Scientific computing in Python

Vincent.Favre-Nicolin@ujf-grenoble.fr

November 3, 2015

Contents

Licence

1 Introduction

A large number of languages for scientific computing (C/C++/fortran) are well-established and have been existing for a long time. So why develop (and learn !) a new one ? The main reason is that traditional languages are either ill-suited for a general-purpose use (calculations and creation of a graphical user interface - fortran), or their syntax is sufficiently complex or rigid to make their use difficult for an occasional programmer (c++). This is the main reason behind the success of commercial software (Mathematica, Maple, Matlab...) and languages (IDL,...) which combine a relatively simple syntax with high-level functions (graphical representations, computing libraries,...), which can also be used using a command-line interface (without prior compilation.

The Python language (*http://www.python.org*) was developed to cope for these limitations inherent to c/c++/fortran. It presents numerous advantages compared to traditional langages:

- **natural** (using indentation, dynamic typing of variables)
- can be used with a **command line** (***interacti*ve mode**)
- **modern** (object-oriented)
- **extendable** (*extreal modules* for scientific calculations and graphical display, written in C/C++/fortran)
- widely used for **scientific computing** (re-using already existing computing libraries)
- **open-source** & **multi-platform** (Linux, windows, MacOS,...)

Python is a high-level language, using "just-in-time" compilation, and therefore slower than a low-level, compiled language (C/C++/FORTRAN). This is only a minor drawback, as each time a truly "intenseive" calculation is necessary, it can be performed in an *external module* written in another, more efficient (but more tedious to use) language. generally speaking, only a small part of a program really needs to be optimised (a rule of thumb is that 95% of calculation time is spent in 5% of the code).

Moreover, it turns out that we often spend more time writing a program than using it, which makes the true importance of the computing optimization quite relative. The interest of python is twofold: (1) writing a python program is quick thanks to its simple syntax, and (2) it is very easy to re-use existing libraries.

Fundamental differences compared to C++ are:

- It is an **interpreted language** (using just-in-time compilation), which can be used with a **command-line**
- "blocks" of code (e.g. defining the inside of a loop) are identified using **indentation** (**the number of spaces at the beginning of the line**), and not using brackets {...}
- **object types** ('int', 'float', 'string') is **dynamic**, i.e. the type of objects does not need to be declared in advance, and can be modified during the program

2 Installing Python

- Linux: Python is installed on all standard distributions - you need to install other modules such as scipy, numpy, matplotlib, etc... depending on what you want to do.
- Windows: the standard distribution available from http://www.python.org should only be used if you want to useplain python. In order to use scientific computing libraries, you should instead install a full package which includes python and most useful libraries. There are two such packages: Enthought Python Distribution (http://www.enthought.com/ , free for an academic use), and Python(x,y) (http://code.google.com/p/pythonxy/).
- MacOS X: the recommended installation would be through MacPorts (http://www.macports.org/). It is also possible to use the Enthought Python Distribution (http://www.enthought.com/ , free for an academic use)

It is also recommended to use a dedicated editor/IDE (Integrated Development Environment): a relatively lightweight and yet powerful one is Spyder (http://code.google.com/p/spyderlib/). FOr windows it is included in Python(xw,y), it is available through MacPorts for MacOS X, and as package for Linux distributions.

3 Programming using Python

3.1 Command line or editor ?

Python can be used from a command line (interpreted language). To launch it, just use the program's shortcut, or type "python" from a console. Then at the prompt (">>>"), type "`print ''toto''`". The result is:

```
[vincent@localhost vincent]$ python
Type "help", "copyright", "credits" or "license" for more information.
>>> print "toto"
toto
>>>
```

Using a different langaue, it would have been necessary to (1) write the program in a file, then (2) compile it before (3) running it. In fact, it is also possible to write the program in a file. TO do this, just open your text editor (such as spyder), then write the program (`print ''toto''`), and save i under the name *toto.py* (not that the name extension ".py" is not mandatory, but it helps the editor recognize the file as a python program, so that syntax highlighting is done correctly). Then run the program using the command "`python toto.py`" from the command line of your console (you can also use "`python -i toto.py`" to remain inside the python interpreter after the program has finished). In practice, writing a program line-by-line on the command-line or typing it beforehand in a file is **rigorously equivalent**.

