“Interactive Computing” Overview

- Introduction to Python
- Introduction to XML
- Introduction to WebServices

- Exercise session (3h) on Thursday

Introduction to Python

2004 CERN School of Computing,

Vico Equense

Andreas Pfeiffer

CERN, PH/SFT

Overview

- Python
- Python types
- OO and Python
- Modules, file I/O and serialization
- Extending Python and modules

Python

- **Python is an Agile Programming Language** excellent for beginners, yet superb for experts
  - highly scalable, suitable for large projects as well as small ones
  - rapid development cycles: ideal for Prototyping
  - portable, cross-platform (Unices, Windows, Mac, ...)
  - embeddable
  - easily extensible
  - object-oriented
  - you can get the job done
  - simple yet elegant
  - stable and mature
  - powerful standard libs
  - wealth of 3rd party packages
- And don't forget that with Python, programming is fun again!

Multi-paradigm

- Imperative
  - “command-line” style
- Object oriented
  - Classes and methods
- Functional
  - Sequential
- Introspection
  - dir(), type(), isinstance(), .__name__

Incremental development

- No edit-compile-debug cycle
  - Interpreted language
- Make tiny changes and test them immediately
- Program state is not lost
- *Can reduce development time by an order of magnitude or more !*
  - *Rapid Application Development (RAD)*

Python types (I)

- Numerical types
  - ***int, float*** (double)
    - 1 , 3.1415926
  - ***Complex***
    - 1 + 1j
  - ***long*** (arbitrary precision)
    - E.g., factorial(100)

Python types (II)

- Sequences
  - ***List*** – mutable, heterogeneous
    - [] , [1, 2] , [1, “hi there”, 42.]
    - [[1,2], [3,4], [4,5]] – list of lists
  - ***Tuple*** – immutable, heterogeneous
    - () , (1,) , (1, ‘hello’, 42.)
    - Parentheses are not always needed
  - ***String*** – immutable, homogeneous
    - ‘a string’, “another string”, ‘with “ quotes’

Dictionaries

- ***Dictionaries*** are hash-tables (maps)
  - >>> d = {} # empty dict
  - >>> d[1] = ‘one’
  - >>> d[‘two’] = 2
- Heavily used in Python’s implementation, so dictionaries are highly optimized
- Replacement for “switch” construct
  - dispatch={ ‘q’ : quit, ‘r’ : redisplay, ‘e’ : evaluate}
  - reply = get_reply() ; dispatch[reply]()

Sequence indexing and slicing

>>> a = range(10)	*[0,1,2,3,4,5,6,7,8,9]*
>>> a[3]	*3*
>>> a[3:6]	*[3,4,5,6]*
>>> a[-1]	*9*
>>> a[-2]	*8*
>>> a[2:5] = [‘x’]	*[0,1,’x’,5,6,7,8,9]*
>>> a[-1:] = [‘a’, ‘b’, ‘c’]	*[0, 1, 'x', 5, 6, 7, 8, 'a', 'b', 'c']*

Unpacking tuples

- >>> a,b,c = 1,2,3
- >>> a,b = b,a
- >>> w = 5,6,7
- >>> x,y,z = w
- Use tuples to return multiple values from functions
  - Output parameters are “un-Pythonic”

Note the lack of parentheses

Indentation

- Python uses indentation to determine the structure of blocks (methods/functions)

```
if True :
    print 'eggs'
    print 'spam'
else :
    print 'ni ni'
```

Blocks start with colons

- Empty block: ***pass***

Loops

- Python has two loop constructs
  - `while … :`
  - `for … in … :`

Both have an optional `else:` *clause*

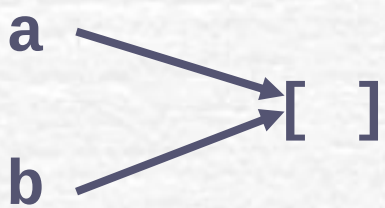
```
for i in range(10):
    print i, i*i
```

