A Crash Course on PYTHON Programming

I. Arregui, A. M. Ferreiro, J. A. García & Á. Leitao

Departamento de Matemáticas, Universidad de La Coruña

July, 2019

https://sites.google.com/site/crashcourseonpython/

Index

Python

- It is the most used programming language in Google

Python

- It is the most used programming language in Google
- Declared by TIOBE language of 2007, 2010 and 2018: `http://www.tiobe.com/`

Python

- It is the most used programming language in Google
- Declared by TIOBE language of 2007, 2010 and 2018: `http://www.tiobe.com/`
- Fourth most popular programming language (May, 2019): `http://www.tiobe.com/`

Introduction to Python

Variables and data types

Python programming

Object–oriented programming

NumPy: Numerical Python

SciPy: Scientific Python

Bidimensional graphics with MatPlotLib

Pandas

Miscelanea

Some features of PYTHON

- PYTHON is a general purpose programming language, designed by Guido van Rossum at the end of the 80's
- Clear and structured programming (for example, the tabulator is part of the language)
- Great productivity, high development velocity
- Multiple programming paradigmes: object oriented, structured, functional, ...
- Portable (Linux, Windows, Mac OSX)
- Interpreted, dynamic, strongly typed, automatic memory management
- Easy to learn
- Large *standard library*: `http://docs.python.org/library/`
- Easy to extend: link with C/C++ (`SWIG, Weave, CPython`), .NET (`IronPython`), CORBA, Java (`Jython`), FORTRAN (`f2py`), ...
- Large number of available packages

Python

Some **applications**:

- web development (Zope, Plone, Django, webpy, TurboGears, Pylons, e-mail, RSS, ...)
- access to databases (pyodbc, mysqldb, ...)
- graphical user interfaces (Tk/Tcl, WxWidgets, Qt, FLTK, Gtk, ...)
- games (PyGame, PyKyra)
- network applications: client–server (Twisted Python), ...
- graphical representation: 2D (MatPlotLib, Chaco), 3D (`VTK`, `MayaVi`), ...
- scientific computing (NumPy, SciPy)
- XML processing, HTML, ...

It works on Linux / Unix platforms, Windows, Mac, JVM (Jython), ... and there are implementations on the Nokia 60 series!

Where is Python used?

- in Google, where it is one of the three development languages (with C / C++ and Java)
- in YouTube
- in BitTorrent
- in animation: DreamWorks Animation, Pixar, Industrial Light & Magic
- the RedHat / FEdora installer (Anaconda) is written in Python
- RocksCluster is a Linux distribution for clustering, implemented on CentOSLinux, that uses Python scripts for node and users management, among others ...
- Los Alamos National Laboratory and Sandia National Laboratories have developed PyTrilinos, ParaView, ...
- Salome and Abaqus (CAD/CAE) and FEniCS (finite elements) use Python as standard script language

Some success stories: `http://www.python.org/about/success/`

Why is Python so extended?

- It is quite easy to develop **wrappers** which allow using almost every software written in C / C++ and Fortran
  - by means of the PythonC API
  - by automatic generators: SWIG, SIP, Weave, f2py

  Python is very used as a *glue*

- Almost any free software library has its corresponding wrapper, so that it can be used from Python

- Documentation is very complete
  - in the console, through function `help`
  - in the different projects webs

- Python community is very active
  - SciPy Conference, once per year in USA and Europe
  - PyCon (in USA) and EuroPython, annual conference
  - in SIAM Annual Meeting, an special session is often dedicated to the use of Python in scientific computing

Editors and command windows

Editors:

- `gedit`, `vi`, `emacs`, ...

- `eric` (`http://eric-ide.python-projects.org/`)
  both of them include directory manager, help manager, debugger,...
- `spyder`
- `Eclipse` + plugin for PYTHON
  (`http://www.eclipse.org/` and `http://pydev.org/`)

Be careful with tabulators !

- the tabulator is part of the language sintax; PYTHON uses the indentation to delimit code blocks (loops, if–loops, functions, ...)
- the standard tabulator has 4 spaces; all tabulators in the same file must be defined in the same way

Command windows (consoles):

```
$ python
$ ipython
```

The second one is more developed

Variables

A **variable** is a space of memory in which we store a value or set of values; each variable is given a name and can have different types (scalar, list, array, dictionary, string, ...)

