An Introduction to Programming in Python

Stephen White <stephen@earth.li>

What is Python?

- Portable open source interpreted programming language
- Object Oriented (by design)
- Named after Monty Python's Flying Circus

Why use Python?

- Fairly simple, clean and consistant syntax (unlike Perl)
- Lots of modules to save re-inventing the wheel (like Perl)
- It's fairly easy to write extentions to Python in C and to execute Python code from within C

This talk

The examples from this talk will assume Python 1.5.2 or newer (though many will work on older versions). I will try and point out when I have used features from Python 2 or Python 2.2

A first Python program

```
#!/usr/bin/python

print "Hello World!"
```

Well it had to be didn't it :)

```
[stephen@eddie stephen]$ ./hello.py
Hello World!
[stephen@eddie stephen]$
```

No surprises there (other than the note that *print* has given us a newline automatically). We can also use python interactively:

```
[stephen@FRANKIE pres]$ python
Python 2.2 (#1, Dec 31 2001, 15:21:18)
[GCC 2.95.3-5 (cygwin special)] on cygwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World"
Hello World
>>> "Hello World"
'Hello World'
>>> 3+4
7
>>> i=7
>>> i**2
49
>>>
```

Some syntax: variables

- Variables are all references to Objects
- Variables are used without declaration
- Lists, tuples and dictionaries (hashes or associative arrays in other languages) are built-in data types

Reference assignment of Lists in action:

```
>>> a=[1,2]
>>> b=a
>>> a.append("lots")
>>> b
[1, 2, 'lots']
```

Python does not try to be clever. Despite runtime type checking and lack of explicit declarations, a String is a String and an int is an int.

```
>>> s="1"
>>> s=s+1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Lists

We've seen that Lists are mutable objects, which have methods such as *list.append(value)*, *list.insert(index,value)* and also *list.pop()*. We've also seen that you can create them using square brackets ('[' and ']').

Elements in lists can be accessed and modified using a similar syntax to arrays in other languages. If we specify a negative index then Python counts from the end of the list.

```
>>> a=[1,2,'lots']
>>> a[0]
1
>>> a[0]=0
>>> a
[0, 2, 'lots']
>>> a[-1]
'lots'
```

This array syntax is extended to support 'slices'. We can select a range of values from the List using *a[0:2]*. If we leave off either the upper or lower bound then the range will extend to the end of the list. If we leave off both bounds we get the whole list. This gives us a way to duplicate (or clone) Lists.

Lists(2)

```
>>> a=[1,2]
>>> b=a[:]
>>> b.append(3)
>>> b
[1, 2, 3]
>>> a
[1, 2]
```

We can assign to slices, and even modify the length of the List by doing so.

```
>>> a=[1,2,"lots"]
>>> a[0:2]=[100]
>>> a
[100, 'lots']
```

We can also remove elements from lists using the *del* keyword, and obtain their length using the built-in function *len(list)*.

```
>>> a=[1,2,"lots"]
>>> len(a)
3
>>> del a[0]
>>> a
[2, 'lots']
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'a' is not defined
```

Strings

Like in JAVA, String objects are immutable - so (unlike Lists) once you have a reference to a String the contents of that String cannot change.

Strings can be specified in various ways:

```
s="a \"normal\" string"
s='single quotes may be used instead'
s='strings can span lines \
like this'
s=r'a raw string, \ has no effect'
s=u'used for Unicode strings in Python 2'
s="""This string has:
* newlines in it
* can contain "quotes"
* We could have used '''string''' to achieve the same"""
```

The '+' character is used to concatenate strings. In Python 1.5 most operations on strings are provided by the string module.

```
>>> import string
>>> string.upper('Hello')
'HELLO'
>>> string.split('Hello world')
['Hello', 'world']
>>> string.center('Hello',76)
'                                   Hello                                    '
```

Strings(2)

In Python 2 most of the methods from the string module are available as methods on the string objects themselves.

```
>>> "Hello".upper()
'HELLO'
```

String to integer conversion should be done by calling built-in methods *int(string)* and *long(string)*.

