Basic python Tutorial

Martin Woudstra
(University of Massachusetts)

Muon Week
10 February 2009

Introduction

- *“Python is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days.”* (Official python website)
- Today: few hours, so touch on main features to get you going
- Documentation:
  - The official python website: http://www.python.org
  - Useful small reference booklet:
    Python Pocket Reference, Mark Lutz (O'Reilly)
- ATLAS currently uses version 2.5

How to run python code

- Interactive in the python interpreter:

```
python
>>> print “hello”
hello
>>> CTRL-d  # to exit python
```

- From a script (containing the above print line):

```
python hello.py
```

- Self-running script:
  - file contents:

```
#!/usr/bin/env python
# This script prints hello to the screen
print hello
```

  - then on shell command prompt:

```
chmod +x hello.py
./hello.py
hello
```

A few words on the syntax

- Everything after a hash (#) is a comment (until the end of the line)
- A statement ends at the end of the line
- Multiple statements on the same line are separated by a semicolon (;)
- A block of code is defined by its equal indentation
  - Don't use tabs, always use spaces ! (tab = 8 spaces)
- A backslash at the end of a line joins it with the next line - like in shell scripts
- An unclosed (), [] or {} pair also continues to the next line(s) until the closing),] or }

```
v = [ “muon” ,
      “electron” ]
```

Built-in basic types

- int : integer (at least 32bit)
- float : floating point, like C++ double
- complex
- str : string (constant), like C++ const char*
- bool : True or False
- list : a vector, like C++ std::vector
- tuple : constant list
- dict : a map, like C++ std::map
- You can ask an object for it's type with the 'type' function:

```
>>> type(“hello”)
<type 'str'>
>>> type(“hello”).__name__
'str'
```

operators

- + - * / ** : addition, subtraction, multiplication, division, power
- += -= *= /= : operator and assignment in one go (as C++)
- % : modulus (int), format (str)
- or and : logical OR and AND
- not : logical negation
- < <= > >= : comparison
- == != : equality
- is is not : object identity (pointer comparison)
- in not in : membership test
- | ^ & ~ << >> : bitwise operators, as in C++
- X < Y < Z : True if Y is in between X and Z

Lists
(std::vector in C++)

```
v = []   # empty list
v = list()  # empty list
v = [ 1, 2, 4, 5 ]  ; v = [ 'a', 'b', 'c' ]
v = range(4,10,2) # results in [ 4,6,8 ]
v = [ 4, 2.5, 'Hi', [ 1,3,5 ] ]  # can mix types
```

- Appending elements

```
>>> v.append( 70 )
>>> print v
```

- Concatenation

```
>>> v += [ 'some', 'more', 'elements' ]
>>> v  # shows the object
```

- Removal of elements

```
>>> v.remove(2.5)
>>> del v[0]
```

- Size

```
>>> len(v)
3
```

Lists (2)

- Element acces read/write

```
>>> v[0]
'hi'
>>> v[0] = 'hey'
>>> v[-1] # last element. Negative = count from the end
>>> v[1:3] # subrange by index (start index, one-beyond-last index)
```

- Test if an element is in a list (or not):

```
>>> if 4 in v:
...   print “Found it”
Found it
>>> if 200 not in v:
...   print “Not found”
Not found
```

- Looping over lists

```
>>> for i in v:
...   print i
```

- Several more functions (sort, reverse, index, count, ...)

List comprehensions

- A compact way to make lists in a for loop

```
>>> v = [ expression for expr1 in sequence1 if condition1 \
                     for expr2 in sequence2 if condition2 \
                     for exprN in sequenceN if conditionN ]
```

  - is short for

```
>>> v = []
>>> for expr1 in sequence1:
...   if condition1:
...     for expr2 in sequence2:
...       if condition2:
...         for exprN in sequenceN:
...           if conditionN:
...             v.append( expression )
```

  - For example

```
>>> v = [ x**2 for x in range(10) if x % 3 == 0 ]
>>> v
[0, 9, 36, 81]
```

strings
(const char* in C++)

- Use single quotes (') , double quotes (“), or triple double quotes (“”“) (for multi-line string):

```
>>> a = 'Hello'
>>> b = “How are you?”
>>> c = “””First line
... Second line”””
>>> aa = “'” ; bb = '”' # one type of quote can be used inside the other
```

- Concatenation:

```
>>> d = a + “! ”
>>> d    # will show object on screen
'Hello! '   # Note the quotes
>>> d += b
>>> print d
Hello! How are you?   # no quotes printed here
```

- Indexing

```
>>> d[1]  # read-only.  d[0] is first character
'e'
```

strings (2)

- Substring

```
>>> d[7:10]  # start at char at index 7 (8th char), up to (but not including) char at index 10 (11th char)
'How'
>>> d[-4:] # start at 4th char from the end, continue until the end
'you?'
```