PYTHON is a dynamically typed language: variables can take **different values**, **of different types**, in different moments

- the result of an expression is assigned to a variable by the assignement operator (=); e.g.,

  ```
  >>> 5 + 2; b = 3 * 4
  ```

  computes de addition (not assigned to any variable) and makes `b` take the value 12
- to continue an expression in the following line, we use the backslash (\)
- we are not obliged to initially declare the variables
- PYTHON distinguishes between capital and lower case characters
- the names of variables start by an alphabetic character

Variables and workspace

A variable can be destroyed:

```
>>> del x
```

The `type` command is used to know the type of a variable:

```
>>> type (a)
```

The workspace is the set of variables and user functions in the memory

In the I–Python shell,

- the `who` command is used to examine all the variables existing in the current instant and the imported packages
- `whos` gives some details on each variable

Data types: logical variables

They can take two values: `True` or `False`.

```
>>> m = True
>>> a = 0; b = -4;          # a and b are integer variables
>>> type (a); type (b)
>>> a = 0.; b = -4.;        # a and b are real variables
>>> type (a); type (b)

>>> xa = bool (a)   # xa is a logical variable, which value is False
>>> xb = bool (b)   # xb is a logical varialbe, which value es True

>>> c1 = (b == True)      # The result is False
                          # because b is not a logical variable
>>> c2 = (xb == False)      # The result is False
                            # because xb takes the value True
>>> c3 = (xb = False)      # We get an execution error
```

Data types: the `None` variable

Designes an empty object.

For example

```
>>> a = None
```

we create variable `a`, which contains nothing and can later be of any type (numeric, string, class object, ...); to check it, we can do:

```
>>> a is None
```

We can also create an empty list, dictionary or tuple by:

```
>>> a1 = []; a2 = { }; a3 = ()
```

```
>>> a        # Whe check the value of a
             # We get no answer because a=None
>>> a = 4.5;      # We assigne a real value
```

Data types: numerical variables

They can be:

- integer (between -2147483648 and 2147483647): `0`, `283`, `-5`
- `long`: `>>> i = 22L`
- double precission, real: `0.`, `-2.45`, `1.084E+14`
- double precission, complex: `a = 1.-4j`, `b = complex(7,-2)`

where `j` is the imaginary unit ($j = \sqrt{-1}$).

We can get the real and imaginary parts of a complex:

```
>>> b.real
>>> b.imag
```

When operating two numerical variables, the result takes the type of the "highest cathegory" operator

Data types: strings

They are enclosed in single or double quotes

```
>>> s = 'Hello, world!'
>>> t = 'And he said:  "Hello, everybody!"'
>>> print s
```

A short example, with specific functions:

```
from string import *
s1 = 'First string'
s2 = 'Second string'
s3 = "Third string"
print ' s1 = ', s1
print ' s2 = ', s2
print ' s3 = ', s3
```

```
s01 = s1 + s2 + s3;  print s01
s02 = s1 + ' ' + s2 + ' ' + s3
print s02
print split (s01, sep=' ')
print count (s01,'st')
print capitalize (s02)
print lower (s01)
print replace (s02,'t','p')
```

Data types

```
>>> a = 'Welcome message'
>>> b = 4; c = 130L; d = 4.56789; m = 4.2-8.424j

>>> cs = str (c)      # '130'
>>> ds = str (d)      # '4.56789'
>>> ms = str (m)      # '(4.2-8.424j)'

>>> bd = float (b)      # 4.0
                  #  can be applied to strings of numerical characters,
                  #  integers and complexes with no imaginary part

>>> di = int (d)      # 4
                  #  returns the nearest integer towards zero;
                  #  can be applied to strings of numerical characters,
                  #  and complexes with no imaginary part
>>> bc = complex (b)      # 4+0j
>>> dc = complex (d)      # 4.56789+0j
```

Tuples and lists

Tuples and **lists** allow the storing of an arbitrary sequence of objects

The stored objects may be of different types: string, numbers, other lists or tuples, ...

We access each element by an index, which indicates the position of the element in the list or tuple

- indexes always start in *zero*

Difference between tuples and lists:

- A tuple can never be modified
- In a list, we can change, add and delete elements

Tuples

A tuple is an arbitrary sequence of objects, enclosed in **parentheses** and separated by *commas*, ( )

The numbering of indices starts at **cero**

```
>>> x = ()                  # Empty tuple
>>> x = (2,)                # Tuple of one only element
>>> x = (1, 4.3, 'hello', (-1,-2))
>>> x[0]                    # We get the first component
1
>>> x[3], x[3][1]
(-1, -2), -2
>>> x[-2]                   # Starting from the end
'hello'
>>> y = 2,3,4               # Parentheses can be omitted
>>> z = 2,                  # Tuple of an only element
```

`len (x)`: length of a tuple

`max (x)`, `min (x)`: maximum/minimum of a tuple

`tuple (seq)`: transforms `seq` in a tuple

```
>>> tuple ('hello')
('h','e','l','l','o')
```

Lists

A list is an arbitrary sequence of objects, enclosed in **brackets** and separated by *commas*, [ ]
The numbering of indices starts at **cero**

```
>>> x = []          # Empty list
>>> x = [1,4.3,'hello',[-1,-2],'finger',math.sin,[-0.4,-23.,45]]
>>> x[0]
1
>>> x [3], x [3][0]
[-1, -2], -1
>>> x[-3]           # Starting by the end
'finger'
>>> x [2:4]         # Returns a (sub)list: [x[2], x[3]]
['hello', [-1,-2]]
>>> x[4:]
['finger', <built-in function sin>, [-0.4,-23,45]]
>>> x[7]
Traceback (most recent call last):
File "<input>", line 1, in ?
IndexError:  list index out of range
```

Useful list functions

`range ((first,)last(,step))`: creates a list of integers, since `first` until `last` (*not included!*) with step `step`.

