we describe the individual Spheres?
- And come up with a generic description of a sphere?



Abstraction

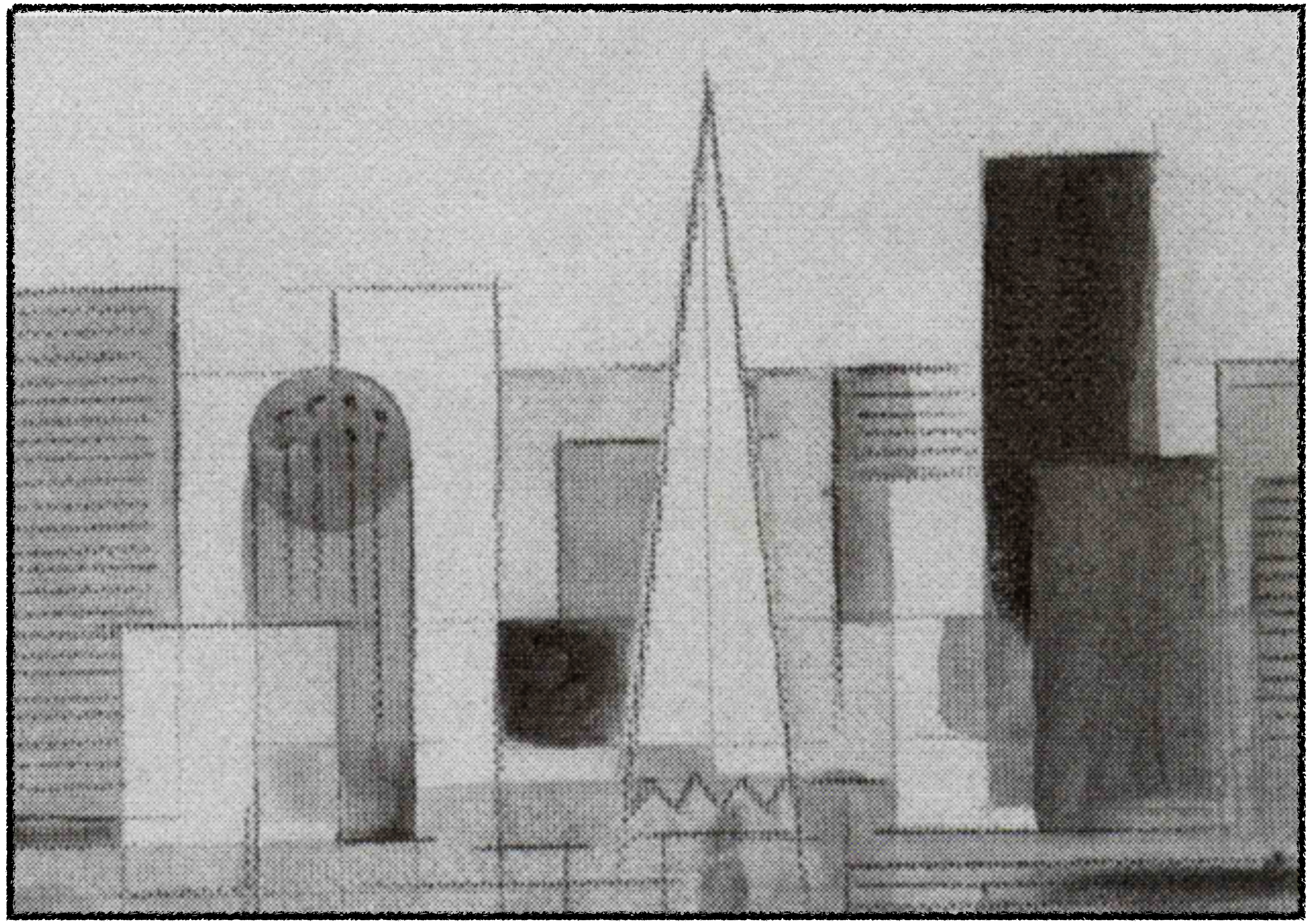
- In philosophical terminology abstraction is the thought process wherein ideas are distanced from objects.