- Searching

```
>>> d.find('y')  # returns the index. -1 if not found
15
>>> d.find('How') # returns index of first character
7
```

- Formatting (similar to C printf)

```
>>> print “%6.4f %4d” % (5.2,200)
5.2000  200
```

  - “%s” prints any object (calls __str__ function on object)
  - “%r” prints any object in python executable format (calls __repr__ function), e.g. for strings it will add the quotes

strings (3)

- Repetition:

```
>>> 10 * “#”
'##########'
```

- split string into list of strings based on split character

```
>>> ds = d.split() # default split character is whitespace
>>> ds
['Hello!', 'How', 'are', 'you?' ]
```

- join a list of strings into one string bases on a join character

```
>>> dj = '~~~'.join(ds)
>>> dj
'Hello!~~~How~~~are~~~you?'
```

- Many more functions (see doc)

Comparison: == vs. is

- == : compare the contents of the object
- is : compare the memory address (pointer comparison)
- Bad usage (seen in ATLAS):

```
>>> test = “hi”
>>> if test is “hi”:
>>>    print “test passed”
>>> else
>>>    print “test failed”
test passed
```

- But it works ????
  - Python optimizes memory usage and recycles strings !

```
>>> test = “h” + “i”
>>> test
>>> test is “hi”
False
>>> test == “hi”
True
```

dicts
(std::map in C++)

- Dict = list with arbitrary objects as index

```
>>> k = { }  # empty dictionary
>>> k = dict() # empty dictionary
>>> k = { 'muon' : 4,  'electron : 2 }  # key : value
>>> k['muon']
4
```

- Set/add members

```
>>> k['jet'] = 1  # add or overwrite an entry
>>> k.update( 'tau' : 1 , 'gamma' : 2 ) # add from other dict
>>> k.setdefault( 'muon', 1 ) # only sets if not already there
```

- Useful access functions:

```
>>> k.get('muon', 0) # returns 0 if 'muon' is not in dict
>>> k.keys()  # returns the keys in a list
>>> k.values()  # returns the values in a list
>>> k.items()  # list of (key,value) tuples
```

dicts (2)

- Removal of elements

```
>>> del k['electron']
```

- Looping

```
>>> for n in k:  # loops over keys
...   print n, n[k]   # bad example, use items() to do this
>>> for n,v in k.items(): # n,v is an implicit tuple filled for every
pass in the loop
...   print “k[%r]=%r” % (n,v)
```

- Test if element is in dict

```
>>> if 'muon' in k:  # searches in the keys
...   print “Got a muon”
>>> if 'electron' not in k:
...   print “No electrons”
```

booleans

- True, False (note capitalization)
- Each type has an associated boolean value

Type	True	False
`int, float`	non-zero	0
`list`	non-empty	empty []
`dict`	non-empty	empty {}
`str`	non-empty	empty “”

`X or Y`

- possible on any type of X and Y
- returns X if X evaluates to True (and does not check Y), else it returns Y
- return type not necessarily bool, but type of X or Y !!!

- `X and Y` same as `or`, except:
  - returns X if X evaluates to False (and does not check Y), else it returns Y

Variables

- All variables are references (to real objects)
- All variables have a dynamic type
  - you don't set the type of the variables
  - they have the type of the object they point to
  - the type can change

```
>>> type(a)
<type 'str'>
>>>  a = 5
>>> type(a)
<type 'int'>
>>> b = a   # b = 5, a and b now point to the same object
>>> a = 7   # does this change the object pointed to?
>>> b = ??? Guess! 5 or 7 ?
```

Variables (2)

```
>>> 5 # a = 7 does not change the object (5),
      # it just points to a new object (7) !
      # b still points to the (unchanged) constant object 5
```

- => behavior of simple objects 'natural'
- Beware of lists (and dicts)

```
>>> a = [ 1, 2, 3, 4 ]
>>> b = a      # b now points to the same vector
>>> b[3] = 6   # changes an entry in the vector
>>> print a    # is changed
[1, 2, 3, 6]
```

  - Now a is changed by changing b !
  - Same for `b.append(7)` and `b += [8,9]`

- And strings...

```
>>> a = “hello”
>>> b = a
>>> a += “ !”
>>> b  # Guess...
```

Making copies

- What about making copies (instead of references) ?
  - lists

```
>>> v1 = [ 1, 2, 3, 4 ]
>>> v2 = v1[:]  # makes a copy ('sub'-vector of the complete vector)
>>> v3 = list(v1)  # another way to make a copy
>>> v3[3] = 8  # leaves v1 unchanged
>>> import copy
>>> v4 = copy.copy(v1)
```

  - dicts

```
>>>  d1 = { “muon” : 2,  “electron” : 2 }
>>> d2 = d1.copy()
>>> d3 = dict(d1)
>>> d4 = copy.copy(v1)
```