- if `first` is not given, it starts in *zero*
- if `step` is not given, the step is *one*

```
>>> range (-1, 5, 1)
[-1, 0, 1, 2, 3, 4]
>>> range (-10, 10, 2)
[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8]
```

`len(x)`, `max(x)`, `min(x)`

```
>>> x = [-120, 34, 1, 0, -3]
>>> len (x)      # Number of elements of x
5
>>> max (x)
34
>>> min (x)
-120
```

Managing lists

`map(func,xlist)`: applies function `func` to every element of `xlist`

```
>>> x = ['Felix','Graziana','ROBERTA','luIS']
>>> map (string.lower,x)      # writes all strings in lower case
['felix','graziana','roberta','luis']
```

The following methods can be applied to any object of type `list`:

- `append(obj)`: adds an element `(obj)` at the end of the list
- `extend(obj)`: adds the elements of list `obj` to the current list

```
>>> z = [1,2]
>>> z.append (['dog','cat'])
>>> z
[1,2,['dog','cat']]
```

```
>>> z.extend ([20,'hello',-40])
>>> z
[1,2,['dog','cat'],20,'hello',-40]
>>> z + [20,'hello',-40]
```

Managing lists

`x.insert(j,obj)`: inserts an element `obj` in position `j` of the list
`x.remove(obj)`: finds the first element coincident with con `(obj)` and removes it from the list
`x.pop(j)`: if an index (`j`) is given, it removes the element in its position; if not, it removes the last element
`x.reverse()`: writes the list in inverse order
`x.sort()`: sorts the list

```
>>> x = [1,'hello',[2,3.4],'good',[-1,0,1],'bye']
>>> x.remove ('hello')
>>> x
[1,[2,3.4],'good',[-1,0,1],'bye']
>>> x.pop (1)
[2,3.4]
>>> x
[1,'good',[-1,0,1],'bye']
```

```
>>> x.insert (2,'text')
>>> x
[1,'good','text',[-1,0,1],'bye']
>>> x.pop ()
'bye'
>>> x
[1,'good','text',[-1,0,1]]
```

Managing lists

We can create a list with all elements equal:

```
>>> n = 5
>>> x = 0.25
>>> a = [x]*n
>>> a
[0.25, 0.25, 0.25, 0.25, 0.25]
```

A string can be considered as a list:

```
>>> s = 'Hello, world'
>>> s[0]
'H'
>>> s[1]
'e'
>>> s[-1]
'd'
```

We can convert an object into a list: `>>> list(obj)`

```
>>> x=(1,2,3); y=list(x)
```

Exercise 1

- We create two empty lists to store names and identity card numbers:

```
>>> names = []
>>> idnumbers = []
```

- We add data to each list:

```
>>> idnumbers.append ('33807659D');
>>> idnumbers.append ('32233322K')
>>> names.append ('Franz Schubert')
>>> names.append (r'Claudio Monteverdi')
```

- We create a list with the two previous lists:

```
>>> data = [idnumbers, names]
```

- We access the data:

```
>>> data[0]
>>> data[1]
>>> data[0][1]
```

Exercise 2

Create a list with the first terms of Fibonacci sequence

```
>>> a = [0, 1]
>>> a.append (a[0] + a[1])
>>> a
>>> a.append (a[1] + a[2])
>>> a.append (a[2] + a[3])
>>> a.append (a[3] + a[4])

>>> a.append (a[-2] + a[-1])
>>> a.append (a[-2] + a[-1])
>>> a.append (a[-2] + a[-1])
>>> a.append (a[-2] + a[-1])
```

Exercise 3

For a 2–D mesh, we want to create:

- a list of nodes, with the two coordinates of each
- a list of elements, with the indices of the vertices

- Coordinates:

```
>>> nodes = [[0., 0.], [1., 0.], [1., 1.], \
        [0., 1.], [2., 0.], [2., 1.]]
>>> nodes [0]     # [0., 0.]
>>> nodes [2]     # [1., 1.]
>>> nodes [5]     # [2., 1.]
```

- Elements:

```
>>> conec = [[0, 1, 2], [0, 2, 3], [1, 4, 5], [1, 5, 2]]
>>> conec [2]
>>> conec [3]
>>> nodes [conec[0][2]]     # [1., 1.]
```

Exercise 3

Another possibility:

- Coordinates:

```
>>> nodes = [(0., 0.), (1., 0.), (1., 1.), \
        (0., 1.), (2., 0.), (2., 1.)]
>>> nodes [0]     # (0., 0.)
>>> nodes [2]     # (1., 1.)
>>> nodes [5]     # (2., 1.)
```

- Elements:

```
>>> conec = [[0, 1, 2], [0, 2, 3], [1, 4, 5], [1, 5, 2]]
>>> conec [2]
>>> conec [3]
>>> nodes [conec[0][2]]     # (1., 1.)
```