- Abstraction uses a strategy of simplification of detail, wherein formerly concrete details are left ambiguous, vague, or undefined; thus speaking of things in the abstract demands that the listener have an intuitive or common experience with the speaker, if the speaker expects to be understood

- For example, many different things have the property of redness: lots of things are red (Parsons 2000)

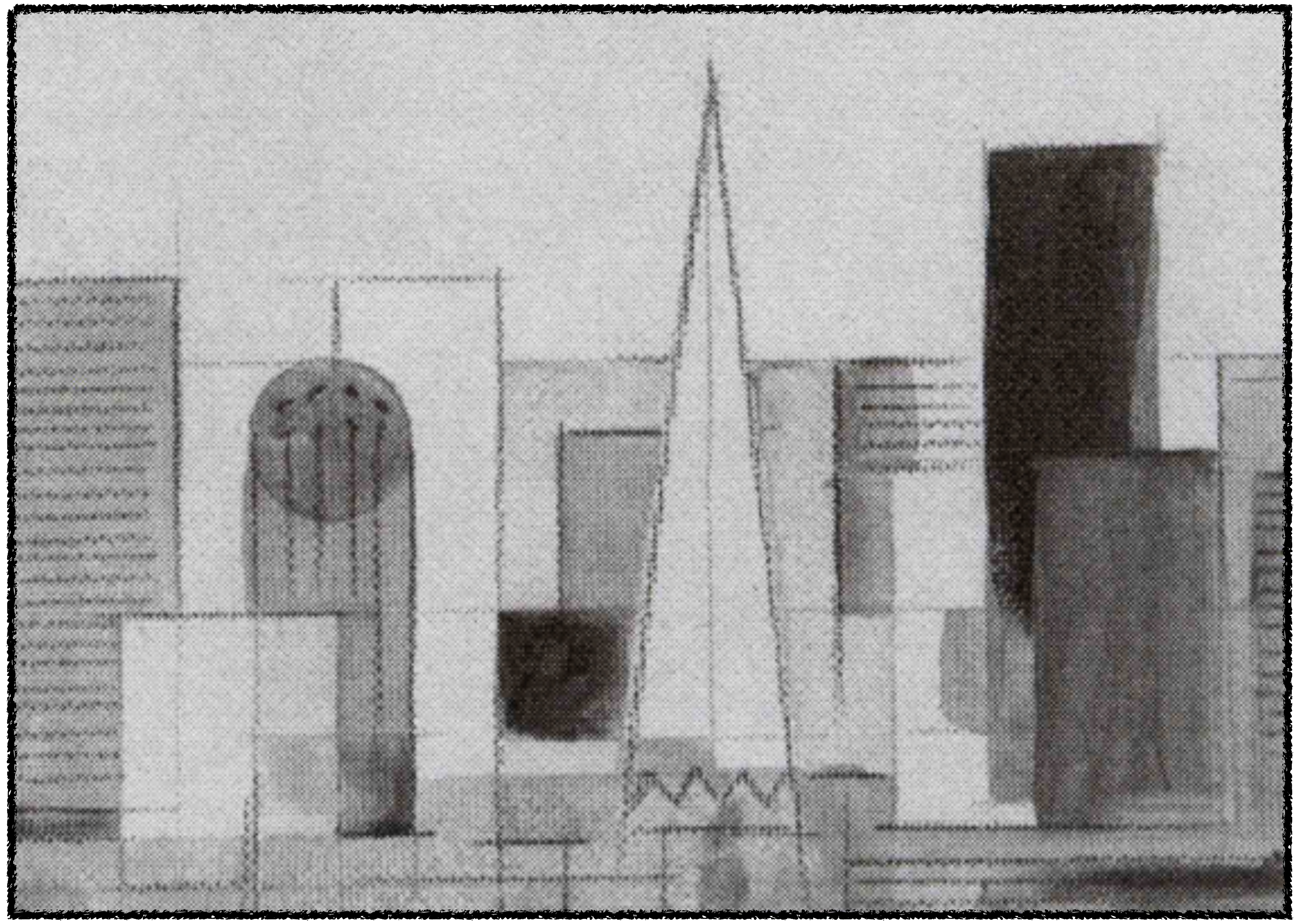
Deductive Abstraction (ZETTL 2008)