  - For-loops work with any iterable
  - Use **`xrange`** rather than **`range`** for large ranges
    - **`range`** creates a list, **`xrange`** generates the numbers as required

Variables, binding, call-by-value

- Variables do not have type; objects have type – dynamic binding
- Binding is the association of a variable with an object
- Python uses call-by-value semantics
  - But all non-numeric values are references !

```
a = [5]
b = a
b[0] = 3
```



a == [3]

Functions

```
def my_function(arg1, arg2) :
    """The optional documentation string
       goes here"""
    a = arg1
    b = arg2
    c = a + b
    return a,b,c  # optional

do_something_else()
```

a,b,c are local variables

returns copies of a,b,c

More on Functions

```
def parrot(voltage, state='a stiff',
                   action='voom', type='Norwegian Blue'): pass
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
```

```
def cheeseshop(kind, *arguments, **keywords): pass
cheeseshop('Limburger',
                   "It's very runny, sir.",
                   "It's really very, VERY runny, sir.",
                   client='John Cleese', shopkeeper='Michael Palin',
                   sketch='Cheese Shop Sketch')
```

OO in Python

- Group code and data together
  - Polymorphism
  - Encapsulation
- Use of “self” in methods
- All methods are virtual
- Private and protected only by convention
- “Magic methods”
  - __init__(self) → constructor
    - baseClass.__init__(self)
    - __str__(self) → formatter for print

*A “cat” **is an** “animal”*
*A “car” **has an** “engine”*

Classes in Python

"old-style" class:

```
class counter :
```

```
class counter (object):
    def __init__ (self,start):
        self.count = start
    def up(self, n=1):
        self.count += n
    def down(self, n=1):
        self.count -= n
```

Constructor

```
a = counter(10)
a.up()
print a
```

Inheritance and operators

```
class addcounter(counter):
   def __repr__(self):
     return '<counter: ' + str(self.count) + '>'
   def __add__(self, other):
     return addcounter(self.count + other.count)
```


```
addcounter(3) + addcounter(4)
```

Private members

- Python does not enforce ‘privacy’
  - Unlike C++ and Java
- *Convention*: names starting with a single underscore, refer to objects you should not access directly, outside their defining scope
  - `self._localVariable`
- Identifiers starting (but not ending) with two underscores will be mangled; intended as protection against ACCIDENTAL clobbering
  - `self.__mangledVariable`
- Identifiers both starting and ending with two underscores are language-defined special names
  - `self.__init__`

Modules

- Group together functionality
- Provide namespaces
  - `import moduleName`
  - `from moduleName import *`

```
> moduleName.spam
> spam
```

- Means of extending Python
  - Also using other languages (C, C++,...)
- Python comes with a vast collection of modules in its standard library
  - “Batteries included”
  - More on “Vaults of Parnassus”
    - http://www.vex.net/parnassus/

The os and sys modules

- Using os.system
  - To execute commands in a shell
  - `os.system("ls -al")`
- Using os.path
  - All kind of PATH related functionality
  - `os.path.walk(dir,self.checkForLocks,"")`
- Extending the PYTHONPATH
  - `sys.path.append("/my/dir/PyModules")`

Dealing with strings

- The string module: lots of useful methods for string handling
  - `words = string.split(line, “-”)`
  - `line  = string.join(words, “:”)`
  - `index = string.find(line,“:”)`
- The re module: regular expressions
  - Very powerful !
  - `m=re.match(r"(?P<int>\d+)\.(\d*)",'3.14')`
    - `m.group(1), m.group(‘int’)` is 3
    - `m.group(2)` is 14
  - “nightmare to debug”

Optionally specify `start,end`

Exceptions

```
try:
  # code body
except ArithmeticError:
  # what to do if arithmetic error
except IndexError, data:
  # what to do if index error
except:
  # what to do for any other error
else:
  # what to do if no exception
```

Throwing exceptions:

```
raise IndexError [,data]
```

optional

```
try:
  # code body
finally:
  # what to do ALWAYS ... e.g. some "clean-up" code
```

LBYL vs. EAFP

- Look Before You Leap

```
if denominator == 0:
  print "Oops"
else:
  print numerator/denominator
```

- Easier to Ask Forgiveness than Permission

```
try:
   print numerator/denominator
except ZeroDivisonError:
   print "Oops"
```

"Pythonic" way !