```
>>> s="1"+"0"
>>> int(s)
10
```

Dictionaries

Dictionaries are associated arrays, like hashes (from Perl). This is an unordered set of key-value pairs.

If you haven't come accross these before an example is probably easiest.

```
>>> names = {'spiffy': '10.0.1.6', 'sticky': '10.0.1.4'}
>>> names['spiffy']
'10.0.1.6'
>>> names['scooby'] = '10.0.1.8'
>>> names
{'sticky': '10.0.1.4', 'scooby': '10.0.1.8', 'spiffy': '10.0.1.6'}
>>> names.keys()
['sticky', 'scooby', 'spiffy']
>>> names.values()
['10.0.1.4', '10.0.1.8', '10.0.1.6']
>>> names.has_key('sticky')
1
>>> names.has_key('foo')
0
>>> names['foo']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: foo
```

Tuples

Tuples are immutable Lists, which makes them useful as (for example) keys in Dictionaries.

```
>>> t = 1,2,'lots'
>>> t[0]
1
>>> t[3]=3
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Tuples can be specified inside parentheses, which allows us to create specify empty tuples and also nest them.

Anyone who's thought about parsing while have realised that using the syntax described the specification of a tuple containing exactly one element is ambigious. The solution is slightly ugly and is to put a comma after the one and only element.

```
>>> t = 'abc',
>>> t
('abc',)
>>> len(t)
1
```

Blocks

Before talking about loops we need to briefly mention blocks of code. Unlike C's *{* and *}* or Pascal's *begin* and *end*, Python has no keywords for the begin and end of a block. Instead Python uses the level of indentation to work out the block structure. While this may seem weird it's not really a problem, since it just forces a coding style that most (sane) people use anyway.

```
>>> print "a"
a
>>> print "b"
b
>>>  print "c"
  File "<stdin>", line 1
    print "c"
    ^
SyntaxError: invalid syntax
```

Yes, that syntax error was a result of the invalid indentation. The error could be more helpful. If you use Python this will catch you out sometimes. I often get caught out by trying to write things like:

```
if first==second
   and third==forth:
```

Loops

```
for i in [1,2,"lots"]:
  print "Iteration",i

for i in range(1,4):
  print "Iteration (2)",i

for i in range(100):
  if i<5: print i,"is small"
  if 13<=i<=19: print i,"is in the teens"
  if i in [6,28,496,8128]: print i,"is perfect"

x = 10
while x >= 0:
  print "x is still not negative."
  x=x-1 # x-- not supported :(, and even x-=1 is Python 2.x only

txt = open("myfile.txt")
for line in txt.readlines():
  print line
close(txt)
```

Procedures

```
def fib(n):
  "Return the nth term of the Fibonacci sequence inefficiently"
  if n<=1:
    return n
  else:
    return fib(n-1) + fib(n-2)
```

Paramters can be given default values to create functions that take variable number of arguments (you can also create functions that take arbitary numbers of paramters, but I'm not covering that).

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
  ...

parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

Functions can be referenced

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

Documentation Strings

My *fib(n)* function on the previous slide had a String as the first line of the body. This is a convention called a 'Documentation String', which may be used by debuggers or IDEs and so on. Furthermore:

```
    >>> fib.__doc__
    'Return the nth term of the Fibonacci sequence inefficiently'
    >>> len.__doc__
    'len(object) -> integer\012\012Return the number of items of a sequence or mappin
    >>> import string
    >>> string.__doc__
    'Common string manipulations.\012\012Public module variables:\012\012whitespace -
```

Classes may have documentation strings as well, though I have generally not included them for brevity.

Classes

```
class foo:
  def __init__(self):
    self.x = 0

  def getX(self):
    return self.x

  def setX(self,val):
    self.x = val
```

That, rather boring, class should be fairly self explanatory to anyone who's done any Object Oriented Programming.

```
>>> a = foo() # Create a new instance of class foo
>>> b = foo() # Create another new instance of class foo
>>> a.getX()
0
>>> a.setX(3)
>>> b.setX(7)
>>> a.x
3
```

- Constructors may take arguments
- Classes may be derived from other classes, using *class bar (foo):* to declare that bar inherits from foo.
- Multiple inheritance is supported.

