Programmation Scientifique avec Python

Vincent.Favre-Nicolin@ujf-grenoble.fr

3 novembre 2015

Table des matières

Licence

1 Introduction

Il existe déjà un grand nombre de langages utilisés pour le calcul scientifique (C/C++/fortran), stabilisés et développés depuis de nombreuses années. Pourquoi développer (et donc apprendre !) un nouveau langage ? Il s'avère que les langages traditionnels sont soit inadaptés à une utilisation généraliste (calculs et création d'un interface graphique - ex : fortran), ou ont une syntaxe suffisamment complexe ou rigide pour rendre difficile son utilisation par un utilisateur occasionnel (C++). C'est ce qui a permis le succès de logiciels commerciaux spécialisés dans le calcul (Mathematica, Maple, Matlab,...), et de langages (IDL,...), qui associent une syntaxe relativement simple avec des fonctionnalités (représentation graphique, librairies de calcul...) étendues, et utilisables directement à partir d'une ligne de commande (sans compilation préalable).

C'est pour pallier ce manque que le langage Python (*http ://www.python.org*) a été développé. Il présente de multiples avantages par rapport aux langages traditionnels :

— **naturel** (utilisation de l'indentation, typage dynamique)
— utilisable en **ligne de commande** (donc ***interactif***)
— **moderne** (orienté objet)
— **extensible** (*modules externes* de calcul et de visualisation écrites en C/C++/fortran)
— très utilisé pour le **calcul scientifique** (grâce à la réutilisation de librairies de calculs déjà existantes)
— **libre** ("open-source") et **multi-plateforme** (Linux, windows, MacOS,...)

Python est un langage de bas niveau, compilé "au dernier moment", et donc plus lent qu'un langage de bas niveau compilé (fortran/C/C++). Ce n'est en fait pas un inconvénient, car chaque fois qu'un calcul "intensif" est nécessaire, il est possible de l'effectuer dans un *module externe* écrit dans un autre langage, plus efficace. En général seule une petite partie d'un programme a besoin d'être vraiment optimisée (en général moins de 5% du code représente 95% du temps de calcul !).

En outre, il s'avère que l'*on passe souvent plus de temps à écrire un programme qu'à l'utiliser*, ce qui relativise l'importance de la "performance pure" du programme. L'intérêt de python est justement d'*accélérer le développement d'un programme par sa syntaxe simple*, et par sa capacité à réutiliser des librairies existantes.

Les **différences fondamentales par rapport au C++** avec lesquelles il faut se familiariser sont :

— C'est un ***langage interprété*** (en fait, ***"compilé au dernier moment"***), utilisable avec une ***ligne de commande***
— Les "blocs" de code (ex : intérieur d'une boucle) sont identifiés par l'***indentation*** (***nombre d'espaces en début de ligne***), et non par des accolades {...}
— Le ***type des objets*** ('int', 'float', 'string') ***est dynamique***, ç.à.d. qu'il n'a pas besoin d'être déclaré à l'avance et qu'il peut être modifié au cours du programme.

Nota bene : ce didacticiel couvre le langage python version 2.x. Les versions de python plus récentes (3.x) présentent des modifications importantes, mais son usage commence juste (en 2015) à se généraliser pour le calcul scientifique.

2 Installation de Python

Sous Linux : Python est installé sur toutes les distributions, en standard - il est par contre nécessaire d'ajouter des modules tels que scipy, numpy et matplotlib, suivant le type de programme souhaité.

Sous Windows : la distribution standard de python est disponible sur *http ://www.python.org*, mais il est recommandé d'installer la version de python de la société Enthought (Enthought Python = Enthon), qui inclut les modules de calculs scientifique les plus utiles *http ://code.enthought.com/enthon/*. De même la distribution "python(x,y)" peut être utilisée.

Une autre distribution populaire (pour Windows, Mac OSX et Linux) est anaconda.

3 Programmer avec Python

3.1 Ligne de Commande ou éditeur ?

Python peut être utilisé à partir d'une ligne de commande (langage *interprété*). Pour lancer l'interpréteur, taper "`python`" dans une console linux, puis à l'invite de commande ("`>>>`"), taper "`print ''toto''`". Cela donne

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

Dans un autre langage, il aurait fallu (1) écrire le texte du programme dans un fichier, puis (2) le compiler avant (3) de l'éxécuter. En fait il est également possible d'écrire le programme dans un fichier. Pour cela ouvrir un éditeur de texte (par exemple **nedit** ou **kate** sous linux, ou un éditeur de texte pur sous windows), écrire le programme (`print ''toto''`), et le sauvegarder sous le nom *toto.py*. Ensuite, éxécuter le programme avec la commande "`python toto.py`" à partir d'une console Linux (il est également possible d'utiliser "`python -i toto.py`" afin de rester dans l'interpréteur après éxécution du programma). De fait, écrire un programme sur la ligne de commande ou le taper au préalable dans un fichier avant de l'exécuter est ***rigoureusement équivalent***.

