COMPUTING METHODS IN HIGH ENERGY PHYSICS

S. Lehti

Helsinki Institute of Physics

Spring 2025

Python

Python is a dynamic programming language used in a wide variety of application domains.

It is object-oriented, interpreted, and interactive programming language.

It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems. New built-in modules are easily written in C or C++ (or other languages).

A recommended coding style is to use 4-space indentation, and no tabs.

CMS software (CMSSW) configuration files use Python.

More in http://www.python.org/

Example 1:

```
#!/usr/bin/env python
import sys,os,re
root_re = re.compile("(?P<rootfile>([^/]*))\.root")
def main():
    path = os.path.abspath(sys.argv[1])
    dirs = execute("ls %s"%path)
    for dir in dirs:
        dir = os.path.join(path,dir)
        if os.path.isdir(dir):
            subdirs = execute("ls %s"%dir)
            if subdirs.count("res") == 1:
                dir = os.path.join(dir,"res")
                files = execute("ls %s"%dir)
                for file in files:
                    match = root_re.search(file)
                    if match:
                        print "file:"+os.path.join(dir,file)
def execute(cmd):
    f = os.popen(cmd)
    ret=[]
    for line in f:
        ret.append(line.replace("\n",""))
    f.close()
    return ret

if __name__ == "__main__":
    main()
```

Command line parameters

If the command line parameters of your python script grow in complexity, one possibility is to use *optparse* instead of sys.argv. Optparse is a flexible parser for command line options. It allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Example 2:

Code:

```
from optparse import OptionParser
parser = OptionParser()

parser.add_option('-f','-file',

dest='filename')
(options, args) = parser.parse_args()
print 'options',options
print 'args',args
print options.filename
```

Output:

```
$ python example1.py -f TEST1 TEST2
options {'filename':  'TEST1'}
args ['TEST2']
TEST1
```

The file name can be accessed from the "options" object:

```
fIN = open(options.filename)
```

Anything left over from parsing options are pushed into list 'args', like the argument 'TEST2' above.

Types supported by optparse are string, int and float.

```
parser.add_option('-n', type='int',
dest='number')

myint = options.number
```

Python

If no type is specified, string type is assumed. Booleans can be set with action store_true/store_false

```
parser.add_option('-v',
action='store_true', dest='verbose')
```

in this case having an argument '-v' will set options.verbose = True.

Default values can be set

```
parser.add_option('-v',action='store_true',
dest='verbose',default=True)
```

Help and usage can be generated automatically by supplying a help value for each option.

```
parser.add_option('-f','-file',
dest='filename', help='Input filename')
```

Example 3:

Output:

```
$ python example1.py -h
Usage:  example1.py [options]
Options:
  -h, -help    show this help message and
exit
  -f FILENAME, -file=FILENAME
                Input filename
```

Regular expressions

The re module provides Perl-style regular expression patterns. You can use REs to match a string, to modify a string or to split it apart in various ways. Most letters and characters will simply match themselves. For example, the regular expression "test" will match the string 'test'

Python

exactly. Some characters are special metacharacters, and don't match themselves, unless escaped with a backslash.

Example 4:

```
import re
root_re = re.compile(
    '(?P<filename>(\S+))\.root$'
)
names =
['test.root','/a/b/c/test.root','root.txt']
for name in names:
    match = root_re.search(name)
    print name
    if match:
        print '    Match
found:',match.group('filename')
    else:
        print '    ',name,'did not match'
```

Output:

```
$ python example3.py
test.root
    Match found:  test
/a/b/c/test.root
    Match found:  /a/b/c/test
root.txt
    root.txt did not match
```

Now, if we are interested only in the filename, not the path, we could use re to select the body:

`re.compile('(?P<label>[^/]\S+)\.root$')`

How to read `'(?P<filename>[^/]\S+)\.root$'`? It starts with a named grouping so that we can select the string inside the group easily: (?P<filename>...). An unnamed grouping is done with ().

Python

Outside the grouping we have \.root$, a dot being a metacharacter matching it needs an escape \., followed by root, and the dollar sign tells the string must end. Inside the group we have [^/]\S+, \S being any non-whitespace character, + meaning it's repeated one or more times, and [^/] meaning it does not start with /.