To use the command line, it is even better to use **ipython**: it is a more interactive python version, which remembers the previously typed commands during priori python sessions (use down and up arrows to recall the commands in your history), and can also perform **auto-completion** : just type the beginning of a command or a variable name and hit the tabulation key to perform the auto-completion. Ipython is included in IDEs like Spyder.

Her you go ! You now can program with Python! In practice, quick tests (a few lines) can be done with the command line, and longer programs are written in an editor or IDE before being run.

Note 1 : to quit the Python interpreter, type `ctrl-d`

Note 2 : the # symbol is used for **comments** - anythong after a # is ignored by python, but can be used to document the code:

```
>>> print ‘‘toto’’ # this is my first python program !
toto
```

Note 3 : to run commands inside a text file (e.g. `toto.py`), you can use the `execfile()` function:"`execfile(‘‘toto.py’’)`"

Note 4 : in order to use **accents in Python**, you must declare the **encoding** used, by putting at the beginning of the python program file, e.g. (other encodings can be used, some IDEs like Spyder will automatically include this line):

```
# -*- coding:  iso-8859-15 -*-
```

3.2 Inline help

All standard objects, as well as those in properly written modules, include an inline help text, which can be invoked using the `help()` function. For example "`help(str)`", "`help(list)`" give access to help about strings and lists. When a module is imported (e.g. '`import math`'), it is possible to read the help corresponding to this module (e.g. '`help(math)`') or a function of this module (e.g. '`help(math.sin)`').

When **ipython** is used, it's even simpler, you can just type the command followed by one or two question marks (one will show the help text, two will show the code of the object or function if it is written in python).

3.3 Basic variables and data types

3.3.1 Simple variables

To create a variable, one just needs to affect a value:

```
>>> x=1
>>> print x
1
```

You can see here a major difference with other languages: it is not necessary to declare *a priori* the **type** of the x variable (integer 'int', floating point number 'float', 'string',...). However, the type *does* exist, and can be determined using the `type`() function:

```
>>> print type(x)
<type 'int'>
```

Python automatically decided that x was an integer (`int` = integer). Try with other values: "`x=1.0`" (`float` =floating point number), "`x=''hello''`" (`string`=character string).

It should also be noted that **this type can be changed** by affecting x with a different value:

```
>>> x=57
>>> type(x)
<type 'int'>
>>> x="Paris"
>>> type(x)
<type 'str'>
>>> x=3.14159263
>>> type(x)
<type 'float'>
```

While this is practical, this can easily lead to programming errors, so a good rule is to **use explicit variable names** (`age` is an integer, `name` a string, etc..) and **do not change variable types in the middle of a program**.

Warning: python is **case-sensitive** (upper and lowercase letters), so that `myvar`, `MyVar` and `MYVAR` are three separated variables. The same is true for all keywords and functions.

3.3.2 Complex numbers

Python handles complex numbers natively, the "i" complex number being noted using the "j" letter:

```
>>> a=2+1j
>>> type(a)
<type 'complex'>
>>> b=3.5+4j
>>> print a+b
(5.5+5j)
>>> print a*b
(3+11.5j)
```

To use mathematical functions with complex numbers, the `cmath` module will need to be imported. However in practice, the `math` and `cmath` modules are limited to operations on single values, so that it is much more practical to use the `numpy` module, which can operate indifferently on single values or arrays, real or complex.

