13th Summer School on SCIENTIFIC VISUALIZATION

Introduction to Python Language

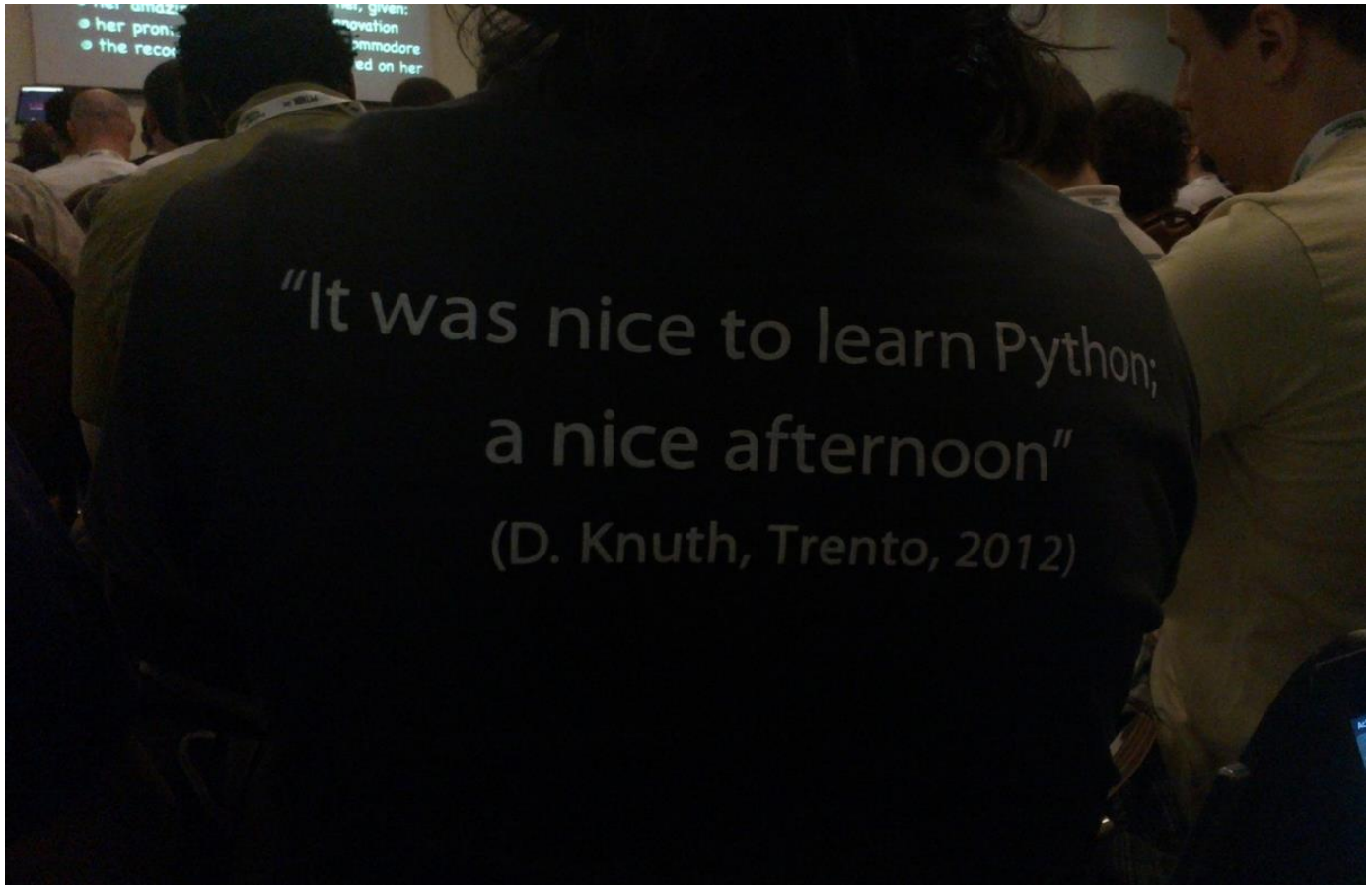
Alice Invernizzi – invernizzi@cineca.it
SuperComputing Applications and Innovation Department

CINECA

INDEX

- Introduction to Python
- Module, OOP introduction, code introspection
- Data type
- Control Flow
- Function
- Read/write to file