Exception hierarchy

- The standard exceptions are organised in an inheritance hierarchy
  - E.g. ZeroDivisionError is a subclass of ArithmeticError
- Allows you to catch a “range” of exceptions with a single statement
  - E.g. : `except ArithmeticError:`
- You can derive (== extend) your own exceptions from any of the standard ones.
  - E.g. `class MyOverflow(ArithmeticError):`

Files

- Print and print >>file

```
>>> file = open('myfile','w')
>>> print >> file, 1,2,3,4
>>> file.write('5 6 7 8')
>>> file.close()

>>> file = open('myfile','r')
>>> for line in file: print line
```

- sys.stdin, sys.stdout, sys.stderr
  - Are “normal” files

Object persistency in Python

- “Serialization” of complex objects
  - Conversion into/from set of bytes
  - Sent over network, stored/read from file
  - Aka: “pickling”, “marshalling” or “flattening”
- “DBM” style
  - Provide namespace (key) for the objects
  - Similar to dictionary and file
  - Can store only strings, no python objects

Object persistency (II)

- Two modules for serialization
  - `marshal` for only simple python objects
  - `pickle` for recursive objects, complex, user-defined classes
- One module for all
  - `shelve` pickling of Python objects as well as a DBM storage for the flattened objects

Command line options

- List of command-line args: **sys.argv**
  - Number of items: **len(sys.argv)**
- Module for manipulation: **getopt**

```
import getopt
try:
  optlist, args = getopt.getopt(sys.argv[1:],['?','h'],['help','tmpDir='])
except :
  print "\nunknown option.\n"
  usage()
  raise
for o, a in optlist:
   if o in ("-?", "-h", "--help"):
      usage() ; sys.exit()
   elif o in ("--tmpDir",):
      tmpDir = a
```

Extending Python

- Using C or C++ modules
- Tools to help with the “boring” part
  - SWIG, boost::python, ...
  - Convert C/C++ header files to Python files ***and*** write the “glue” code using the C-API of Python for you !
  - Flexibility to change interface
    - Adding, removing, renaming of methods
    - Templates, STL ongoing work, but about ok

Items for further study

- Lambda
  - The anonymous function
- Generators
  - Iterating “like in C++” ☺
- List Comprehensions
  - `[ x*x for x in range(10) ]`
- Testing
  - The unittest framework

Python in HEP

- Interactive sessions of experiment or analysis framework
  - Using the C++ classes of the framework
- “Glue” various toolkits/frameworks together
  - Loosely coupled components with well-defined (abstract) interfaces as Python modules !
- Rapid Application Development
  - Develop an algorithm in Python, then convert it into C++ component (performance) and deploy it

Thanks

- To Jacek Generowicz for allowing me to (re-)use his slides
- To Guido van Rossum for the creation of Python
- To all writers of modules
- To you for coming !

References

- Python (with lots of interesting links)
  - http://www.python.org
- The “Python Cookbook”, lots of “recipes”
  - http://aspn.activestate.com/ASPN/Python/Cookbook/
- Vaults of Parnassus, lots of user-land Python modules
  - http://www.vex.net/parnassus/

Optional slides

The Zen of Python - part 1/2

(Formulated by Tim Peters)

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.

The Zen of Python - part 2/2

12. In the face of ambiguity,
    refuse the temptation to guess.
13. There should be one —and preferably only one— obvious way
    to do it.
14. Although that way may not be obvious at first unless you're
    Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain,
    it's a bad idea.
18. If the implementation is easy to explain,
    it may be a good idea.
19. Namespaces are one honking great idea
    — let's do more of those!