Pour l'utilisation en ligne de commande, il est encore plus avantageux d'utiliser ***ipython*** : "`ipython toto.py`" - c'est une version plus interactive de python, qui se souvient des instructions tapées lors de la dernière session ipython (utiliser les flèches haute et basse pour naviguer dans l'historique des commandes), et qui sait "compléter" les commandes (pour utiliser une fonction ou un objet, taper le début du nom et appuyer sur la touche tabulation - le nom sera complété ou différents choix seront proposés).

Enfin, il existe des environnement de développement intégrés pour python : on peut citer *IDLE* (installé par défaut mais assez rudimentaire) *Eric* (très complet, notamment pour la création d'interface graphique), ***SpyDer*** (plus simple, mais a l'avantage d'intégrer une console ipython - je le recommande pour commencer).

Voilà ! Vous savez programmer en Python. Dans la pratique, les tests rapides sont faits avec la ligne de commande, et les programmes longs sont écrits dans un fichier avant d'être exécutés.

Note 1 : pour sortir de l'interpréteur Python, taper `ctrl-d`

Note 2 : le symbole # sert à commencer un **commentaire** en python. Tout ce qui se trouve sur une ligne après un # n'est pas interprété par python :

```
>>> print ‘‘toto’’ # ceci est mon premier programme en python !
toto
```

Note 3 : pour éxécuter des commandes dans un fichier texte depuis l'interpréteur python, il faut utiliser la fonction `execfile` (e.g. `execfile(‘‘toto.py’’)`)

Note 4 : pour pouvoir utiliser des ***accents dans Python***, il faut préciser l'encodage que nous utilisons, en mettant au début du fichier python la ligne suivante :

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

3.2 Aide en ligne

Tous les objets standards (de même que les modules correctement conçus) possèdent une aide en ligne, qui peut ête invoquée avec la fonction `help()`. Par exemple `help(str)`, `help(list)`, permettent d'accéder à l'aide sur les chaînes de caractère et sur les listes. Lorsqu'un module est importé (e.g. '`import math`'), il est possible de lire l'aide correspondant à ce module (e.g. '`help(math)`') ou à une fonction de ce module (e.g. '`help(math.sin)`').

Lorsque ***ipython*** est utilisé, c'est encore plus simple, il suffit de taper la commande suivie de un ou deux points d'interrogations (suivant le détail de la documentation souhaitée), par exemple : `'list?'`.

3.3 Variables et types de données élémentaires

3.3.1 Variables simples

Pour créer une variable, il suffit de lui affecter une valeur :

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

On voit ici une différence importante avec d'autres langages : il n'est pas nécessaire de déclarer *a priori* le **type** de la variable x (entier 'int', nombre flottant 'float', chaine de caractère 'string',...). Pourtant ce type existe bien, si vous demandez ensuite :

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

Python a donc automatiquement décidé que x était un entier (*int = integer*). Essayez avec d'autres initialisations : "`x=1.0`" (nombre flottant ou *float*), "`x=''bonjour''`" (chaine de caractère ou *string*),...

Il est à noter qu'il est possible de **changer le type** de x en lui affectant des valeurs différentes :

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

Bien que cela soit pratique, cela peut entraîner des erreurs de programmation, donc dans la mesure du possible **il faut toujours utiliser des noms de variables explicites** (`age` sera un entier, `nom` une chaîne de caractères, etc...), et **ne pas changer de type en cours de programme**.

Attention : ***majuscules* et *minuscules* sont différenciées par python**, i.e. `mavariable`, `MAVARIABLE` et `MaVariable` sont trois variables distinctes. Il en va de même pour tous les mot-clefs et les fonctions en python.

3.3.2 Nombres complexes

Il est possible de manipuler des nombres complexes en Python, le nombre complexe "i" étant noté `j` :

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

Pour utiliser des fonctions mathématiques sur des complexes, il faudra importer le module `cmath` en plus du module `math` (voir la section sur l'utilisation de modules), ou bien le module `scipy` qui inclut également les tableaux.