Introduction to Python



'*It was nice to learn Python; a nice afternoon*' D.Knuth, Trento 2012

Introduction to Python

```
import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to
do it.
Although that way may not be obvious at first unless you're
Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good
idea.
Namespaces are one honking great idea -- let's do more of
those!
```

Introduction to Python

Python is an easy to learn, powerful programming language.
PYTHON HIGHLIGHTS

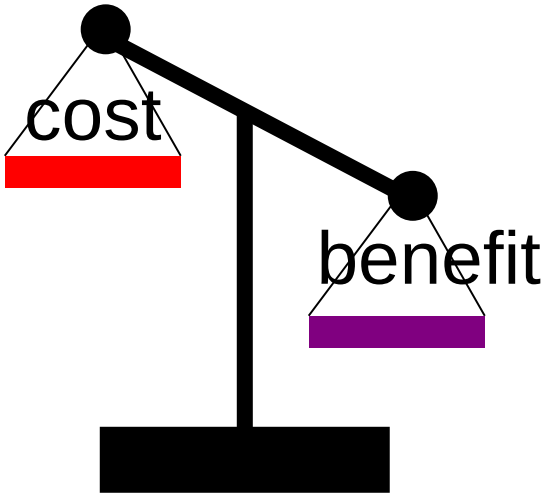
- Automatic garbage collection
- Dynamic typing
- Interpreted and interactive
- Object-oriented
- “Batteries Included”
- Free
- Portable
- Easy to Learn and Use
- Truly Modular

Introduction to Python

Programming Family Language

Interpretated	Compilated
Python	C/C++
Matlab ®	Fortran
Perl	Java

- Execution Performance
- Development effort
- Code Portability
- Code Readability



Introduction to Python

Who is using Python?