3.3.3 Strings

Beyond the simple assignment (`myString=''hello''`), it is possible to write a string in the same manner as a `printf` function, i.e. by inserting codes in the string (`%s` for a string, `%d` for an integer,`%f` for a floating point value, with optionally the number of characters before and after the decimal point: `%4.2f` means '*floating point number written using 4 characters, with 2 digits after the point*'), followed by %(var1,var2,...) where var1, var2 are the values to be inserted:

```
>>> age=5
>>> name="Vincent"
>>> height=1.73
>>> myString="%s is %d years old and is %4.2f meters tall"%(name,age,height)
>>> print myString
Vincent is 5 years old and is 1.73 meters tall
```

It is also possible to extract part of a string:

```
>>> print myString[2:6]    # extract characters 2 to 6 (6 excluded)
ncen
```

However it is not possible to modify the string this way: "`myString[2:6]=''tapl''`" will lead to an error.

You can also note that numbering in python starts at 0 (like in C/C++ but unlike fortran), so that the first character of the string is `myString[0]`.

Strings can be concatenated using the + sign:

```
>>> print "Scientific" + " Programming"
Scientific Programming
```

The length of a string can be obtained using `len()`:

```
>>> len("hello")
5
```

`help(str)` will give you all the functions which can be used for any string.

3.3.4 Lists

A particularly interesting type is a `list`, which allows to aggregate any list of objects:

```
>>> x=[1,4,32]
>>> print x
[1, 4, 32]
>>> print type(x)
<type 'list'>
```

```
>>> print x[2]
32
```

Again here, the numbering of items in the list begin at 0: if there are n elements in `myList`, they can be accessed by `myList[0]`...`myList[n-1]`.

A `list` can contain objects of *any type* (contrary to an array, which will be used with the `numpy` module):

```
>>> MyList=[1,"Grenoble",57.0]
>>> print MyList[0]
1
>>> print MyList[1]
Grenoble
>>> print MyList[2]
57.0
>>> print type(MyList[0]),type(MyList[1]),type(MyList[2])
<type 'int'> <type 'str'> <type 'float'>
```

An existing list can be modified:

```
MyList[1]=42          # the 2nd element of MyList is now 42
MyList.append(45)     # adds object ''45'' (an integer) at the end of MyList
MyList.insert(i,x)    # inserts x at position i in MyList
MyList.reverse()      # reverses the order of elements in the list
MyList.sort()         # sorts the elements of the list (NB: this will only works
if the elements can be compared to another)
```

The length of list is obtained with the `len()` function: `len(MyList)`.

Two lists can be concatenated using the + operator:

```
>>> list1=[1,"ab",3.5]
>>> list2=["toto",57]
>>> print list1+list2
[1, 'ab', 3.5, 'toto', 57]
```

One element of a list can be deleted using `del()` :

```
>>> MyList=[1, 'ab', 3.5, 'toto', 57]
>>> print MyList
[1, 'ab', 3.5, 'toto', 57]
>>> del(MyList[1])
>>> print MaListe
[1, 3.5, 'toto', 57]
```

Lists are very important objects in pyton, because loops often use an iteration on the elements of a list. The complete list of functions which can be used for a list can be obtained using help(list).

3.3.5 Tuple *[optional: you can skip this notion]*

A tuple is an object very similar to a list: the main difference is that a tuple is immutable, i.e. it cannot be modified like a list (no appending of elements). It is a list of objects separated by commas; the list can be surrounded by parenthesis to more clearly define the tuple:

```
>>> myTuple=('a',"toto",54.2)
>>> type(myTuple)
<type 'tuple'>
```

Objects can be extracted from a tuple:

```
>>> print myTuple[1]
```

```
toto
>>> print myTuple[1:3]
('toto', 54.200000000000003)
```

However it is not possible to modify an object inside a tuple (i.e. "myTuple[1]=''tata''"is an illegal instruction), or add (insert/append) elements to a tuple.