3.3.3 Chaînes de caractères

Outre l'attribution simple (`maChaine=''Toto''`), il est possible d'"écrire" une chaîne de caractère de la même manière qu'avec la fonction `printf` classique, c'est à dire en insérant des codes dans la chaîne (`%s` pour une chaîne de caractères,, `%d` pour un entier, `%f` pour un nombre réel, avec éventuellement le nombre de caractère total et après la virgule : `%4.2f` veut dire '*nombre réel représenté sur 4 caractères, avec 2 chiffres après la virgule*'), suivie de % avec entre parenthèses toutes les variables dont les valeurs sont à insérer :

```
>>> age=5
>>> nom="Vincent"
>>> taille=1.73
>>> maChaine="%s a %d ans et mesure %4.2f m"%(nom,age,taille)
>>> print maChaine
Vincent a 5 ans et mesure 1.73 m
```

Il est possible d'extraire des caractères d'une chaîne :

```
>>> print maChaine[2:6]    # extrait les caractères 2 à 6 (exclu)
ncen
```

Par contre il n'est pas possible de modifier de cette manière une sous-chaîne de caractère : `maChaine[2:6]=''tapl''` génère une erreur. Au passage on voit que la numérotation commence à zéro - le premier caractère est `maChaine[0]`.

Il est possible d'additionner des chaînes de caractères :

```
>>> print "Programmation" + " Scientifique"
Programmation Scientifique
```

On peut obtenir la longueur d'une chîne de caractère avec la fonction `len()`.

```
>>> len("toto")
4
```

`help(str)` donne la liste des fonctions utilisables avec des chaînes de caractères.

3.3.4 Listes

Un type particulièrement intéressant est la ***liste*** :

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

```
<type 'list'>
>>> print x[2]
32
```

Attention ! On voit ici que la numérotation des objets de la liste commence à 0 (i.e. si il y a n éléments dans `liste`, ils sont accessibles par `liste[0]`...`liste[n-1]`).

Une *liste* peut contenir des *objets de types différents*, contrairement à un *tableau :*

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

Il est possible de *modifier une liste existante* :

```
MaListe[1]=42          # le 2ème élément de MaListe vaut maintenant 42
MaListe.append(45)     # ajoute l'objet ''42'' à la fin de la liste
MaListe.insert(i,x)    # insère l'élément x à l'indice i dans MaListe
MaListe.reverse()      # inverse l'ordre des éléments de la liste
MaListe.sort()         # trie les éléments de la liste par ordre croissant
```

On peut obtenir la **longueur d'une liste** (le nombre d'objets dans la liste) avec la fonction `len()` : `len(MaListe)`.

Il est possible de **concaténer deux listes** avec l'opérateur + :

```
>>> liste1=[1,"ab",3.5]
>>> liste2=["toto",57]
>>> print liste1+liste2
[1, 'ab', 3.5, 'toto', 57]
```

On peut également **supprimer un élément d'une liste** avec la fonction `del()` :

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

Les listes sont des objets particulièrement importants car les boucles utilisent souvent une itération sur des éléments d'une liste. La liste complète des fonctions utilisables pour une liste peut être obtenue avec l'aide `help(list)`. Il existe en particulier une *fonction de tri* `sort()` : *essayez-la sur une liste de nombres (entiers et réels mélangés).*

3.3.5 Tuple *[Facultatif]*

Un *tuple* est un objet similaire à une liste, la principale différence étant qu'un tuple ne peut pas être modifié comme une liste. C'est une liste d'objets séparés par des virgules ; on peut (mais ce n'est pas obligatoire) encadrer ces objets par des parenthèses pour mieux délimiter le tuple :

```
>>> monTuple=('a',"toto",54.2)
>>> type(monTuple)
```

```
<type 'tuple'>
```
Il est possible d'extraire un ou plusieurs objets d'un tuple :
```
>>> print monTuple[1]
toto
>>> print monTuple[1:3]
('toto', 54.200000000000003)
```

Par contre il n'est pas possible de modifier un objet dans un tuple (`monTuple[1]=''tata''` est une instruction illégale), ou d'ajouter des éléments dans un tuple.
En général on ne crée pas explicitement de variable de type tuple, mais ce sont des variables qui aparaissent de manière temporaire, lorsqu'on veut manipuler ensemble plusieurs objets :
```
>>> a,b=5,3        #création de deux variables en une seule ligne
>>> print a,b
5 3
>>> a,b=b,a         # échange de deux variables
>>> print a,b
3 5
```