NATIONAL SPACE TELESCOPE LABORATORY

LAWRENCE LIVERMORE NATIONAL LABORATORIES

WALT DISNEY

REDHAT

ENTHOUGHT

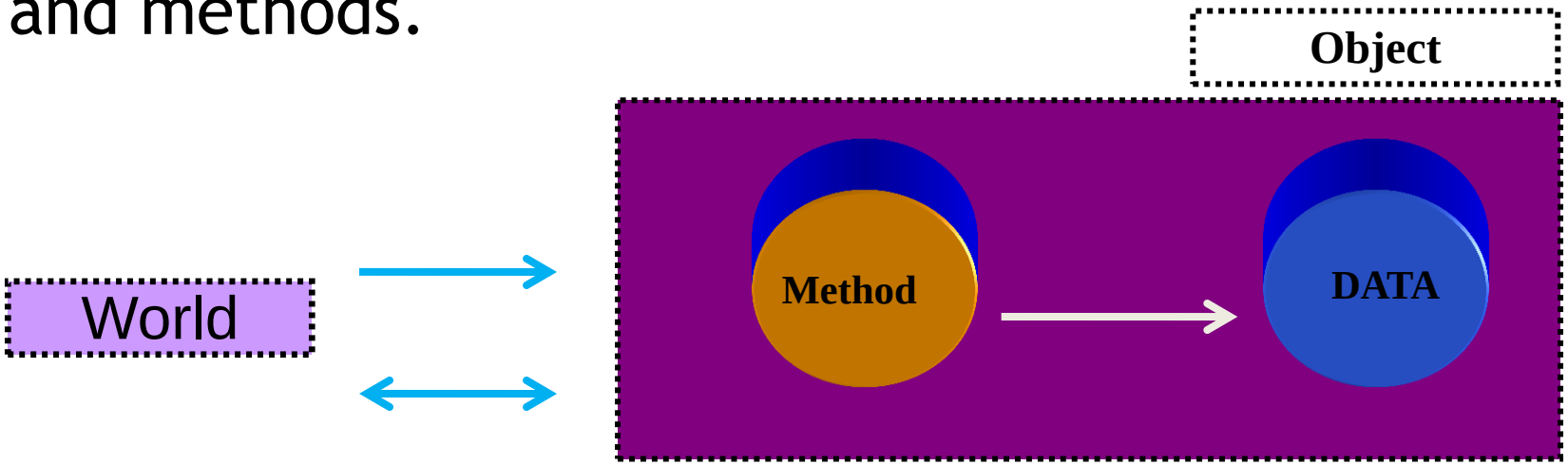
PAINT SHOP PRO 8

…

May 2014	May 2013	Change	Programming Language	Ratings	Change
1	1		C	16.926%	-1.80%
2	2		Java	16.907%	-0.01%
3	3		Objective-C	11.791%	+1.36%
4	4		C++	5.986%	-3.21%
5	7	^	(Visual) Basic	4.197%	-0.46%
6	5	v	C#	3.745%	-2.37%
7	6	v	PHP	3.386%	-2.40%
8	8		Python	3.057%	-1.26%
9	11	^	JavaScript	1.788%	+0.25%
10	9	v	Perl	1.470%	-0.81%
11	12	^	Visual Basic .NET	1.264%	+0.13%
12	10	v	Ruby	1.242%	-0.43%
13	38	^^	F#	1.030%	+0.79%
14	14		Transact-SQL	1.025%	+0.21%
15	17	^	Delphi/Object Pascal	0.974%	+0.24%
16	13	v	Lisp	0.967%	+0.07%
17	19	^	Assembly	0.773%	+0.14%
18	15	v	Pascal	0.752%	-0.05%
19	21	^	MATLAB	0.711%	+0.12%
20	42	^^	ActionScript	0.674%	+0.47%

OOP

- *everything in Python is an object*. Strings are objects. Lists are objects. Functions are objects. Even modules are objects. Everything has attributes and methods.



OOP

In OOP the focus is on the data, and the properties of the data.
Object Oriented Programming is based on:

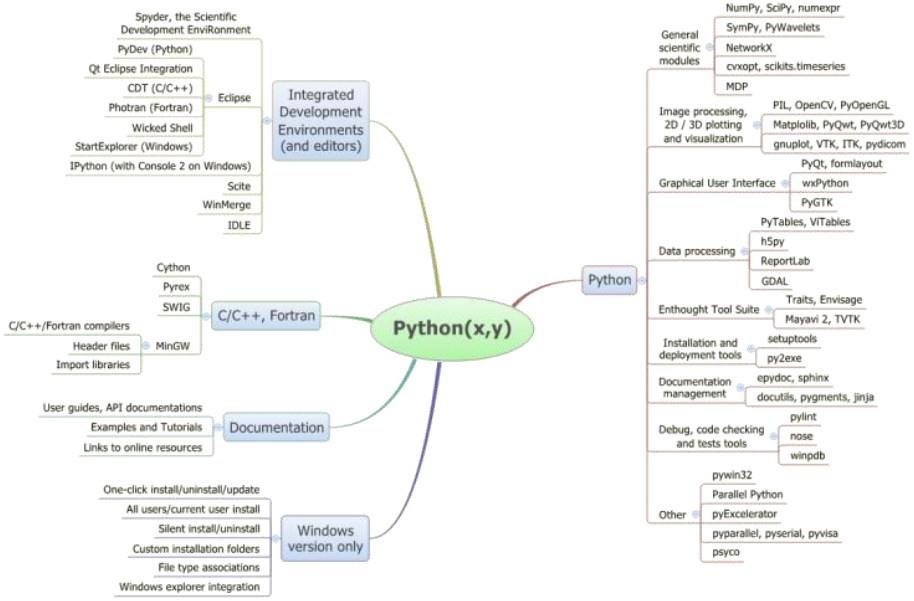
- Data abstraction
- Encapsulation
- Polymorphism
- Inheritance

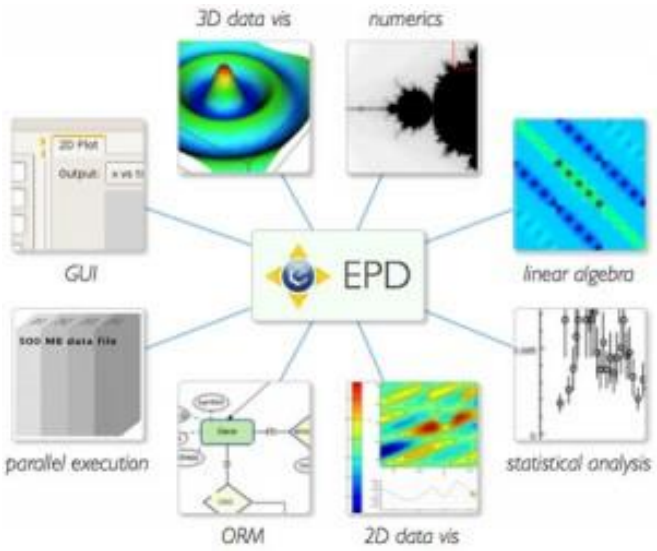
Programming using object is quite easy. During the course we will often program using object (python is OO, VTK library is OO, Qt framework is OO).
Object Oriented Programming is more complex and we don’t need to use it during the course.