- Shallow copy vs deep copy
  - all methods above make a 'shallow' copy: make a new list/dict and fill it with the references to the elements of the original
  - v5 = copy.deepcopy(v1) makes also copies of the elements, recursively

Memory management

- Objects are ref-counted and are deleted automatically if no more references point to it

```
>>> del a  # deletes the variable a, not the object
>>> del b  # delete last reference, hence deletes object
```

Flow control

- If-then-else

```
if 'muon' in d1:
  print “Found a muon”
elif 'electron' in d1:
  print “Found an electron, but no muon”
else:
  print “Did not find a muon nor an electron”
```

- for loop

```
for i in v1: # loop over all entries in v1
  if i == 'muon':
    processMuon()
    continue  # go to next item in the list
  if i == 'unknown':
    break  # bail out of the loop
else:
  # executed if no 'break' was executed
  print “All OK: No unknown particles”
```

- while loop : see documentation

Exceptions

- *“Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is raised at the point where the error is detected; it may be handled by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred.”*
- The Python interpreter raises an exception when it detects a run-time error. Users can also raise exceptions in their code.
- To raise an exception:

```
raise RuntimeError(“Something went terribly wrong”)
```

Exception class

Information attached to the exception

Exceptions

- To catch exceptions:

```
try:
  # exceptions raised in this block are caught
  mistake = d1['oops']
except KeyError, e:
  # A KeyError exceptions was raised, and will be assigned to e
  print e # print the exception
except AttributeError, e:
  # An AttributeError was raised
  print e
except Exception, e:
   # any other exception raised (that derives from class Exception)
   print e
else:
  # No exception was raised
  pass # do nothing
```

Exceptions

- To raise an exception:

```
raise RuntimeError("Something went terribly wrong")
```

- Many built-in exceptions used by python, e.g.
  - KeyError : raised when reading a non-existing key in a dict
  - IndexError : raised in case of an out-of-range index in a list
  - AttributeError : raised when reading a non-existing attribute (data member or member function)
  - NameError: raised when using a variable that has not been assigned
  - SyntaxError: Raised when the parser encounters a syntax error
  - ImportError: raised when an import fails to find a module or the requested attribute
  - RuntimeError: "a rarely used catch-all" (pocket reference)
- All built-in exceptions derive from class Exception

Exceptions

- Full list of built-in Exceptions: http://docs.python.org/library/exceptions.html
- Make your own custom exception class

```
class MyException(Exception):
  pass
```

  - Recommended to derive from class Exception (not required)

Miscellaneous

- `pass` : no-operation, sometimes needed for correct syntax
- exec <some_string> : execute string as python command

```
>>> exec “z = 10”
>>> z
10
```

  - real life example (from MuonRecFlags.py):

```
for flag in [“makeRIO”, “readRDOPool”]:
  for tech in [“MDT”, “RPC”, “CSC”, “TGC”]:
    exec 'DetFlags.%s.%s_setOn()' % (flag,tech)
```

- `execfile(“hello.py”)` : execute statements in file hello.py
- `None` : a singleton object, useful for setting something to 'unspecified' or 'uninitialized'

```
a = None
if a is None: # can use 'is' because there is only one None object
  a = 40 # take some default value
```

functions

- Definition of a function:

```
def muonSelector(particle, momentum=500, *vargs, **kwargs):
  if particle == 'muon':
    return momentum
  else:
    return 0
```

  - `particle` is an obligatory argument
  - `momentum` is an optional argument with default value 500
  - `vargs` is a list of (unspecified) optional positional arguments.
    - name 'vargs' has no special meaning – just easy to remember: vector of arguments
    - not too useful (?)
  - `kwargs` is a dict containing optional keyword arguments (see later
    - name 'kwargs' has no special meaning – just easy to remember: keyword arguments
    - very useful – used in configurables
  - arguments specify no type (since dynamic), but in practice the function expects certain types internally

functions (2)

- The function return type is also dynamic (can be any type), but in practice the caller expects a certain type to be returned
- If no `return` statement is hit before the end of the function, `None` is returned
- Calling a function

```
p = selectMuon('muon', 200, “cosmic”, 0.3, bField=True, source=”data”)
```

  - inside the function (of the previous slide), the variables will be

```
particle = 'muon'
momentum = 200
vargs = [ “cosmic”, 0.3 ]
kwargs = { “bField” : True , “source” : “data” }
```

  - one can also call positional variables by their name:

```
p = selectedMuon(particle='muon', momentum=200)
```

classes

```
class Track:
   """Documentation string"""
   def __init__(self,p=0,eta=0,phi=0):
     self.p = p
     self.eta = eta
     self.phi = phi

>>> t1 = Track()
>>> t2 = Track(200)
>>> t3 = Track(200,0.5,1.2)
>>> print t1

>>> t1.p = 100 # change existing member
>>> t1.z       # will raise exception
>>> t1.z = 0.5 # add new data member
>>> t1.z
>>> del t1.z  # remove attribute
```

