Introduction to Python and Python/EMAN

Steve Ludtke
sludtke@bcm.tmc.edu

Language History

8512 documented languages (vs. 2376)

- Four of the first modern languages (50s):
  - FORTRAN (FORmula TRANslator)
  - LISP (LISt Processor)
  - ALGOL
  - COBOL (COmmon Business Oriented Language)
- C (1972)
- C++ (1983)
- Perl (1990)
- Python (1991)
- Ruby (1992)
- HTML (1994)
- Java (1995)

Python ?

PYTHON OOL- developed by Guido van Rossum, and named after Monty Python.(No one Expects the Inquisition) a simple high-level interpreted language. Combines ideas from ABC, C, Modula-3, and ICON. It bridges the gap between C and shell programming, making it suitable for rapid prototyping or as an extension of C. Rossum wanted to correct some of the ABC problems and keep the best features. At the time, he was working on the AMOEBA distributed OS group, and was looking for a scripting language with a syntax like ABC but with the access to the AMOEBA system calls, so he decided to create a language that was extensible; it is OO and supports packages, modules, classes, user-defined exceptions, a good C interface, dynamic loading of C modules and has no arbritrary restrictions.

www.python.org

So, why not X ?

- C/C++ : Fast code, good reusability, but: risky, pointers, memory management, hard for beginners
- Fortran 77/90/95 - yeeech
- Java : Strongly structured, enforces good programming, but: a pain to use, many portability/version problems, language gets in the way of getting work done
- Perl - Great for text processing, concise syntax, but impossible to learn, remember syntax, or write clean code
- Tcl - Used as embedded control language, and provided Tk, but very limited language features, somewhat difficult syntax

Python Features

- Simple, easy to learn
- Gigantic set of built-in libraries
- Portable (virtually all computers/OS supported)
- Interpreted and byte-compiled (like Java)
- Object oriented
- Very popular for application scripting (especially in scientific apps)
- Python for high-level work, with compute-intensive work in C++/Fortran libraries

Hello World

- Python

```
print “Hello, world!”
```

- Perl

```
print "Hello, world!\n";
```

- C:

```
#include <stdio.h>

main() {
    printf(“Hello, world!\n”);
    exit(0);
}
```

The first program you learn in any programming language.

EMAN2

```
>>> from EMAN2 import *
>>> img=test_image()

>>> imgs=EMData.read_images("imgs.hed")          <-- reads all images
>>> imgs[0].get_attr_dict()
...
>>> for i in imgs: print i.get_attr("maximum")
...
>>> img=EMData()
>>> img.read_image("file.mrc")
>>> print img.get_xsize(),img.get_ysize,img.get_zsize()
...
>>> for i in imgs[1:]: imgs[0]+=i
>>> imgs[0]/=len(imgs)
>>> display(imgs[0])
>>> imgs[0].write_image("out.mrc")
```

Using a Processor

- dump_processors()
- dump_processors_list() - introspection

```
a=EMData()
...
b=a.process("ham e",{ "m":2.0,  "b ":1. 0})
```

- or -

```
a.process_inplace("hame ", {key:value,key:value})
```

- or -

```
# if initialization is expensive
p=Processor.get("ham e", {k:v,k:v})
for i in imgs:
    p.process_inplace(i)
```

Other Modular classes

- dump_aligners()
  - newimg=img.align(“name”,img2,{k:v},”cmpname”,{k,v})
- dump_cmps()
  - img.cmp(“name”, img2,{k:v})
- dump_projectors()
  - prj=vol.project(“name”, {k:v,k:v})

Other Modular classes

- dump_reconstructors()

r=Reconstructors.get(“name”,{k:v,k:v})
r.setup()
r.insert_slice(img,Transform3D)
...
vol=r.finish()

- dump_averagers()

r=Averagers.get(“name”, {k:v,k:v})
r.add_image(img)
...
avg=r.finish()

Interactive Python

- Python as a calculator
- strings, tuples, lists, dictionaries
- import (os,math,sys,string,random,etc.)
- help()
- if, while, for
- def
- files, readline, readlines