Python with bacteries

python(x,y)

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS





Python Syntax Basic

- Python was designed to be a highly readable language.
- Python uses indentation to delimit program blocks: **You must indent your code correctly for it to work!**
- Comments start with '#', until end of line.
- Multi-lines comments are also allowed. You can use triple-quoted string

```
'''
This is a multi-line comment
'''
```

PYTHON DATA TYPES: NUMBER

NUMBER (int,float,complex)

```
>>> type(1)
>>> <type 'int'>
>>> type(1.)
>>> <type 'float'>
>>> type(1 + 0j)
>>> <type 'complex'>
```

Python Data type: strings

Creation

```
# using double quotes
>>> s = “hello world”
>>> print s
hello world
# single quotes also
#work
>>> s = ‘hello world’
>>> print s
hello world
# triple quotes are used
# for mutli-line strings
>>> a = ”””hello
... world”””
>>> print a
hello
world
```

Formatting

```
>>> s = “some
numbers:”
>>> x = 1.34
>>> y = 2
>>> s = “%s %f, %d” %
(s,x,y)
>>> print s
some numbers: 1.34, 2
```

Operation *, +

```
# concatenating two
#strings
>>> “hello “ + “world”
‘hello world’
# repeating a string
>>> “hello “ * 3
‘hello hello hello ’
```

Methods

```
>>> s = “hello world”
>>> s.split()
[‘hello’, ‘world’]
>>> ‘ ‘.join(s.split())
hello world
>>> s.replace(’world’
,’Mars’)
‘hello Mars’
# strip whitespace
>>> s = “\t hello \n”
>>> s.strip()
‘hello’
```

Python Data type: list

Creation

```
>>> l = [10,11,12,13,14]
>>> print l
[10, 11, 12, 13, 14]
#eterogeneuos container
>>>l =
[10,'eleven',[12,13]]
# the range method is helpful
# for creating a sequence
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(2,7)
[2, 3, 4, 5, 6]
>>> range(2,7,2)
[2, 4, 6]
```

Operation *,+

```
#Concatenation
>>> [10, 11] + [12,13]
[10, 11, 12, 13]
#Repetition
>>> [10, 11] * 3
[10, 11, 10, 11, 10, 11]
```

Setting /retrieving an element

```
>>> l = [10,11,12,13,14]
>>> l[0]
 10
>>>l[0]=100
#negative index
>>>l[-1]
14
```

Slicing [start:stop:step]

```
 >>> l = [10,11,12,13,14]
 >>> l[1:3]
 [11, 12]
# negative indices work also
>>> l[1:-2]
[11, 12]
>>> l[-4:3]
[11, 12]
# grab first three elements
>>> l[:3]
[10,11,12]
# grab last two elements
>>> l[-2:]
[13,14]
```

Python Data type: list

```
>>> l = [10,21,23,11,24]
# add an element to the list
>>> l.append(11)
>>> print l
[10,21,23,11,24,11]
# how many 11s are there?
>>> l.count(11)
2
# where does 11 first occur?
>>> l.index(11)
3
# remove the first 11
>>> l.remove(11)
>>> print l
[10,21,23,24,11]
```

```
# sort the list
>>> l.sort()
>>> print l
[10,11,21,23,24]
# reverse the list
>>> l.reverse()
>>> print l
[24,23,21,11,10]
#len of a list
>>> len(l)
5
#deleting an element
>>> del l[2]
>>> l
[24,23,11,10]
# use in or not in
>>> l = [10,11,12,13,14]
>>> 13 in l
1
```

Python Data type: dictionary

Dictionaries store key/value pairs. Indexing a dictionary by a key returns the value associated with it.