Dictionaries

A dictionary allows to store an arbitrary sequence of objects

- We access its elements through **keys**, which are variables of any type (not only integers)
- The elements are sorted by keys alphabetical order

Exemple:

```
>>> x = { key01:  'first element' }
```

or:

```
>>> x = { }        # empty dictionary
>>> x [key01] = 'first element'
>>> x [key02] = 25.50


>>> x
{ key01:  'first element', key02:  25.5 }
```

Dictionaries

Other example:

```
>>> clients = {'smith':  ['Adam Smith',38,'44000111D'],
              'roberts':  ['Mary Roberts',17,'33221998L']}
>>> clients['roberts']
['Mary Roberts',17,'33221998L']
```

How to add, modify and delete elements?

```
>>> clients ['white'] = ['Charles White',23,'44555111L'] # add
>>> clients ['smith'] = ['Adam Smith',29,'44000112D'] # modify
>>> del clients ['roberts']
>>> clients
```

`>>> namedict.has_key (namekey)`: returns `True` if the dictionary `namedict` has the key `namekey`; otherwise, it returns `False`

Dictionaries

- `len (ndic)`: number of elements of the dictionary `ndic`
- `ndic.keys ()`: returns a list with the keys
- `ndic.values ()`: returns a list with the values
- `ndic.items ()`: returns the contents in tuples
- `ndic.update (ndic2)`: adds the elements of one dict. to another
- `ndic.clear ()`: deletes all the elements of ndic

For example,

```
>>> clients = {'wagner':['R. Wagner',19],
           'byrd':['W. Byrd',45], 'white':['C. White',23]}
>>> clients.keys()
['wagner','byrd','white']
>>> len (clients)
3
>>> clients.clear()
>>> clients
{ }
```

Exercise

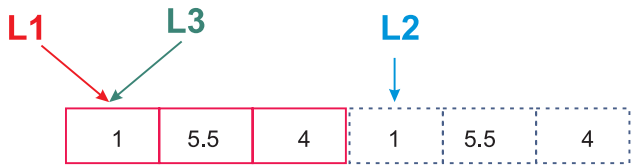
In the previous 2–D mesh, make a dictionary of the elements

```
>>> elem = {  }
>>> elem[0] = [0, 1, 2]
>>> elem[1] = [0, 2, 3]
>>> elem[2] = [1, 4, 5]
>>> elem[3] = [1, 5, 2]

>>> elem
>>> elem.keys ()
>>> elem.values ()
>>> elem.items ()
```

Value copy

A value copy of a list/tuple/dictionary is a different object with the same content than the original one; the new object points to **a different memory position**

```
>>> L1 = [1,  5.5,  4]
>>> L2 = L1[:]          # A copy of the list
>>> L2 is L1
False
>>> L2.append (100)     # If we modify L2, then L1 doesn't change
```



Another way of doing the same process is by means of the `copy` function:

```
>>> import copy
>>> L1 = {'a':1,  'b':22,  'c':4}
>>> L2 = copy.copy (L1)
>>> L2 is L1
False
>>> L1['d'] = 88     # If we modify L1, then L2 is not affected
```

Reference copy

By a reference copy, we create an object that **points to the same memory position** of the original object

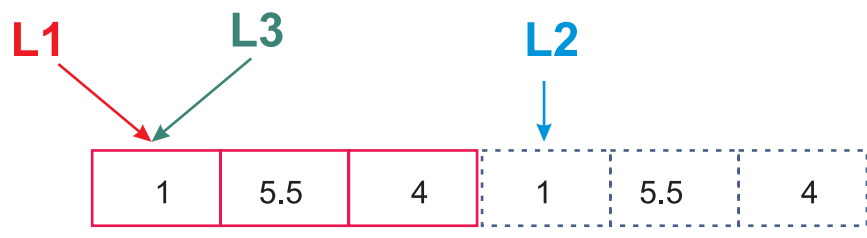
```
>>> L1 = [1, 5.5, 4]
>>> L3 = L1      # Creates a new variable, that points to L1
>>> L3 is L1
True
```

If we modify `L3`, then we also modify `L1`:

```
>>> L3.append (100)
>>> L1
[1,  5.5,  4,  100]
```

If we delete `L1`, then `L3` is not deleted:

```
>>> del L1;   L1
Traceback (most recent call last):
File "<input>", line 1, in ?
NameError:  name 'L1' is not defined
>>> L3
[1,  5.5,  4,  100]
```



Introduction to Python

Variables and data types

Python programming

- Control sentences
- Functions
- Modules
- The standard library
- Input / output
- Exceptions

Object–oriented programming

NumPy: Numerical Python

SciPy: Scientific Python

Bidimensional graphics with MatPlotLib

PYTHON programming

Names of the files: `namefile.py`

They contain sequences of commands and orders

Script

- they are executed by typing:

```
$ python namefile.py
```

or

```
In[1] run namefile.py
```

Functions

- its structure is:

```
def namefunc (x1,x2,...):
        ⋮
    return [a,b,c]
```

- they are executed by a call from a script, another function or the command window (PYTHON or I–PYTHON shell)

They are recursive

Compiling and linking are not necessary

Bifurcations (if)

```
if condition:
    sentence(s)
```

```
if condition1:
    sentence(s)
elif condition2:
    sentence(s)
```

```
if condition1:
    sentence(s)
elif condition2:
    sentence(s)
elif ... :
    ...
else:
    sentence(s)
```

- Conditions can be boolean, scalar, vectorial or matricial; in this case, vectors and matrices must have equal dimensions
- They can be imbricated
- Only a block of sentences is executed
- Operators:

  `<` `<=` `>` `>=` `==` `!=` `or` `and` `not`

- **IMPORTANT:** Indentation must be done with the tabulator !!!