Basic Syntax Reference

- Indent
- Numbers:

0 1.5 1.2e43+2j

- Strings:

“test string” 'this too'

“””multiple line
string”””

- Lists:

```
lst=[1,2,'abc',1+3j]
lst[0]
```

- Dictionaries:

```
dict={'key1':'value1','key2':'value2',3:'value3'}
dict['key2']
dict[3]
```

- import:

```
import os
from math import *
```

- print:

```
print 'x=',x,'   y=',y
print 'x=%f y=%f'%(x,y)
```

- if,else:

```
if x>5:  print “x>5”
elif x<0:  print “x<0”
else:  print “0<x<5”
```

- for loops:

```
for i in list: print i
```

- while loops:

```
while x<10:
    print x
    x*=1.5
```

Python as a Calculator

> *python*
>>> ***5*10*** <-- Note that bold italics indicate what you should type at the prompt
50
>>> ***5*10/3***
16
>>> ***5.0*10/3***
16.666666666666668
>>> ***5**2***
25
>>> ***5**3***
125
>>> ***sqrt(4)***
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in -toplevel-
    sqrt(4)
NameError: name 'sqrt' is not defined

Python as a Calculator

```
>>> import math          <-- before we use special math functions, we need to 'import' the
>>> math.sqrt(4)              math library
2.0
>>> math.sqrt(-1)        <-- normal math library does not support imaginary numbers
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in -toplevel-
    math.sqrt(-1)
ValueError: math domain error

>>> import cmath         <-- cmath stands for 'complex math' and supports complex numbers
>>> cmath.sqrt(-1)
1j

>>> from math import *   <-- the '*' is a wildcard meaning read everything, after doing
>>> pi                        this, all math operations are available without 'math.'
3.1415926535897931
>>> sin(pi/2)
1.0
```

Your Own Functions

```
>>> def y(m,x,b):                <-- Now we define it as an actual function we can re-use
        return m*x+b

>>> y(1,2,3)                     <-- Call the function with m=1, x=2 and b=3
5
>>> f(1.0,2,3)
5.0
>>> def g(x,y):
        return float(x)**int(y)          <-- '**' means raise to the power of in python
>>> g(5.0,3)
125.0
>>> g(5,3.0)
(what do you think?)

>>> cos(pi)
-1.0
>>> def cos(x): return x+1.0
>>> cos(pi)
4.1415926535897931
```

Simple Strings

```
>>> "Hello there"
'Hello there'
>>> 'Hello there'
'Hello there'
>>> """Hello There"""
'Hello There'
>>> "'Hello' there"
"'Hello' there"
>>> print "'Hello' there"
'Hello' there
>>> print """This is a
multiline string, denoted by
triple quotes"""
This is a
multiline string, denoted by
triple quotes
```

<--- A simple string

<--- Single quotes equivalent to double

<--- Triple-double-quotes let you span multiple lines

String Slicing and Dicing

```
>>> "Hello there world"
'Hello there world'
>>> "Hello there world"[1]          <-- Returns element number 1 from the string
'e'                                      but the first element is 0, so 1 is the second
>>> "Hello there world"[6:11]       <-- A 'slice of the string, the 7th thru 11th elements
'there'                                  the last element (number 11) is never included
>>> "Hello there world"[:5]         <-- Starts at the beginning if the first number
'Hello'                                  is missing
>>> "Hello there world"[-5:]        <-- Negative values count from the END of the
'world'                                  string, if the 2nd number is missing, go to end
```

```
           1111111
0123456789012345678      <-- counting from the beginning
Hello there world
         987654321       <-- counting from the end
```

Lists and Tuples

```
>>> a=[1,2,3,4]                    # a list of numbers
>>> a
[1, 2, 3, 4]
>>> a[3]
4
>>> sum(a)
10
>>> a=["Hello",2,3]                # a list with a string and 2 numbers
>>> a
['Hello', 2, 3]
>>> sum(a)
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in -toplevel-
    sum(a)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> a[0]=1
>>> sum(a)
6
>>> a
[1, 2, 3]
```