Generally speaking, a tuple is not explicitly created, but they correspond to variables which are temporary, when several objects need to be manipulated:

```
>>> a,b=5,3        #creation of two variables on a single line
>>> print a,b
5 3
>>> a,b=b,a         # swap two variables
>>> print a,b
3 5
```

3.3.6 Dictionaries

A `dictionary` is also used to list objects, but instead of numbering the objects, these can be accessed using a `key`, which can be any type of object (as long as the list of keys can be sorted, i.e. is hashable). It is therefore a list of `key:value` objects (for those used to C++, it is equivalent to `std::map<>`).

```
>>> fra_deu={"un":"ein","trois":"drei","deux":"zwei"}    # Creation
>>> fra_deu["trois"]            # Access one element of the dictionary
'drei'
>>> fra_deu["quatre"]              # Key does not exist !
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
KeyError:  'quatre'
>>> fra_deu["quatre"]="vier"    # Add the missing entry
>>> fra_deu["quatre"]
'vier'
```

Dictionaries can be extremely useful to store data, e.g. if you have datasets with temperature, humidity, annotations - all these can be stored using different key in a single dictionary object.

You can list all the functions available for a dictionary using `help(dict)`.

3.3.7 COnversion from one type to another

A variable can be converted from one type to another:

```
>>> x=float(1) # integer to floating point number
>>> print type(x)
<type 'float'>
>>> x=float("1.0") # floating point number from a string
>>> print type(x)
<type 'float'>
>>> print x
1.0
```

3.4 Loops and operators

3.4.1 operators `==`, `and`, `or`, `not`

The == operator can be used to compare two objects:

```
>>> 1==2     # are 1 and 2 equal ?
```

```
False
>>> 1==1
True
```
Other basic comparison operators are <, >, <=, >=,!=.

Note that in python any non-zero number is interpreted as `True`, i.e. "`if n`" where `n` is a number, will always be `True` unless n is equal to zero.

Operator `not(x)` will return the negation (in the boolean sense, `True`/`False`) of an expression:
```
>>> not(26)
False
>>> not(0)
True
```
`and` and `or` permit usual logical operations (these are not bitwise operations, they can only be used for boolean tests):
```
>>> 1 and 0
0
>>> 1 and 1
1
>>> 1 or 0
1
```

3.4.2 Instructions `if...elif..else`

A conditional instruction can be executed using `if`:
```
>>> x=1
>>> if x==2:                 # is x equal to 2 ?
...    print "x equals 2"  # note the spaces at the beginning of the line (indentation)
...  elif x==3:
...    print "x equals 3"
...  else:
...    print "x is not equal to 2"
...    print "or 3"
...
x is not equal to 2
or 3
```

The spaces at the beginning of the lines (indentation) play an essential role: as long as the spacing remains the same, we are at the same execution depth, in the same manner as brackets {} are used in other languages to indicate the beginning/end of the group of instructions to execute. Nested levels of instructions can be used by increasing the indentation depth:
```
>>> x=12
>>> if (x%2)==0:             # is x modulo 2 equal to 0 ?
...    if (x%3)==0:
...        if (x%7)==0:
...            print "x is a multiple of 2,3 and 7 !"
...        else:
...            print "x is a multiple of 2,3, but not 7..."
...    else:
...        print "x is a multiple of 2,but not 3"
...else:
...    print "x is not a multiple of 2"
...
```

```
x is a multiple of 2,3, but not 7...
```

3.4.3 The `while` loop

A series of instructions can be executed as long as a condition is `True`:

```
>>> x=7
>>> while x>=1:
...    print x,">=1 !  Continue..."
...    x = x-1                 # this could also be written:  x -= 1
7 >=1 !  Continue...
6 >=1 !  Continue...
5 >=1 !  Continue...
4 >=1 !  Continue...
3 >=1 !  Continue...
2 >=1 !  Continue...
1 >=1 !  Continue...
```

3.4.4 `for` loops

a for loop is written:

```
>>> for i in [2,3,7]:     # Generally:  for variable in liste:
...    print i
...
2
3
7
```

Since it's fatidious to write a large number of values, the `range()` function can be used to return a list of integers

```
>>> range(5) # returns a list of 5 integers starting a 0
[0, 1, 2, 3, 4]
>>> range(2,5) # returns a list of integers from 2 to 5 (5 excluded)
[2, 3, 4]
>>> for i in range(2,5):
...    print i
...
2
3
4
```

Note: the `range()` function creates list of numbers which can be used for iteration ; it can also be used to create a list without any loop. In fact, for a loop, it is better to use the `xrange()` function, which does not allocate the entire list of numbers in memory but only an iterator which allows the loop to work. It is much more efficient memory-wise, especially if the number of iteration is very large (think, larger than the number of available bytes in memory).