Bifurcations

The result of each *condition* will be `True` or `False`.

In particular, if we do

```
if var
```

where `var` contains a variable, the result will be:

- `False`, if `var` is:
  - the `None` variable
  - a numerical variable, with value *zero*
  - a boolean variable, with value `False`
  - an empty tuple, list or dictionary
- `True`, otherwise

Loops

```
for i in range (i1,i2,step):
    sentence(s)
```

```
while (condition):
    sentence(s)
```

- if `step` is *one*, it can be omitted
- if `i1` is *zero* and the step is *one*, they both can be omitted

```
for item in ObjIt:
    sentence(s)
```

where `ObjIt` is an iterable object

Indentation must be done with the tabulator!

```
1 import math
2 methods = [math.sin, math.cos, math.tan, math.exp]
3 for k in methods:
4     print k.__name__, '(pi)= ', k (math.pi)    # Python 2.x
5     # print (k.__name__, '(pi)= ', k (math.pi)) # Python 3.x
```

Loops

We can use several lists simultaneously:

```
for x, y, z in zip (xlist,ylist,zlist):
    print x, y, z                    # Python 2.x
    # print (x, y, z)                # Python 3.x
```

- In this case, the number of iterations will be the dimension of the shortest list

Loops can be imbricated:

```
for i in range (i1,i2,step1):
    for j in range (j1,j2,step2):
        print a[i][j]                # Python x.x
        # print (a[i][j])            # Python 3.x
```

Two important commands:

`break`: quits the inner loop

`continue`: starts a new iteration of the same loop

Exercises

Write a PYTHON code ("`days.py`") which read a date (day, month and year), check if it is correct and write it in the screen.

```
import types
y = input ('Write the year:     ')
if (y <= 0 or type (y) != types.IntType):
    print 'The year is not correct ... '       # Python 2.x
    # print ('The year is not correct ... ')  # Python 3.x
```

or:

```
import types
y = -1
while (y <= 0 or type (y) != types.IntType):
    y = input ('Write the year:     ')
print 'The date is:   \%2d.\%2d.\%4d'  \% (d,m,y)
# print ('The date is:   \%2d.\%2d.\%4d'  \% (d,m,y))
```

Exercises

In the previous code, compute the sum of the days of the months previous to the current one.

```
aux = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
total = 0
for i in range (0,m-1):
    total = total + aux (i)
```

or:

```
aux = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
total = sum (aux(0:m-1))
```

Exercises

In the previous code, determine if the current year is a leap–year and modify the code to get the correct result.

```
b = 0
if (y % 4 == 0):
    b = 1
    if (y % 100 == 0 and y % 400 != 0):
        b = 0
aux[1] += b          # aux[1] = aux[1] + b
```

Functions

Their structure is:

```
def f (x1,x2,x3):
    ...
    y1 = ...; y2 = ...;
    return [y1, y2]
```

They can return any type of variable: string, scalar, list, tuple, array, class object, class instance, ...

They admit a variable number of arguments

```
y = f (a, b, c, d, p, q, r, s)
def f (x1, x2, x3, *x4):
    if len (x4) == 4:
```

Default arguments are allowed:

```
def f (x1, x2, x3=1.0, x4=None):
```

Functions

We can give, as argument of a function, the name of another function

```
import math as mt
def f (x):
  return x**2

def g (x):
  return x**3

def h (x):
  return mt.sqrt (abs(x))

def func (x,f):
  map (f,x)
```

```
a = range (-5,6)
opt = 1
if opt == 1:
  print func (a,f)
elif opt == 2:
  print func (a,g)
else:
  print func (a,h)
```

Interactive I/O

Data reading by keyboard in PYTHON 3.X:

```
x = input ('message')
```

'message' is written in the screen and waits until a data is provided; it is stored in x variable as a string.