Creation

```
# create an empty dictionary using curly brackets
>>> record = {}
>>> record['first'] = 'Jmes'
>>> record['last'] = 'Maxwell'
>>> record['born'] = 1831
>>> print record
{'first': 'Jmes', 'born': 1831, 'last': 'Maxwell'}
# create another dictionary with initial entries
>>> new_record = {'first': 'James', 'middle':'Clerk'}
# now update the first dictionary with values from the new one
>>> record.update(new_record)
>>> print record
{'first': 'James', 'middle': 'Clerk', 'last':'Maxwell', 'born': 1831}
```

Python Data type: dictionary

```
>>> d = {‘cows’: 1,’dogs’: 5, ... ‘cats’: 3}
# create a copy.
>>> dd = d.copy()
>>> print dd
{'dogs':5,'cats':3,'cows': 1}
# test for chickens.
>>> d.has_key(‘chickens’)
0
# get a list of all keys
```

```
>>> d.keys()
[‘cats’,’dogs’,’cows’]
# get a list of all values
>>> d.values()
[3, 5, 1]
# return the key/value pairs
>>> d.items()
[('cats', 3), ('dogs', 5),
 ('cows', 1)]
# clear the dictionary
>>> d.clear()
>>> print d
{}
```

Python Data type: tuple

Tuples are a sequence of objects just like lists. Unlike lists, tuples are immutable objects. While there are some functions and statements that require tuples, they are rare. A good rule of thumb is to use lists whenever you need a generic sequence.

Creation

```
# tuples are built from a comma separated list enclosed by ( )
>>> t = (1,'two')
>>> print t
(1,'two')
>>> t[0]
1
# assignments to tuples fail
>>> t[0] = 2
```

Python Data type: set

- Sets: non ordered, unique items

```
>>> s = set(('a', 'b', 'c', 'a'))
>>> s
>>> set(['a', 'b', 'c'])
>>> s.difference(('a', 'b'))
>>> set(['c'])
Sets cannot be indexed:
>>> s[1]
------------------------------------------------------------
--------
TypeError Traceback (most recent call last)
TypeError: 'set' object does not support indexing
```

Python Data Types

Mutable Objects

```
# Mutable objects, such as
# lists, can be changed
# in-place.
# insert new values into list
>>> l = [10,11,12,13,14]
>>> l[1:3] = [5,6]
>>> print l
[10, 5, 6, 13, 14]
```

Immutable Objects

```
# Immutable objects, such as
# strings, cannot be changed
# in-place.
# try inserting values into
# a string
>>> s = ‘abcde’
>>> s[1:3] = ‘xy’
Traceback (innermost last):
File "<interactive input>",line 1,in ?
TypeError: object doesn't support
slice assignment
```

Python OO

- Everything in Python is an object

```
>>> l=[1,2,3]
>>>dir(l)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getslice__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort`]

#Dot notation to access data/method of a class
>>>l.count(1)
```

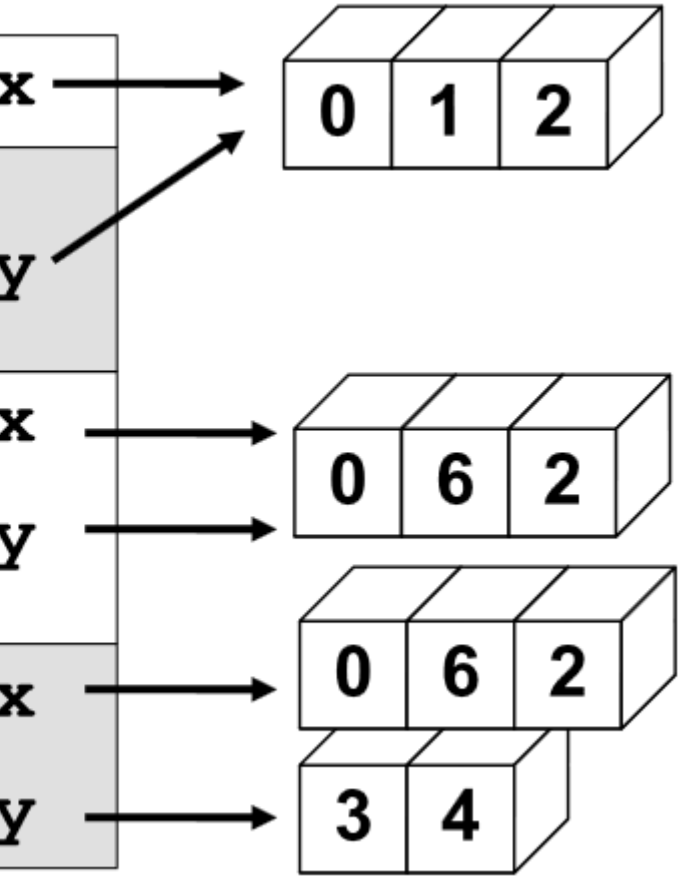
Python Data Types

```
>>> x = [0, 1, 2]
```

```
# y = x cause x and y to point
# at the same list
>>> y = x
```

```
# changes to y also change x
>>> y[1] = 6
>>> print x
[0, 6, 2]
```

```
# re-assigning y to a new list
# decouples the two lists
>>> y = [3, 4]
```