In the deductive approach to abstraction
we move from photographic realism to
the essential qualities of the event



Inductive Abstraction (Zettl 2008)

In the inductive approach to abstraction
we study the formal elements of a painting, or of
television or film, and arrange these elements to
express the essential qualities of an event. In this case,
we combine lines, circles, and area to build up
(inductively) the essence of a cityscape

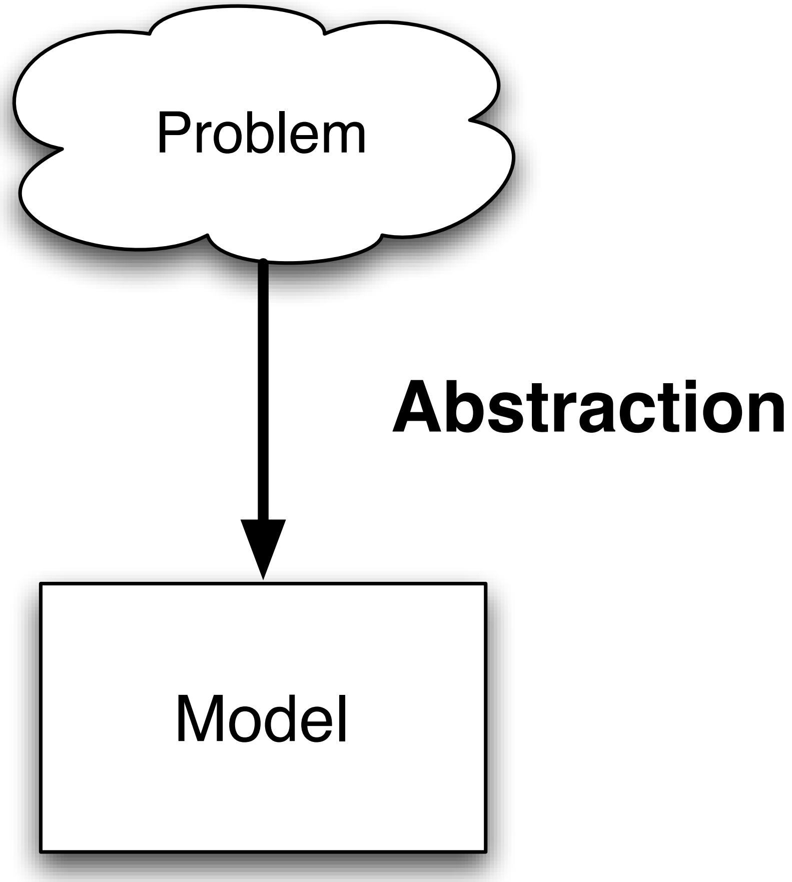


So which approach do we use?

- It depends upon the situation
- Our experience in design and programming
- Factors about the system we are designing
- Factors about development environment.