3.4.5 Exercise : prime factors decomposition

Write a loop which, given an integer n, gives the list of its prime factors by writing then on the screen (or stores them into a list). The standard mathematical operators are * (multiplication), / (division), % (modulo), and the comparison operators are: ==, <, <=, >, >=.

Note: contrary to C and C++, the modulo operation x%n returns a value within [0;n-1], even if x is negative.

3.5 Functions

Functions cane be used in Python. We have already used `print()` and `type()`. In general, a new function is called in the following way: `functionName(parameters)` or if the functions returns something: `returnedVariable=functionName(parameters)`.

To define a new function, the keyword `def` must be used:

```
>>> def MyFonction(name,age):
...    print name,"is",age,"years old"        #note the spaces (indentation) at
the beginning of the line
...
>>> MyFonction("Pinocchio",5)
Pinocchio is 5 years old
```

`name` and `age` are the *function parameters* ; again, it is not necessary to declare the type of those parameters. In fact, we can call the function by using a string for age:

```
>>> MyFonction("Pinocchio","five")
Pinocchio is five years old
```

However, if the declaration of the function does not impose an *a priori* control on the parameter types, they must be compatible with the function code. Try:

```
>>> def Addition(a,b):
...    return a+b          # Returns a value using the ‘‘return’’ keyword
...
>>> print Addition(1,2)        #two integers
3
>>> print Addition(1,3.0)      #integer + float returns a float
4.0
>>> print Addition(1,"toto")   # integer + string = error !
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in Addition
TypeError:  unsupported operand type(s) for +:  'int' and 'str'
>>> print Addition("tata","toto")      # two strings
tatatoto
```

Note: exactly as in the case of loops, it is the ***indentation*** (the number of spaces at the beginning of the line) which defines which instructions are within the function. The replaces the use of brackets {} in other languages.

Exercise: reuse the code written for the prim factor decomposition, and write a function which takes an integer as parameter, and returns a list of prime factors of that integer.

3.6 Objects scope : local and global variables

As for other languages, variables in Python can be local or global. The former only exist at the level where they have been defined (and not above). By default, all variables are local, e.g.:

```
>>> x=12.3              # First variable ‘‘x’’
>>> def fonction1():
...    x=1              # Second variable ‘‘x’’
...    print x
...
>>> def fonction2():
...    x="Toto"         # Third variable ‘‘x’’
...    print x
```

```
...
>>> print x
12.3
>>> fonction1()
1
>>> fonction2()
Toto
```

In this example the three variables share the same name but are completely independent. It should be noted that a given indentation level cannot modify variables from a lower indentation (contrary to what happens in C/C++). try for example:

```
>>> ct=0
>>> def counter():
...     ct+=1
...     return ct
...
>>> counter()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in compteur
UnboundLocalError:  local variable 'ct' referenced before assignment
```

However, it is perfectly possible to read the `ct` variable, without modifying it.

To be able to modify a variable at different positions (other than by passing the variable as a parameter to a function), the `global` keyword can be used:

```
>>> ct=0
>>> def counter():
...     global ct
...     ct+=1
...     return ct
...
>>> counter()
1
```

Note : ***in general, it is best to avoid global variables, by passing variables as parameters to functions, or by storing these variables in a module.***

3.7 Exercise

Write a functions which takes as argument two lists inlcuding the list of atomic positions in space (e.g. ["C1", 0, 0.5 , 0.7]), and returns the interatomic distance with a string like:

"The distance between atoms C1 and H3 is 3.5 A"

TO calculate a square root, use the `sqrt` function in the `math` module ("`import math`", then use ``` ``math.sqrt()`` ").