Control flow: if/else

- if/elif/else provide conditional execution of code blocks.

Syntax

```
if <condition>:
 <statements>
elif <condition>:
 <statements>
else:
 <statements>
```

Example

```
# a simple if statement
>>> x = 10
>>> if x > 0:
...         print 1
... elif x == 0:
...         print 0
... else:
...         print -1
...
1
```

Control flow: for loop

if/elif/else provide conditional execution of code blocks.

for <loop_var> in <sequence>:
<statements>

Example

```
>>> for i in range(5):
... print i,
... < hit return >
0 1 2 3 4
```

Loop over a string

```
>>> for i in ‘abcde’:
... print i,
... < hit return >
a b c d e
```

Loop over a list

```
>>>
l=[‘dogs’,’cats’,’bears’]
>>> accum = ‘’
>>> for item in l:
... accum = accum +
item + ' '
>>> print accum
dogs cats bears
```

Control flow: while loop

While loops iterate until a condition is met.

while <condition>:
<statements>

While loop

```
# the condition tested is
# whether lst is empty.
>>> lst = range(3)
>>> while lst:
... print lst
... lst = lst[1:]
[0, 1, 2]
[1, 2]
[2]
```

Breaking out of a loop

```
# breaking from an infinite loop.
>>> i = 0
>>> while 1:
        ... if i < 3:
        ...   print i,
        ... else:
        ...   break
        ... i = i + 1
0 1 2
```

Function

Anatomy of a function

The keyword def indicates the start of a function.

Function arguments are listed separated by commas. They are passed by assignment.

Indentation is used to indicate the contents of the function. It is not optional, but a part of the syntax.

```
def func(arg0,arg1,argN=N) :
    a=arg0+arg1+argN
    return a
```

A colon (:) terminates the function definition.

An optional return statement specifies the value returned from the function. If return is omitted, the function returns the special value None.

Function

```
# We'll create our function
# on the fly in the
# interpreter.
>>> def add(x,y):
... a = x + y
... return a
# test it out with numbers
>>> x = 2
>>> y = 3
>>> add(x,y)
5
```

```
# how about strings?
>>> x = 'foo'
>>> y = 'bar'
>>> add(x,y)
'foobar'
# functions can be
assigned
# to variables
>>> func = add
>>> func(x,y)
'foobar'
```

Function

#Mandatory Parameter

```
>>>def double_it(x):
....: return x * 2
>>> double_it(3)
 6
>>> double_it()
TypeError: double_it() takes exactly 1 argument (0 given)
```

#Optional Parameter

```
>>def double_it(y,x=2):
....: return y+x * 2
>>>double_it(3)
 7
>>> double_it(3,x=1)
3
```

Function

```
#Mutable/immutable parameter
def foo(x, y):
...: x = 23
...: y.append(42)
...: print('x is %d' % x)
...: print('y is %d' % y)
...:
>>>a = 77 # immutable variable
>>> b = [99] # mutable variable
>>> foo(a, b)
x is 23
y is [99, 42]
>>>print a,b
77
[99, 42]
```

Module

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended

```
# ex1.py
PI = 3.1416
def sum(lst):
   tot = lst[0]
   for value in lst[1:]:
        tot = tot + value
   return tot

l = [0,1,2,3]
print sum(l), PI
```

```
# Import a module
# load and execute the module
>>> import ex1
6, 3.1416
# get/set a module variable.
>>> ex1.PI
3.141599999999999
>>> ex1.PI = 3.14159
>>> ex1.PI
3.141589999999999
# call a module variable.
>>> t = [2,3,4]
>>> ex1.sum(t)
9
```

```
#Executing as a script
[ej@bull ej]$ python
ex1.py
6, 3.1416
```