We cannot operate with x

```
>>> x = input ('Introduce a number:  ')
124.5
```

y = x**2 **is not possible**, because x='124.5'

We should do: y = float(x)**2

Exceptions

We can distinguish between two kinds of errors:

- Syntax errors

```
>>> while True print ('hello')
File "<stdin>", line 1
   while True print ('hello')
                    ^
SyntaxError: invalid syntax
```

  In this example, the colon (":") is missing

- Exceptions

```
>>> 5 / 0
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Exceptions can be captured:

```
try:
   sentences
except [TypeException]:
   sentences
```

Exceptions (example I)

```
>>> 3./ 0.
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError:  float division
```

Exceptions (example I)

```
>>> 3./ 0.
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError:  float division
```

Control of the error:

```
a = 5.0
b = 0.0
try:
    res = a / b
except ZeroDivisionError:
    print ('Division by zero')
```

Exceptions (example II)

```
>>> f = open ('myfile.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IOError:  [Errno 2] No such file or directory:  'myfile.txt'
```

Exceptions (example II)

```
>>> f = open ('myfile.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IOError:  [Errno 2] No such file or directory:  'myfile.txt'
```

Control of the error:

```
import sys
try:
    f = open ('myfile.txt')
    s = f.readline ()
    i = int (s.strip ())
except IOError,(errno,strerror):
    print ("I/O error (%s):  %s"  % (errno,strerror))
except ValueError:
    print ("The data cannot be converted to integer")
except:
    print ("Error:", sys.exc_info()[0])
```

Exercises

1. Write a PYTHON code that:
   (a) reads the components of a vector
      - from the screen
      - from a file
   (b) computes its harmonique average α:

$$\alpha^{-1} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{a_i}$$

   (c) computes its norms:

$$\|a\|_2 = \left(\sum_{i=1}^{n}|a_i|^2\right)^{1/2}$$

$$\|a\|_\infty = \max_{i=1,\ldots,n}|a_i|$$

Exercises

2. Solve the equation:

$$f(x) = x^5 - 3.8x^3 - 4 = 0 \qquad x \in (-2, 2.5)$$

by the bissection method: a_0, b_0 given,

$$x_k = \frac{1}{2}(a_k + b_k)$$

$$\text{si} \;\; f(x_k)f(a_k) < 0 \quad \Longrightarrow \quad \begin{cases} a_{k+1} = a_k \\ b_{k+1} = x_k \end{cases}$$

$$\text{si} \;\; f(x_k)f(b_k) < 0 \quad \Longrightarrow \quad \begin{cases} a_{k+1} = x_k \\ b_{k+1} = b_k \end{cases}$$

Exercises

3. Consider the (convergent) numerical series: $S = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{\sqrt{1+4n}}$.

   Write a PYTHON code that computes its sum with an error less than a given parameter.

4. Write a PYTHON code that reads a date (day, month, year), check if it is a correct date and computes the days since January 1st of that year.

5. Compute the age of a person in days

   Which are leap–years? Multiples of 4, unless multiples of 100 which are not multiples of 400

Exercises

6. Write a PYTHON code that generates a dictionary with:
   - titles and production year of films
   - names of directors
   - names of actors

   and show the information of the films sorted by:
   - key
   - date
   - director name

Object–oriented programming

Object–Oriented Programming (OOP) is a programming paradigm that tries to represent entities (objects) grouping data and methods that describe their features and behavior

In OPP, programs are organized "as in the real life", by means of classes and objects that interact with them
For example, let us consider the points $(1, 2)$, $(3, 4)$ y $(5, 7)$

- all of them are different
- but they have all two coordinates, with different values
- and we can apply the same methods (functions) to all of them

OOP advantages

- The code is much simpler to write, maintain, re-use and extend
- Creating more complex codes is easier
- The implementation of the codes is related to the "organization of the real world"

Object–oriented programming

- A **class** is a sort of *template*, where attributes and methods are defined
- Classes are not manipulated directly; they are used to define new types (**instancias**) of the class or **objects**

  For example, a class could be the plan of a house; it is generic. The instances are the different houses that can be built from the plan
- A class **method** is a function that can be applied to any instance of the class. A class **attribute** is a variable associated to each instance (a property)
- A method and/or attribute is **public** if it is accesible at any point of the code, and it is private if it is only accesible by the methods of the class

Object–oriented programming

OOP is posible in Python

- Creation of a class: `class ClassName:`
- A class has an init method, which is used to create instances of the class

```
class ClassName:
        def __init__ (self,args):
        ...
```

- By convention, the first argument of a class method is `self`, which is a pointer to the class; it is equivalent to `this` in C++ or Java
  - It is not a reserved word, but it is used by convention
- Although `self` must be the first argument of a class method, we do not need to include it as Python invokes it automatically

Instances of a class

- For creating an object, or instance, of a class we have just to invoke the class as if it was a function, giving it the arguments of the `__init__` method

- The returned value is the created instance