Note: to import a module (such as `math`), there are two ways to do it: (a) "`import math`" or (b) "`from math import *`". If (a) is used, all the functions in the math module can be used with the "math." prefix (example: "`math.sin(math.pi/3.0)`"). If method (b) is used , the functions can be called directly "`sin(pi/3.0)`". Method (a) is recommended if there is a risk of conflict between names of functions, for example if you want to redefine the `sin()` function. It is therefore recommended if you import many modules.

4 Classes and Objects

4.1 Introduction

Historically, several programming approaches have been used:

In the **procedural approach**, a single program executes a series if instructions. This is well suited for relatively short programs, writtent by a single person.

For longer programs (and several people involved, in parallel or successively in time), it becomes necessary to separate different tasks in the program, using differet functions: this is a **structured approach**.

The most recent approach is **Object-Oriented-Programming** (OOP): data to be analyzed by the software are stored in objects, which comprise both data and functions which are used by the object. For example, a Molecul object ca contain a list of atoms, their positions, and also functions to calculate the molecule's energy, the total charge, dipolar moment, etc... Moreover, it is possible to create a hierarchy of objects, which inherit teh propertoes of their ancestors, etc...

***Warning**: even if object-oriented programing is the most modern approach, the other two remain perfectly valid, especially for small projects. It is not useful to use OOP for calculations which can be written in just a few lines !*

4.2 Example

Let us construct a class of objects corresponding to a "`Polygon`":

```
   >>> class Polygon:             # Class declaration
   ...    circ=1.0                  # An data member (or attribute) of the class,
to store the circumference
   ...    def Circumference(self):  # function returning the value of the circumference
   ...          return self.circ
   ...
   >>> a=Polygon()                 # Creation of one object of type Polygon (called
an instance of the Polygon class)
   >>> print a.circ                # Direct access to the circumference
   1.0
   >>> print a.Circumference()     # Get the circumference using the function
   1.0
   >>> a.circ=12.3                 # Change the circumference
   >>> print a.Circumference()     # Check our change
   12.3
```

In the above declaration, the `'self'` keyword is used: this corresponds, for the code within the class, to the object (the instantiated object) itself. For example "`self.circ`" means "variable `circ` inside the `Polygon` object within which the code is executed (if within the `Polygon` "a", it corresponds to "`a.circ`").

4.3 Data members (attributes)

A **data member** or **attribute** is a variable stored within an object, like the 'circ' variable in the previous example. There can be as many attribute as needed. It is even possible (contrary to C++) to **add attributes outside the class declaration** code. Continuing from the code abovde, try:

```
   >>> Polygon.surface=45.6   # Adds data member 'surface' to the class
   >>> b=Polygon()
   >>> print b.circ,b.surface,a.circ,a.surface
   1.0 45.6 12.3 45.6
```

The added attribute can be used in all Polygon objects, including 'a' which was created beforehand. It is also possible to add an attribute in a single instance of a class... Of course, it is highly advised not to abuse this, and avoid adding data members after the class declaration, because it makes the code very difficult to read.

4.4 Member functions (methods)

A **function member** (also called a **method**) is a function which is defined within the declaration of the class, and can therefore access the data memebers, such as the `Circumference(self)` function, defined for the `Polygon` class. It takes at least one parameter, "`self`", which corresponds to the object itself (equivalent to the "`this`" pointer of C++), and gives access to the data members.
Any number of function members can be declared.

It is generally recommended (in OOP) not to access directly data members from outside the class, but to rather use member functions to access all data. The idea is that code outside the class should not know how the data is stored, so that this storage may vary, whereas the member functions will remain the same (the API -Application Programming Interface- remains the same).

For example in the `Polygon` class, the `circumference` is stored as a data member `circ`, but in a `Square` class (see below), there are no `circ` attribute, since the `circumference` can be calculated from the side length. Nevertheless, both classes provide the same Circumference() function, althought the code executed in each case will be different. The function members are used as a "abstraction layer" which hides the way the calculations and data storage is actually done.