Module

- How to import a module.

```
from os import *
from os import path
from os import path as PP
import os
import os as O
```

Current namespace is modified depending on how you import a module:

```
>>>import math
>>>math.sin(math.pi)
```

```
>>>from  math import *
>>>sin(pi)
```

Reading a File

How to Open a File?

f=open(name[, mode[, buffering]])
-> file object

How To Read a File?

f.read(size)
f.readlines(size)
f.readline(size)

```
#rcs.txt
#freq (MHz) vv (dB) hh (dB)
100 -20.3 -31.2
200 -22.7 -33.6

>>> results = []
>>> f = open('rcs.txt','r')
# read lines and discard header
>>> lines = f.readlines()[1:]
>>> f.close()
>>> for l in lines:
# split line into fields
    fields = line.split()
# convert text to numbers
    freq = float(fields[0])
    vv = float(fields[1])
    hh = float(fields[2])
# group & append to results
    all = [freq,vv,hh]
    results.append(all)
```

Writing to a file

How to write to a file?

f.write(string)

f.writelines(sequence of string)

How to close a file?

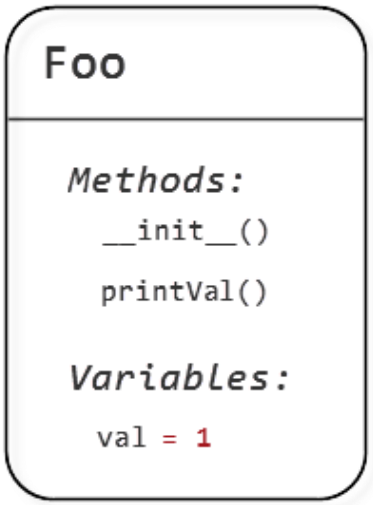
f.close()

```
>>> f = open('rcs.txt','w')
>>> f.write('hello\n')
>>> f.write('world!\n')
>>> for el in xrange(10):
            f.write(str(el)+'\n')
>>>f.writelines(['Str1','Str2','Str3'])
>>>f.close()
```

OOP in Python

- *Classes* are a way of grouping related data (attributes) together into a single unit (also known as an *object*), along with functions that can be called to manipulate that object (also known as *methods*).
- Defining a Python Class:

```
class Foo:
    def __init__(self, val):
        self.val = val
    def printVal(self):
        print(self.val)
f=Foo(3)  #f is an instance of Foo class
```



OOP in Python

- Both methods defined in the previous example take a parameter called self. Also attribute are defined with self. prepended their name.
- Because we can create many instances of a class, when a class method is called, it needs to know which instance it is working with, and that's what Python will pass in via the self parameter.

```
f=Foo(3)            #instance of Foo
d=Foo(4)            #instance of Foo
d.printVal          #call printVal on d instance
f.printVal          #call printVal on f instance
```

You access the object's attributes using the dot operator with object.

OOP in Python

- _init_ is a special method that is called whenever Python creates a new instance of the class. It works like a constructor of the class.

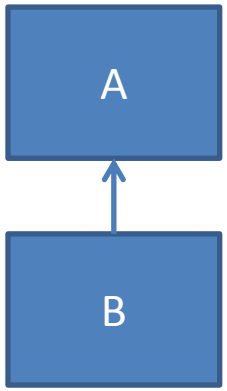
- Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute:
  - **__dict__** : Dictionary containing the class's namespace.
  - **__doc__** : Class documentation string, or None if undefined.
  - **__name__**: Class name.
  - **__module__**: Module name in which the class is defined. This attribute is "__main__" in interactive mode.
  - **__bases__** : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

OOP in Python

Class Inheritance

- You can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.
- The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class.

```
class Base:
    def __init__(self):
        self.x=10
class Derivate(Base):
    def __init__(self):
        Base.__init__(self)
        self.y=20
```



OOP in Python

```
class Parent: # define parent class
    def __init__(self):
        print "Calling parent constructor"
        self.parent=100
    def parentMethod(self):
        print 'Calling parent method'

class Child(Parent): # define child class
    def __init__(self):
        Parent.__init__(self)
        self.child=300
        print "Calling child constructor"
    def childMethod(self):
        print 'Calling child method'
```