```
1 class point:
2     def __init__ (self, x, y):
3         self.x = x            # attribute
4         self.y = y            # attribute
5
6 p1 = point (0.0, 1.0)
7 p2 = point (3.0, 5.0)
8 print ('p1 is p2: ', p1 is p2)
```

Operator overloading

- PYTHON does not allow the method overloading (C++ and JAVA do)
- But operator overloading is allowed: we can redefine operators as $+$, $-$, $*$, $/$, `print`, for example

Method	Overload	How to call it
`__init__`	Initializer	`A = ClassName ()`
`__del__`	Destructor	`del A`
`__str__`	Printing	`print (A), str(A)`
`__call__`	Evaluates the object in the args	`A(args)`
`__add__`	`+` operator	`A+B, A += B`
`__sub__`	`-` operator	`A-B, A -= B`
`__mul__`	`*` operator	`A*B, A *= B`
`__div__`	`\` operator	`A\B, A\=B`
`__pow__`	`**` operator	`A**B, pow(A,B)`
`__eq__`	`==` operator	`A == B`
`__lt__`	$<$ operator	`A < B`
`__comp__`	Comparison	`A==B, A<=B,A>=B, A<B,A>B, A!=B`
`__or__`	`|` (`OR`) operator	`A|B`
`__getitem__`	Gets the value of `k` index	`A[k]`
`__setitem__`	Assigns a value to `k` index	`A[k] = value`
`__getattr__`	Gets an attribute	`A.attribute`
`__setattr__`	Sets the value of an attribute	`A.attribute = value`

Example: a `point` class

```
1  class point:
2      def __init__ (self, x=0, y=0):
3          self.x = x
4          self.y = y
5
6      def print (self):
7          print ' Point: ( ', self.x,' ,  ', self.y,' )'
8          # print (' Point: ( ', self.x,' ,  ', self.y,' )')
9
10     def modify_coord (self, newx, newy):
11         self.x = newx
12         self.y = newy
13
14     def move (self, u, v):
15         self.x += u
16         self.y += v
17
18     def newpoint_move (self, u, v):
19         p2 = punto ()
20         p2.x = self.x + u
21         p2.y = self.y + v
22         return p2
```

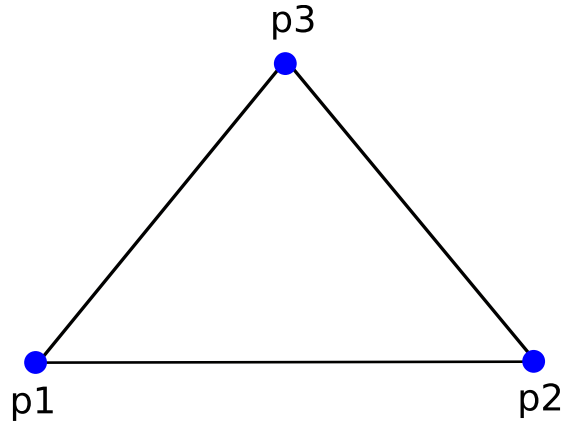
Example: a `point` class

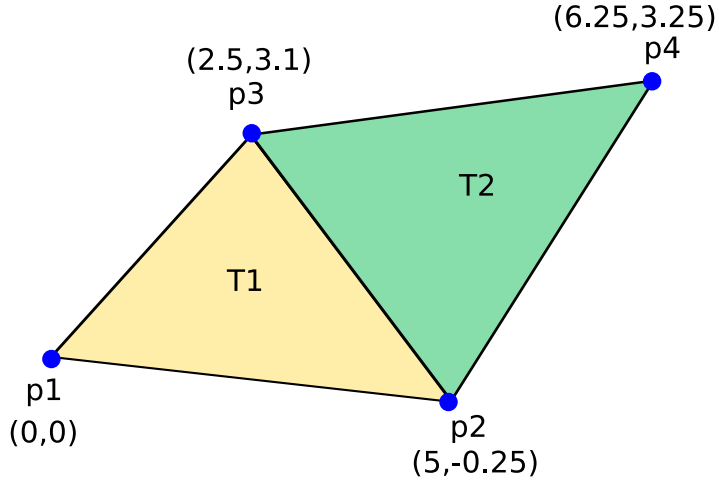
```
    def __add__ (self, p):
        self.x += p.x
        self.y += p.y
        return self

p0 = point ()
p1 = point (0., 1.)
p2 = point (3, 5)

p1.modify_coord (2.0, 6.0)
p1.print ()

p1.print ()
print (' p2: (%f, %f)' % p2.x, p2.y)
p3 = p1 + p2
p3.print ()
```

Example: a `triangle` class

```
1  import point
2
3  class triangle:
4
5      def __init__ (self, p1, p2, p3):
6          self.vertices = [p1, p2, p3]
7
8      def print (self):
9          # Writes the points coordinates
10         print '---  Vertices of the triangle ---'
11         # print ('---  Vertices of the triangle ---')
12         for k in self.vertices:
13             k.print ()
14
15     def mod_pointcoord (self, ind, x, y):
16         self.vertices [ind].modify_coord (x,y)
17
18     def surface (self):
19         v = self.vertices
20         x1 = v [1].x - v [0].x
21         y1 = v [1].y - v [0].y
22         x2 = v [2].x - v [0].x
23         y2 = v [2].y - v [0].y
24
25         return abs (x1*y2-x2*y1)
```

Example: a `triangle` class





Example: a `triangle` class

```
1  import point
2  import triangle
3
4  p1 = point (0.0,0.0); p2 = point (5,0.1)
5  p3 = point (2.5,3.5); p4 = point (7.2,1.5)
6
7  t1 = triangle (p1, p2, p3)
8  t2 = triangle (p2, p4, p3)
9
10 for t in [t1,t2]:
11     t.print ()
12     print 'Triangle surface: ', t.surface ()
13     # print ('Triangle surface: ', t.surface ())
```

Example: a `triangle` class