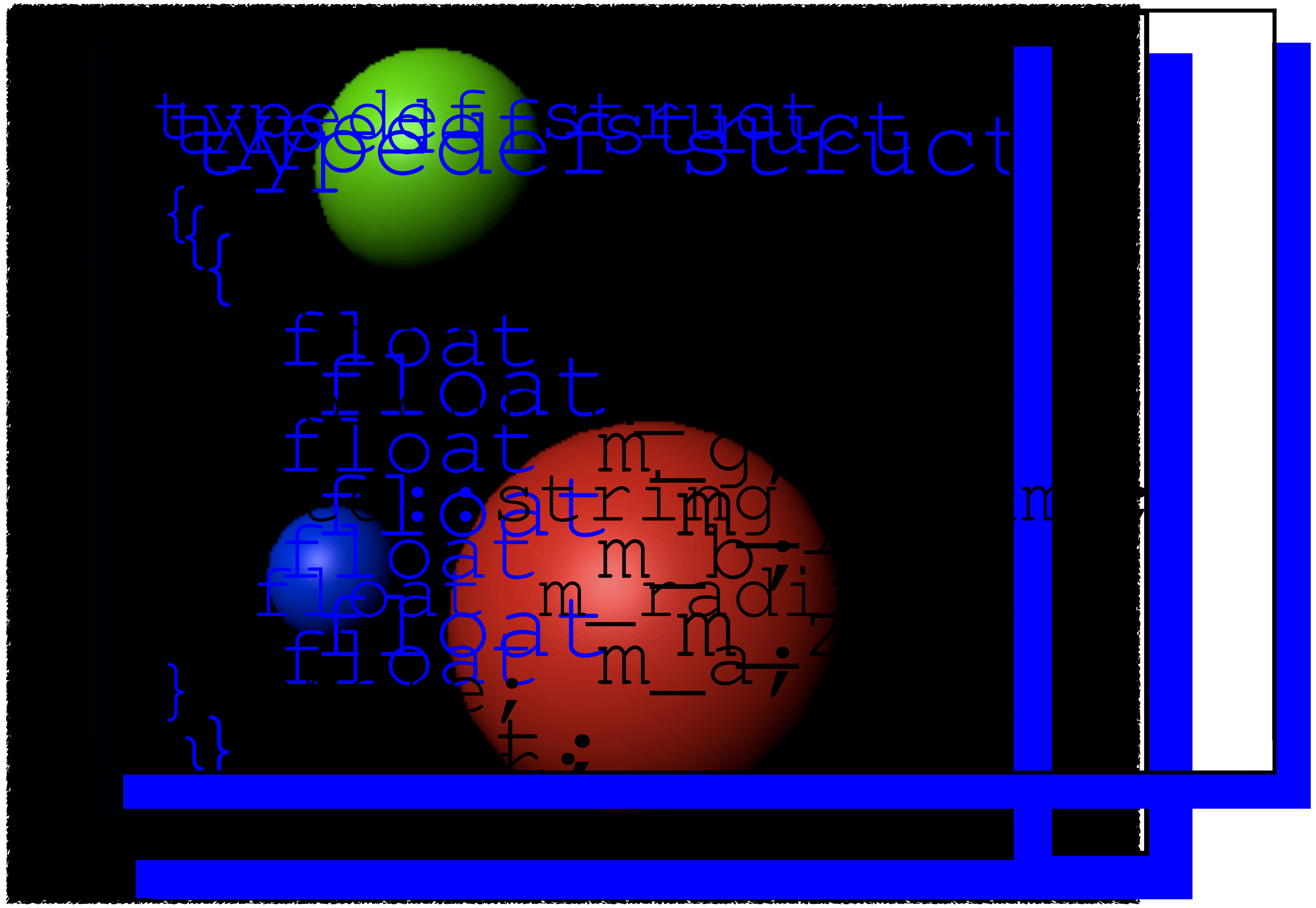
Handling Problems

- Designing Software for real life problems (or CGI in our case)
- However real life problems are “nebulous”
- So we must separate the “necessary” from the “unnecessary”
- This is know as “abstraction”



Exercise Pt 2.

- Consider the Sphere
  - Position
  - Colour
  - Identifier
  - Radius



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>
```

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)


  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

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

```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) :
    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
```

```python
#!/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

```
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__

- __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

```python
#!/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

```
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, >=
```

```
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.

```
#!/usr/bin/python

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.

```
__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
```

```
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

```
__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
```

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)
```

Composition

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

```
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)
```

```
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)
```

```
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
```

```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")
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

```
#!/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
```

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

class Parent(object):

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

  def bar(self) :
    print "bar␣called␣self=%s" %(self)

  def __str__(self) :
    return "Parent"

#######################################
class Child(Parent):
  def foo(self):
    print "foo␣called␣self=%s" %(self)

  def __str__(self) :
    return "Child"

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

parent = Parent()
child = Child()

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

```
foo called self=Parent
foo called self=Child
bar called self=Parent
bar called self=Child
```

```
#!/usr/bin/python

class Parent(object):

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

  def foo(self):
    print "foo␣called␣self=%s␣a=%r" %(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␣called␣self=%s␣a=%r␣b=%r" %(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://www.redhat.com/docs/manuals/linux/RHL-7.2-Manual/getting-started-guide/ch-doslinux.html
- http://www.artima.com/weblogs/viewpost.jsp?thread=4829
- http://www.tutorialspoint.com/python/python_variable_types.htm
- https://docs.python.org/2/tutorial/modules.html

Further Reading

- 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