In a member function, it is essential to distinguish the member functions (those preceded by "`self.`") and the local variables. The data members are the only ones which will not be destroyed upon termination of a function member, and therefore can be used to store data in a persistent manner inside the object.

```
>>> class MyObject:
...     var1=1.0       # data member declared at the beginning of the class
...     def __init__(self):  # Constructeur
...         varlocale=1.0    # Local variable
...         self.var1=2.4    # Change data member self.var1
...         self.var2="toto" # Add a second data member self.var2
...     def Function1(self):
...         print self.var1,self.var2
...         varlocale="tata" # this variable is only local
...
>>> a=MyObject()
>>> a.var1
2.3999999999999999
>>> a.var2
'toto'
>>> a.Function1()
2.4 toto
>>> a.varlocale     # error because varlocale is not a data member
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
AttributeError:  MyObject instance has no attribute 'varlocale'
```

For every object, it is important to remember to use "`self.variableName`" to systematically distinguish member and local variables.

4.5 Special member functions *[advanced]*

Some meber functions are predefined in python because they are used to perform predefined operations. These functions always begin and end with two underscore signs.

4.5.1 Constructor `__init__`

A constructor is the member function which is called when a new object is created - declaring a new constructor can be used to pass parameters during initialization:

```
>>> class Polygon:
...     circ=1.0
...     def __init__(self,circ0):     # Takes as parameter the circumference value
...         self.circ=circ0             # Initialize the circumference value
...     def Circumference(self):
...         return self.circ
...
>>> a=Polygon(10.5)                   # Creates a Polygon with circumference equal
to 10.5.  Calls the constructor
>>> print a.Circumference()
10.5
```

4.5.2 Destructor `__del__`

A destructor is the function which is called when destroying the object. It takes only one argument (`self`). It can be used to 'clean up' data created by the object.

4.5.3 Operators `__add__`, `__sub__`, `__mul__`, `__div__`, `__pow__`

These are operators (+, -, *, /, **) which allow mathematical operations. They take two arguments, `self` and another which designates the second member of the operation (ususally noted `rhs` because it is the right-hand-side variable).

```
>>> class Polygon:
...    circ=1.0
...    def __add__(self,poly):   # addition
...       s=Polygon()             # Create a new object which will be returned
...       s.circ=self.circ+poly.circ
...       return s
...
>>> a=Polygon()
>>> b=Polygon()
>>> b.circ=3.5
>>> print (a+b).circ      # Circumference of the sum of a and b
4.5
```

Redefining these operators can be practical, but it is advised to only do this when maintaining the original intent of the operator (e.g. do not redefine `__add__` and actually perform a subtraction !).

4.6 Inheritance *[adavanced]*

Inheritance can be used to create classes which inherit all or part of a parent class. For example, a `Rectangle` class can inherit from the `Polygon` class:

```
>>> class Polygon:
...      def __init__(self,circ0=2.1):      # A default value is given for the parameter
to the constructor
...          self.circ=circ0
```

```
...      def Circumference(self):
...          return self.circ
...
>>> class Rectangle(Polygon):         # Inherits from class Polygon
...      def __init__(self,x0=2.5,y0=3.5):
...          self.x=x0
...          self.y=y0
...          self.circ=2.0*(x0+y0)
...
>>> a=Rectangle(3.0,4.0)
>>> print a.Circumference()  # The Circumference function is inherited
14.0
```

Another class can be further derived for a square, and can replace the original `Circumference()` function:

```
>>> class Square(Rectangle):     # Inherits from Rectangle (and therefore from
Polygon)
...    def __init__(self,x0):
...        self.x=x0
...    def Circumference(self):       # The Circumference function is rewritten
...        return 4*self.x
...
>>> a=Square(4.0)                   # Square of size 4.0
>>> print a.Circumference()
16.0
```

Inheritance can be very useful when creating classes with similar properties. However, this approach should not be over-used, since it can lead to needlessly complex programming ! It is generally useful only for at least medium-sized projects (>1000 lines), but it can also be useful to re-use code (coming from example from an existing library), by adding a few functions.