```
# If we modify the coordinates of p2, changes affect to its
    vertices
# as they have the same memory direction

p2.modify_coord (4.5, -0.25)
for t in [t1,t2]:
    t.print ()

t1.mod_pointcoord (2, 2.65, 10)

for t in [t1,t2]:
    t.print ()
```

Object–oriented programming: heritage

Heritage is one of the most important features of OOP

- New classes can be created from an existing class, from which they inherit attributes and methods, that can be adapted and/or extended
  For example:
  - A car, a plane, a train and a ship are all vehicles which have common attributes and methods (number of passengers, maximum speed, turn off, turn on, ...); each of them can have its particular attributes and methods (number of wheels, number of wings, ...)
    We can consider a `vehicle` class, and a `car`, `plane`, `train` and `ship` subclasses
  - A triangle and a rectangle are both polygons
- The new class is known as derived class, child class, heir class or subclass
- The original class is known as base class, parent class or superclass

Object–oriented programming: heritage

How to indicate that a class inherits from another?

```
class subClassName (superClassName):
```

- If both are in the same file, the superclass must be defined before the subclass
- The subclass inherits all the methods from the superclass
- If a method is simultaneously defined in both classes, the one of the subclass is considered
- If the subclass has its own method `__init__`, this one must explicitely call to the base class `__init__` method

Object–oriented programming: heritage

```
class polygon:
  def __init__(self, np=0):
      self.numpoints = np
      self.__color = 'red'
  def get_color (self):
      return self.__color
  def set_color (self, c):
      self.__color = c
  def surface (self):
      pass
```

```
class triangle (polygon):
    def __init__(self):
      polygon.__init__(self,3)
      self.vertices = None
    def set_pts(self,pt0,pt1,pt2):
      if((pt0[0]==pt1[0]
          and pt0[0]==pt2[0])
       or (pt0[1]==pt1[1]
           and pt0[1]==pt2[1])):
        print ' Error'
        # print (' Error')
      else:
        self.vertices=[pt0,pt1,pt2]
    def surface (self):
      [x0,y0]=self.vertices[0]
      [x1,y1]=self.vertices[1]
      [x2,y2]=self.vertices[2]
      s = abs((x1-x0)*(y1-y0)-
          (x1-x0)*(y1-y0))/2.0
      return s
```

```
t1 = triangle ()
t1.set_pts ([0,0],[1,0],[1,1])
t2 = triangle ()
t2.set_pts ([-1,0],[0,0],[-1,2])
t2.set_color ('blue')
print 't2 surface: ', t2.surface ()
# print ('t2 surface: ', t2.surface ())
```

Object–oriented programming: heritage

We can also use a class defined in a different file:

In file `polygon.py`:

```
class polygon:
    ...
```

In file `triangle.py`:

```
from polygon import polygon
class triangle (polygon):
    ...
```

Exercise: Write the `rectangle` class, child of the `polygon` class

- Each rectangle is defined by an lower left and an upper right vertices
- Write the surface method
- Write a method that gives the four vertices of the rectangle
- From an external code, build different triangles and rectangles and test the implemented methods.

Object–oriented programming: further topics

- Any variable of the form `self.var` is public and accesible from any method of the class, and from the program where the class instance is used

- Writing `__` before a variable or a function, we create a *pseudo-private* variable or function: `self.__privatevar`

Object–oriented programming: further topics

- Attribute data (also called *instance variables* in JAVA and *member variables* in C++) are the variables associated to a specific instance of a class
  - PYTHON also has class attributes, which are variables associated to the class but not to a specific instance
- From outside the class, we access the attribute `nombre` of the instace `obj` by writing obj.name
- Inside the class, we do `self.name`
- An attribute exists since the moment a value is assigned to it

Instance attributes and class attributes

```
1  class Point:        # File: ej1_class_ptos.py
2      """ Class Point """
3
4      numPts = 0
5
6      def __init__ (self, x=0, y=0):
7          Point.numPts += 1   # variable of the class, shared
               by all its instances
8          self.x = x          # variable of the instance
9          self.y = y
10
11     def print_pt (self):
12         print ' Point: ( ', self.x,' , ', self.y,' )'
13        # print (' Point: ( ', self.x,' , ', self.y,' )')
14
15 if __name__ == '__main__':
16   pt1 = Point ()
17
18   pt2 = Point (3,5)
19
20   pto1.print_pt ()
21   print 'p2: (%f, %f)' % pto2.x, pto2.y
22   print 'Total number of points: ', pto1.numPts
23   # print ('p2: (%f, %f)' % pto2.x, pto2.y)
24   # print ('Total number of points: ', pto1.numPts)
```

Polymorphism

- **Polymorphism** is the capacity of objects and methods of a class to react in a different way depending on the parameters or arguments they receive

- Being a dynamical language, polymorphism is no very important in PYTHON