Classes (2)

- Attributes may be made 'class private' by naming them with two leading underscores, e.g. *self.__i*. This isn't bullet-proof, if some-one knows what they're doing they can still modify __i from outside your class.

Explicit inclusision of 'self' in the parameter list allows us to call methods in the superclass without special syntax:

```
class foo (bar):
  def a_method(self,param):
    # Call super
    bar.a_method(self,param)
    # Do stuff
    print "Some stuff"
```

CGI (and print)

```
#!/usr/bin/python

import cgi
form = cgi.FieldStorage()

print "Content-type: text/plain\n\n"

print "Yup, it's a CGI in Python\n"

if (form.has_key("user")): print "user given is %s" % (form["user"].value)
```

That example introduces a new operater, the '%' format operator for strings. It works much like *printf* in C. After the '%' is a tuple of values that get inserted at the points marked by '%s', '%i', etc. There is a modification of this syntax that is useful for turning multi-line strings into templates for web pages (or whatever).

```
>>> a = '%(lang)s has %(c)03d quote types.' % {'c':2, 'lang':'Python'}
>>> a
'Python has 002 quote types.'
```

Exceptions

```
import getopt, sys

try:
  optlist, args = getopt.getopt(sys.argv[1:],"hv",["help","version"])
except getopt.error, e:
  print "Error:",e
  usage()
```

Exceptions are thrown using *raise*. String, classes or instances can be thrown as exceptions. In the above example *getopt* would have code such as *raise getopt.error, "Some error message*. You need not give a tuple to raise if the exception contains all the information needed without extra arguments.

Operator Overloading

```
class frac:
  def __init__(self,num=0,den=1):
    self.__num = num
    self.__den = den
    self.normalize()

  def normalize(self): pass # TODO: implement GCD stuff :)

  def __repr__(self):
    if self.__den == 1: return '%s' % self.__num
    else:              return '%s/%s' % (self.__num,self.__den)

  def __mul__(self, b):
    return frac(self.__num * b.__num, self.__den * b.__den)

>>> f = frac()
>>> print f
0
>>> a = frac(1,4)
>>> b = frac(2)
>>> print a*b
2/4

# __setitem__(self,index,value)
# __getitem__(self,index) - make class look like array
# __call__(self,arg) - make class look like function
# __getattr__(self,name) - overload attribute calls
# __del__(self) - destructor
# __contains__(self,x) - overload if a in foo
```

Binary Data-Structures

When reading from files or sockets we may encouter binary structures that need processing. Use the module 'struct'.

```
import struct

head = struct.unpack("2i", sock.recv(8))
ver = head[0]
msgLen = head[1]
```

The format string can include characters to change the endianness and alignment characteristics. There is a utility function *calcsize(fmt)* to tell you how many bytes of data a given format string corresponds to.

Since Python Strings are not null-terminated binary data can be (is) stored as Strings prior to unpacking, or after packing.

Databases

- There is a standard Python API for accessing Databases.
- I don't think more code with SQL mized in would serve any purpose here, so here are some URLS.
- http://www.python.org/topics/database/DatabaseAPI-2.0.html
- There are modules to talk to most major databases (inc ODBC) at: http://www.python.org/topics/database/modules.html
- See also http://www.amk.ca/python/writing/DB-API.html

Embedding Python in C

There's quite a nice API for embedding Python in C programs, and for calling C functions from Python. It's documented on the web, but for the impatient ...

```
#include <Python.h>
...

int main(int argc, char *argv[]) {
  Py_Initialize();
  pModule = PyImport_Import(PyString_FromString("watch"));
  if (pModule == NULL) {
    printf("Error: python module load failed\n"); PyErr_Print(); exit(1);
  }
  pDict = PyModule_GetDict(pModule);

  PyObject *pFunc = PyDict_GetItemString(pDict,somefunctionname);
  if (pFunc && PyCallable_Check(pFunc)) {
    PyObject *pArgs = PyTuple_New(0);
    PyObject *pValue = PyObject_CallObject(pFunc, pArgs);
    if (pValue != NULL) {
      ...
      Py_DECREF(pValue);
    } else { ... }
    Py_DECREF(pArgs);
  }

  Py_DECREF(pModule);
  Py_Finalize();
  return 0;
}
```