4.7 Warning: operator "==", difference between *equality* and *identity*, *copy* and *reference [advanced]*

Let's use the previously declared functions, and compare the declared objects:

```
>>> a=Square(4.0)
>>> b=Square(4.0)
>>> c=a
>>> print a==b, a==c, b==c
False True False
```

What happened here ? CLearly, the three squares are 'equal' and we could expect "`True True True`" for an answer... In fact, the == operator did not comare the values (are the squares equal), but only the *addresses* in memory, thus assessing the *identity* of the three objects. This can be better understood if we write :

```
>>> print a.x,c.x
4.0 4.0
>>> c.x=5.7
>>> print a.x,c.x
5.7 5.7
```

What happens when we write "c=a" is not the creation of a new Square object, but rather the creation of c as a **reference** to a. In other words, c points to the same memory addresses as a, so that they represent exactly the same object (as far as the 'dumb' computer is concerned).

The idea behind this behavior (which is a cornerstone of Python), is to minimize the memory footprint of the program by making by default a reference of an object rather than a true copy. It is important to keep this in mind, because when writing "a=b", any subsequent modification to b will also affect a !! Note that this is only true for 'complex' objects, i.e. it excludes simple data types such as integers and floats.

To avoid this behavior, it is possible to force a true copy using the `copy()` or `deepcopy()` functions of the `copy` **module**:

```
>>> import copy
>>> a=Square(4.0)
>>> b=Square(4.0)
>>> c=copy.copy(a) # in fact it is better to use c=copy.deepcopy(a)
>>> print a==b, a==c, b==c
False False False
```

Note 1: to implement a "mathematical" comparison of the three square objects, all that was needed would be to add a special member function `__eq__()` which would be called when writing "`a==b`".

Note 2: parameters to a function are also passed as a *reference*, so that if you modify a variable which was supplied to a function, it will effectively modify the original variable.

4.8 Exercise

Create an `Atom` class with four attributes: name and x,y,z coordinates, and with functions allowing to create the class and giving access to the four attributes (`Name(), X(), Y(), Z()`). Also write a function which computes the distance with another atom.

then, create a `Molecule` class which includes a list of `Atom` objects as attribute, and has member functions allowing:

- Adding an atom
- Access one atom (using the `__getitem__()` special function)
- Display the list of all interatomic distances of the molecule's atom.

5 Basic file input/output

A python `file` object can be created using the `open()` function:

```
>>> MyFile = open(''filename'','r')
```

The mode used can be 'r' (read only), 'w' (write only)... Other modes (mixed, read/write, binary) also exist.

Reading text files (not binary) can be performed using the `read()`, `readline()` and `readlines()` functions:

```
>>> MyFile.read()     # Read the entire file as a single string
>>> MyFile.readline()    # Read the next line as a string
>>> MyFile.readlines()    # Read the entire file and returns a list of strings,
one for each line
```

In order to read a list of numbers on one line, just read the line into a string using `readline()`, and then separate the different fields using the string's `split()` function. For example:

```
>>> f=open("tmp0",'r')
>>> line=f.readline()       # read one line
```

```
   >>> print line             # display the read string
   Toto 1.8 9.2
   >>> print line.split()    # split all fields (separated by space(s) or a tabulation)
into a list
   ['Toto', '0.2', '9.2']
   >>> print float(ligne.split()[1]) # Convert to float
   0.2
   >>> f.close()      # close the file
```

`help(file)` lists the function available for a a `file` object.

NOTE: well-formatted data can be read much more easily using numpy's function (e.g. `numpy.loadtxt()`)

5.1 Exercise

Create a file with a list of atoms (one name and three coordinates per line). Add a function `ImportFromFile(self, filename)` in the `Molecule` class, to import the atom list. And then list all the interatomic distances.

Then add a function which can write in a file the list of atoms... and check that you read it back afterwards !!!