Example 5:

A CMSSW (crab) job produces two different output files like tau_99_1_zRD.root and tau-highpurity_99_1_zRD.root. How to select the correct files only? A re 'tau_\d+_\d_\S+\.root' will select the former files, while 'tau-highpurity_\d+_\d_\S+\.root' the latter.

There are other possibilities to construct the expressions for this use case (try finding some), but these two examples are easily readable.

Example 6:

Picking up a value for parameter 'dataVersion' from crab.log from a string like `'-CMSSW.pycfg_params= dataVersion=74Xdata:energy=8: trigger=HLT_IsoMu15_v7:runOnCrab=1'` can be done using regular expression `'dataVersion=(?P<dataVersion>\S+?):'`. So it picks up a string between 'dataVersion=' and a ':'. However, since regexp is greedy, it wants to select a string

'74Xdata:energy=8:trigger=HLT_IsoMu15_v7' between the first 'dataVersion=' and the last ':'. This is prevented by telling the re to be non-greedy: this is done by using a qualifier '+?' instead of (the greedy) '+'.

Paths

os.path is a very powerful tool for path manipulation. How it works is obvious from the commands

- joined path = os.path.join(path1,path2,path2)
- filename = os.path.basename(pathplusfile)
- if os.path.exists(mypath): ...
- dir = os.path.dirname(pathplusfile)
- abs path = os.path.abspath("../../somepath")

You might also need a 'PWD': os.getcwd()

Data structures

- Lists l = ['a','b','c']
- Nested lists nl = [['a','b','c'],['d','e','f']] (= matrix)
- Tuples consist of a number of values separated by commas: t = 12345, 54321, 'hello!'
- Sets are unordered collections with no duplicate elements. s = set(['a','b','c'])
- Dictionaries are unordered sets of key: value pairs d = 'first': 12, 'second': 34

Python

User's own data structures with class:

```
class MyClass:
    def __init__(self,var1,var2):
        self.var1 = var1
        self.var2 = var2
```

Class inheritance

```
class BaseClass:
class MyClass(BaseClass):
```

Data hiding

Python protects hidden members by internally changing the name to include the class name. You can access such attributes as object._className__attrName.

Example 7:

```
#!/usr/bin/python
class JustCounter:
    __secretCount = 0
    def count(self):
        self.__secretCount += 1
        print self.__secretCount
counter = JustCounter()
counter.count()
counter.count()

print counter.__secretCount
```

When the above code is executed, it produces the following result:

```
1
2
Traceback (most recent call last):
  File 'example6.py', line 12, in <module>
    print counter.__secretCount

AttributeError:  JustCounter instance has no attribute

'__secretCount'
```

This, however, will work:

```
print counter._JustCounter__secretCount
```

The sys module

The sys module provides information about constants, functions and methods of the Python interpreter.
* Command line arguments
sys.argv
* Changing the output behaviour of the interactive Python shell
sys.displayhook
* Standard data streams
sys.stdin, sys.stdout, sys.stderr
* Redirections
* module paths
sys.path

Example 8:

Code:

```
import sys
print('Coming through stdout')
save_stdout = sys.stdout
fh = open('test.txt','w')
sys.stdout = fh
print('This line goes to test.txt')
# return to normal:
sys.stdout = save_stdout

fh.close()
```

Numerical computations: numpy

- scientific computing with Python
- separate package, needs to be installed before usage.

Python

- include arrays, linear algebra, Fourier transform, random numbers, tools for integrating C/C++/Fortran code

Numpy reference manual

Example 9:

Code:

```
import numpy
print 'Array'
a = numpy.array([1,2,3])
print a
print a.dtype

print 'Matrix'
m2x2 = numpy.array([(1.0,0.0),
(0.0,1.0)])
print m2x2
print m2x2.dtype
print numpy.trace(m2x2)
print numpy.linalg.eig(m2x2)
print numpy.linalg.norm(m2x2)
data = [1,2,3,4]
print 'Data=',data
print 'Mean',numpy.mean(data)
print 'Std',numpy.std(data)
```

The lambda-operator

Python has two tools for building functions: def and lambda. The lambda syntax lets you define one-line mini-functions on the fly. Borrowed from Lisp, these lambda functions can be used anywhere a function is required.