Embedding Python in C

Again, to avoid having to read the documentation, if you're running Redhat Linux you can compile that with:

```
gcc foo.c -o foo -L/usr/lib/python1.5/config/ -lpython1.5 -lpython1.5
  -lm -lpthread -ldl
```

C can provide functions to Python too, there's a standard API that can be used within programs like the one on the last slide as well as in modules design to be used from 'normal' Python. The following would allow Python called from the previous example to 'import stuff' then use 'stuff.aMethod(i)' to call the C method 'aMethod'.

```
static PyObject* aMethod (PyObject *self, PyObject *args) {
  if (PyArg_ParseTuple(args,"i",&retVal)) {
    ..
  } else {
    return NULL;
  }
}

static PyMethodDef pyMethods[] = {
  { "aMethod", aMethod, METH_VARARGS, "Do some stuff with a int argument" },
  {NULL, NULL, 0, NULL} };

...

Py_InitModule("stuff", pyMethods);
```

OS Module

Python provides modules to provide access to operating system features. There are specific modules for each operating system, but you can get at the common functionality by using the 'os' module (which automatically delegates to the right module for your operating system). The os module also provides information, such as the path seperator character.

```
#!/usr/bin/python

import os
import pwd
import string

name = string.split(pwd.getpwuid(os.getuid())[4],',')[0];

print 'hello', name
```

Credits and Questions

- Created using Pythonpoint from Reportlab
- http://www.python.org./doc/current/tut/tut.html
- http://www.python.org./doc/current/modindex.html
- Questions?

Extras

- *a,b = 1,2* works
- Re-usable scripts

```
#!/usr/bin/python

def do_stuff(someArgs):
  ...

if __name__ == '__main__':
  import someModules
  do_stuff(someValues)
```

ReportLab

```
Before reportLab would work I needed to install the imaging module:

http://www.pythonware.com/downloads/Imaging-1.1.3.tar.gz
run 'python2 setup.py build; sudo python2 setup.py install' on Redhat 7.2
on cygwin (with X installed) symlink /usr/X11R6/include/X11 to /usr/include/X11
and in /usr/lib/  libtcl8.0.a -> libtcl80.a and libtk8.0.a -> libtk80.a
```

Functional Programming

A few functional programming constructs are supported, as well as list comprehensions (examples of which I haven't included). Lack of Lazy evalution makes it much less fun though. Map is useful, since it is written in C is can be faster than 'for' loops in some circumstances.

```
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])

>>> foldl = reduce    # also have map and filter
>>> def inc(x,y): return x+1
...
>>> foldl(inc,range(1,10000))
9999
>>> foldl(inc,range(1,40000000))
[ 3 minutes of CPU and 614MB of RAM later ]
39999999

However hugs can't do that the naive way either
inc x y = x+1

Main> foldl inc 1 [1..40000]

ERROR - Control stack overflow
Main> foldl inc 1 [1..40000000]

ERROR - Garbage collection fails to reclaim sufficient space
```

Sockets

There's a socket module. There's a surprise.

```
from socket import *
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(HOST, PORT)
```

Iterators and Generators

Python 2.2 includes some new features, it defines 'iterators' - a away of writing classes that behave like sequences - and generators - a rather bizzare way of having a function return a sequence.

It's possible to achieve some of this with older versions of Python, as demonstrated in the following example.

```
>>> class foo:
...   def __getitem__(self,i):
...     if (i <= 100): return "[%s]" % i
...     else:          raise IndexError()
...
>>> for i in foo():
...   print i
...
[0]
[1]
[2]
....
[98]
[99]
[100]
>>>
```