OOP in Python

```
c = Child() # instance of child
print dir(c)
c.childMethod() # child calls its method
c.parentMethod() # calls parent's method
```

OUTPUT

```
Calling parent constructor
Calling child constructor
['__doc__', '__init__', '__module__', 'child',
'childMethod', 'getAttr', 'parent', 'parentMethod',
'setAttr']
Calling child method
Calling parent method
```

Operating system funcionalities

- os: *"A portable way of using operating system dependent functionality."*

```
>>> import os
>>> os.getcwd()
'/ccc/cont005/dsku/jaspe/home/user/ra1147/invernia‘
>>> import os
>>> os.getcwd()
'/ccc/cont005/dsku/jaspe/home/user/ra1147/invernia‘
>>> os.mkdir('test')
>>> os.rmdir('test')
>>> os.system('ls')
OpenFOAM  actuatorLine_v4.1_1.7.1
actuatorLine_v4.1_1.7.1.tar  epd-7.2-1-rh5-x86_64.sh  exe
prova_dev  scripts
0
```

Operating system funcionalities

Path manipulation with `os.path`

```
>>> fp=open('file.txt','w')
>>> fp.close()
>>> os.path.abspath('file.txt')
'/ccc/cont005/dsku/jaspe/home/user/ra1147/invernia/file.txt'
>>> os.path.join(os.getcwd(),'file.txt')
'/ccc/cont005/dsku/jaspe/home/user/ra1147/invernia/file.txt‘
>>> os.path.isfile('file.txt')
True
>>> os.path.isdir('file.txt')
False
```

Get parameter from standard input

- raw_input

```
import sys
if __name__ == '__main__':
  while(True):
        print 'PLEASE INSERT AN INTEGER NUMBER IN THE RANGE 0-10'
        param1 = raw_input()
        if int(param1) in range(11):
                        while(True):
                                        print 'PLEASE INSERT A CHAR PARAMETER IN [A,B,C]'
                                        param2 = raw_input()
                                        if param2 in ['A','B','C']:
                                                        print ,param1, param2
                                                        sys.exit()
                                        else: print 'TRY AGAIN PLEASE'
        else: print 'TRY AGAIN PLEASE'
```

Get parameter from standard input

- **input()**

```
import sys
if __name__ == '__main__':
  while(True):
      print 'PLEASE INSERT AN INTEGER NUMBER IN THE RANGE 0-10'
      param1 = input() #notare che c'è eval e int() non va messo
      if param1 in range(11):
            while(True):
                  print 'PLEASE INSERT A CHAR PARAMETER IN [A,B,C]'
                  param2 = input() # causa eval bisogna usare il simbolo di carattere
                  if param2 in ['A','B','C']:
                        print 'uso I due parametri passati dall utente: ',param1, param2
                        sys.exit()
                  else: print 'TRY AGAIN PLEASE'
      else: print 'TRY AGAIN PLEASE'
```

How to launch a script

- We can launch a python script in different ways:

  1) [user@prompt] python myscript.py

  2) [user@prompt] ./myscript.py

  This imply that a shebang line in inserted at the top of your script file

  #myscripy.py

  !#/usr/bin/python

  import os

  import math

  3) From Ipython shell

  In [4]: run myscript.py

How to launch a script

- How to pass input parameter to a python script? sys.argv

```
# script che prende 2 parametri di input
import sys
usage="""necessita di due parametri di input (param1, param2)
correct usage: python script.py param1 param2"""

if __name__ == '__main__':
  if len(sys.argv) < 2:
    print 'Io script: ',sys.argv[0],usage
    sys.exit(0) # termina dopo aver stampato la stringa di usage
  param1 = sys.argv[1]
  param2 = sys.argv[2]
  print 'uso I due parametri passati al momento dell'invocazione dello script:\ ',param1,
param2
```

Python Introspection

- Introspection refers to the ability to examine something to determine what it is, what it knows, and what it is capable of doing. Introspection gives programmers a great deal of flexibility and control. Python's support for introspection runs deep and wide throughout the language.

```
>>>l=[1,2,3]
>>>dir(l)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
>>>help(l.sort)
Help on built-in function sort:
sort(...)
    L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
    cmp(x, y) -> -1, 0, 1
>>>a=5
>>>type(a)
<type'int'>
```