- __init__ is the constructor
- the first argument ('self' by convention) is the reference to the object (the instance of the class). Required in all member functions.
- the data members (p,eta,phi) are added dynamically to the object by the statements in __init__. They are stored in the member __dict__.

classes (2)

- Pretty printout through function __str__. Called by
  - print
  - built-in function str()
  - string format %s
  - equivalent of C++
    ostream& operator<<(ostream& os, MyClass& anObject)

```
class Track:
   <snip>
   def __str__(self):
     """nice string representation"""
     return "p=%8.2f eta=%5.2f phi=5.2f" % (self.p,self.eta,self.phi)
>>> t1 = Track()
>>> print t1
```

- Member functions that access data members should always use self.<member> (different from C++)

some built-in functions

- accessing attributes

```
getattr(t1,”p”)          # same as t1.p
setattr(t1,”eta”, -2.1)  # same as t1.eta = -2.1
delattr(t1,”z”)          # same as del t1.z
hasattr(t1,”x”) # check if object has a member (returns True or False)
if hasattr(ToolSvc,”MySuperTool”):
   # do something with the tool
else:
   # add the tool
```

`dir([object])` : list with names of variables of object or of current scope if no object given

`vars([object])` : dict with variables (name,theVariable) of object or current scope

- type conversion

```
str(t1) : convert object t1 to string ('pretty print')
bool(t1) : convert object t1 to bool
float(x) : convert number or string x to a floating point
int(x) : convert number or stringr x to an integer
```

classes (3)

- Inheritance

```
class MuonTrack(Track,Muon):
    def __init__(self,p,eta,phi,muonType):
      super(MuonTrack,self).__init__(p,eta,phi)
      self.muonType = muonType

    def __str__(self):
      s = super(MuonTrack,self).__str__()
      s += “muon type=%s” % self.muonType
```

- base classes given in parenthesis
- derived class __init__ should call base class __init__, for example with super construct.

- Private vs. public
  - All members are public by default
  - If a member starts with one underscore (_), it is considered private (by convention only – use at own risk)
  - If a member starts with two underscores (__), it is (kind of) private: the member name is mangled with the class name, so you can still access it if you really want to (at even greater risk)

__slots__

- Not really basic python, but used in ATLAS
- It is a special class data member holding the list of possible object data members

```
class Track:
    __slots__ = ( 'p', 'eta', 'phi' )
    def __init__(self,p=0,eta=0,phi=0):
      self.p = p
      self.eta = eta
      self.phi = phi
```

- One can no longer add new data members dynamically. The purpose (in ATLAS) is to avoid typos.

```
>>> t = Track()
>>> t.z = 0.5  # will throw an exception
```

- Needs to be repeated in each derived class, otherwise one can still add members to that class (which defeats the purpose)

Modules

- A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended.
- You can access code in modules by importing it:

```
import MyModule
```

- All code resides in the namespace of the module. If a module defines a class SomeClass, then the class can be accessed with

```
a = MyModule.SomeClass()
```

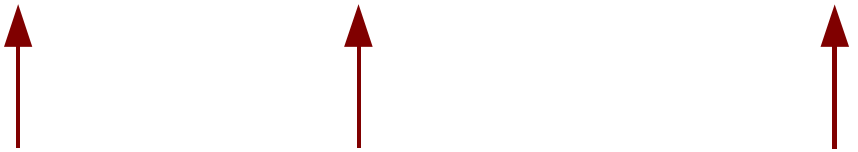
- You can also makes code directly available in the current namespace:

```
from MyModule import SomeClass
a = SomeClass()
```

Modules

- If a directory contains a file __init__.py, this directory is considered a module. The __init__.py file is executed when the module is imported, but it can be empty. The python files in that directory are then sub-modules of that module.

```
from MuonRecExample.MuonRecFlags import muonRecFlags
```



- import looks for modules in the paths given in PYTHONPATH
- The first import statements executes all code in the module
- All following imports don't execute anything, they just get a reference to the already loaded module

Standard modules

- Extensive list of modules part of the language
- sys : system information

```
>>> import sys
>>> sys.path  # list with paths to search for python modules
              # (i.e. PYTHONPATH + some system paths)
>>> sys.modules # dict of loaded modules
>>>> sys.argv # list of command line arguments
              # sys.argv[0] = name of python script being run
              # sys.argv[1:] : arguments to the script
```

- os : operating system access, file access
- math : mathematical functions (sin,cos,pow,sqrt,pi,...)
- re : regular expressions
- Full list : http://docs.python.org/modindex.html