Dictionaries

```
>>> a={"a":0,"boat":4,"in":2,"the":3,"river":5}      # a dictionary. Associates items with
>>> a.keys()                                        # other items. The key is used for
['a', 'the', 'river', 'boat', 'in']                 # retrieving the value. The key must be
>>> a.values()                                      # immutable (number, string, tuple)
[0, 3, 5, 4, 2]
>>> a["a"]
0
>>> a["river"]
5
>>> a["a"]+a["boat"]+a["river"]
9
>>> a["tree"]="root"
>>> a
{'a': 0, 'tree': 'root', 'in': 2, 'the': 3, 'river': 5, 'boat': 4}
>>> a["a"]+a["boat"]+a["tree"]
Traceback (most recent call last):
  File "<pyshell#140>", line 1, in -toplevel-
    a["a"]+a["boat"]+a["tree"]
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> str(a["a"])+str(a["boat"])+a["tree"]
(what do you think?)
```

Help!

Python reference manual online (also often installed with python):
http://docs.python.org/lib/lib.html

>>> help()

...

>>> help(str)

…

>>> help(str.join)

…

>>> help(list.sort)

if

To perform an action only if some condition is true

```
>>> def sign(x):
        if x>0 : return 1
        elif x<0 : return -1
        else return 0

>>> f(1000.0)
1
>>> f(-1000.0)
-1
>>> f(0)
0
>>> f("abc")
1
>>> f("-1")
1
```

if

```
>>> def f(x):
        if not isinstance(x,str) : return "Not a string"
        elif len(s)<5: return "A short string"
        elif len(s)>=5 and len(s)<20 : return "A medium string"
        elif len(s)>=20: return "A long string"

>>> f("abc")
"A short string"
>>> f("This is a test")
"A medium string"
>>> f(5)
"Not a string"
```

Could the above function be simplified ?

if

exactly equivalent:

```
>>> def f(x):
        if not isinstance(x,str) : return “Not a string”
        if len(s)<5: return “A short string”
        if len(s)<20 : return “A medium string”
        return “A long string”
```

Useful operators for if:

<, >, ==, <=, >=, !=
and, or, not, ()
isinstance()
% (modulus)

try, except, pass

To handle errors in a program gracefully

```
>>> int("abc")

Traceback (most recent call last):
  File "<pyshell#52>", line 1, in -toplevel-
    int("abc")
ValueError: invalid literal for int(): abc

>>> try: int("abc")                    # Try to do something
>>> except: print "an error occurred"  # if an error occurs, do this instead

an error occured
```

while

To repeat an action until some condition is met

```
>>> x=0
>>> y=0
>>> while (x<10):                    # will repeat the next 3 lines (indented the same)
        x+=1                         # until x>=10
        y+=x
        print x,y
```

```
1 1
2 3
3 6
4 10
5 15
6 21
7 28
8 36
9 45
10 55
```

for

To repeat an action for each item in a list

```
>>> a=0
>>> for i in [1,2,3]:                    # executes a+=i for i=1,2,3
        a+=i
>>> print a
6

>>> sum([1,2,3])
6

>>> sum(["a","b","c"])
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in -toplevel-
    sum(["a","b"])
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> a=0
>>> for i in ["a","b","c"]:
                a+=i
>>> a
"abc"
```

for

```
>>> a=0
>>> for i in range(1000): a+=i
>>> a
499500

>>> a=["Abc",125.2,100,200,"def",300,1.0]
>>> for i in a:                                    # finds all integers in a list
        if isinstance(i,int) : print i
100
200
300

>>> b=[]
>>> for i in a:
        if isinstance(i,int) : b.append(i)
>>> b
[100,200,300]
```

for

A neat trick:

```
>>> c=[i for i in a if isinstance(i,int)]
>>> c
[100,200,300]
```

Break that down a bit:

```
>>> a=[1,2,3,4]
>>> a=[i*2 for i in a]
>>> a
[2,4,6,8]
```