Example 10:

Code:

```
def square(number):
    return number*number
lsquare = lambda x:  x*x
a = 5
print a,'sq=',square(a),'(ftion def)'
print a,'sq=',lsquare(a),'(lambda
ftion)'
```

Python

Example 11:

Code:

```
nums = range(1, 10)
for i in range(2, 4):
    nums = filter(lambda x:  x % i == 0,
nums)

print(list(nums))
```

This filters out all the numbers between 1-9 which are divisible by first 2, then 3.
The advantage of the lambda function comes with functions like map(), filter() and reduce(), which take a function as the first argument and a list as the second.

Example 12:

Code:

```
from functools import reduce
list = [1,2,3,4]
a = reduce(lambda x,y:  x+y, list)

print(a)
```

Here the first two elements of list will be applied to the function. Next the function will be applied on the previous result and the third element of the list and so on. Continue like this until just one element is left and return this element as the result of reduce().

Exception handling

Errors detected during execution are called exceptions. Exceptions come in different types, and the type is printed as part of the message. Built-in Exceptions lists the built-in exceptions and their meanings.

Python

Example 13:

Code:

```
  while True:
      try:
          x = int(raw_input('Please enter
a number:  '))
          break
      except ValueError:
          print 'Not a valid number...'
```

There may be several except clauses, and the last one may omit the exception names.

Example 14:

Code:

```
  import sys
  try:
      f = open('myfile.txt')
      s = f.readline()
      i = int(s.strip())
  except IOError as e:
      print 'I/O error({0}):
{1}'.format(e.errno, e.strerror)
  except ValueError:
      print 'Could not convert data to an
integer.'
  except:
      print 'Unexpected error:',
sys.exc_info()[0]
      raise
  else:
      print 'Success!',i
```

Coloring the output

Changing the color of the output helps sometimes. For example when validating something, passed reports can be colored green, and failed ones red, to make spotting the problems faster, easier and more robust.

Python

Example 15:

Code:

```
from itertools import chain
m = 'A message'
colors =
chain(range(30,39),range(40,47),range(90,99))
for color in colors:
    print('\033[%im'%color,m,
          '\033[0m','color=',color)
print('\033[1m',m,'\033[0m','bold')
print('\033[3m',m,'\033[0m','italics')
print('\033[4m',m,'\033[0m','underl.')
```

Magic methods

Magic methods are special methods that you can define to add 'magic' to your classes. They're always surrounded by double underscores like __init__ .
The most basic magic method, __init__, is the way we can define the initialization behavior of an object.

Actually, a method called __new__ creates the instance, then passes any arguments at creation on to the initializer. A method __del__ is the destructor. One of the biggest advantages of using Python's magic methods is that they provide a simple way to make objects behave like built-in types. If for example we define __eq__, we can use the operator ==
if instance == other_instance:..

Example 16:

Code:

```
class Counter:
    def __init__(self,name):
        self.name = name
        self.count = 0
    def __iadd__(self,number):
        self.count += number

        return self
```

Python

```
      def __add__(self,other):
        tmp = Counter(self.name + "+" +
other.name)
        tmp.count = self.count +
other.count
        return tmp
     def __str__(self):
        return 'Counter '+self.name+'
'+str(self.count)

  a = Counter('foo')
  b = Counter('bar')

  a += 2
  b += 1
  c = a+b

  print a
  print b

  print c
```

Multiprocessing

If you can split your one job into several processes, run in parallel, there is quite a gain the execution time of the job.

In multiprocessing, new independent subprocesses are spawned. There is no quarantee of the order the subprocesses are executed. Communication is possible only through dedicated, shared communications channels. These channels must be created before subprocesses are forked.

Example 17:

```
import multiprocessing as mp
MAX_WORKERS =
max(mp.cpu_count()-1,1)
pool = mp.Pool(MAX_WORKERS)
lock = mp.Lock()
N = len(datasets)
for i,d in enumerate(datasets):
    txt = "Dataset %s/%s"%(i+1,N)
    p = mp.Process(target=eventloop,
                   args=[..,txt,lock])
    p.start()
    pids.append(p.pid)
```

Python

```
import psutil
nalive = len(pids)
print "Processes running",nalive
i = 0
os.system('setterm -cursor off')
while nalive > 0:
    al = 0
    for pid in pids:
        if psutil.Process(pid).status() != psutil.STATUS_ZOMBIE:
            al = al+1
    nalive = al
    dt = time.time()-t0
    lock.acquire()
    sys.stdout.write("Processes running %s/%s"%(nalive,len(pids)))
    sys.stdout.flush()
    lock.release()
```