3.3.6 Dictionnaires *[Facultatif]*

Un *dictionnaire* est également un type de liste d'objet, mais au lieu de repérer les objets par un indice commençant à 0, les différents objets sont accessible à l'aide d'une *clef*, qui peut être un objet quelconque. (pour les personnes familières avec la librairie standard du C++, c'est l'équivalent de `std::map<>`).
```
>>> fra_deu={"un":"ein","trois":"drei","deux":"zwei"}    # Création
>>> fra_deu["trois"]            # Accès à un élément du dictionnaire
'drei'
>>> fra_deu["quatre"]              # Clef n'existant pas !
Traceback (most recent call last):
 File "<stdin>", line 1, in?
KeyError: 'quatre'
>>> fra_deu["quatre"]="vier"    # Ajout de l'entrée manquante
>>> fra_deu["quatre"]
'vier'
```

Vous pouvez obtenir les différentes fonctions disponibles pour un dictionnaire avec la commande `help(dict)`.

3.3.7 Conversion de variables entre différents types

On peut convertir une variable d'un type à un autre :
```
>>> x=float(1) # nombre réel à partir d'un entier
>>> print type(x)
<type 'float'>
>>> x=float("1.0") # nombre réel à partir d'une chaîne de caractère
>>> print type(x)
<type 'float'>
>>> print x
1.0
```

3.4 Boucles et opérateurs

3.3.1 opérateurs `==`, `and`, `or`, `not`

L'opérateur `==` permet de comparer deux objets :

```
>>> 1==2     # est-ce que 1 est égal à 2?
False
>>> 1==1
True
```
Les autres opérateurs de comparaison sont <, >, <=, >=, !=.

Notez qu'en python tout nombre non nul est interprété comme True, i.e. if(n), où n est un nombre, sera toujours vrai sauf si n est égal à zéro.

L'opérateur `not(x)` renvoie la négation (au sens booléen, vrai/faux) d'une expression :
```
>>> not(26)
False
>>> not(0)
True
```
`and` et `or` permettent de réaliser les opérations logiques usuelles (*attention il ne s'agit pas d'opérations bit à bit*, il ne faut donc utiliser and et or que pour des tests booléens vrai/faux) :
```
>>> 1 and 0
0
>>> 1 and 1
1
>>> 1 or 0
1
```

3.4.2 L'instruction `if...elif..else`

On peut éxécuter une instruction conditonnelle avec if :
```
>>> x=1
>>> if x==2:                 # est-ce que x est égal à 2?
...    print "x est égal à 2"  # notez les espaces en début de ligne (indentation)
... elif x==3:
...    print "x est égal à 3"
... else:
...    print "x n'est pas égal à 2"
...    print "ni à 3"
...
x n'est pas égal à 2
ni à 3
```

Les ***espaces en début de ligne*** (***indentation***) jouent un rôle important : tant que l'espace reste le même en début de ligne, on reste dans le même "niveau" d'éxécution, de même que dans d'autres langages on utilise des accolades {} pour délimiter les instructions à réaliser. On peut ainsi réaliser des *instructions imbriquées* en augmentant le niveau d'indentation :
```
>>> x=12
>>> if (x%2)==0:            # est-ce que x modulo 2 est égal à 0?
...    if (x%3)==0:
...        if (x%7)==0:
...            print "x est multiple de 2,3 et 7!"
...        else:
...            print "x est multiple de 2 et 3, mais pas de 7..."
...    else:
...        print "x est multiple de 2 mais pas de 3"
...else:
...    print "x n'est pas multiple de 2"
```

```
...
x est multiple de 2 et 3, mais pas de 7...
```

3.4.3 La boucle `while`

On peut réaliser une série d'instructions tant qu'une condition est réalisée :

```
>>> x=7
>>> while x>=1:
...     print x,">=1 ! On continue..."
...     x = x-1                 # on pourrait aussi écrire x -= 1
7 >=1 ! On continue...
6 >=1 ! On continue...
5 >=1 ! On continue...
4 >=1 ! On continue...
3 >=1 ! On continue...
2 >=1 ! On continue...
1 >=1 ! On continue...
```

3.4.4 Les boucles `for`

Une boucle for s'écrit de la manière suivante :

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

Comme il est fastidieux d'écrire un grand nombre de valeurs, on peut utiliser la fonction `range()` qui renvoie une liste d'entiers :

```
>>> range(5) # renvoie une liste de 5 entier en partant de 0
[0, 1, 2, 3, 4]
>>> range(2,5) # renvoie une liste d'entiers, de 2 à 5 (exclu)
[2, 3, 4]
>>> for i in range(2,5):
...     print i
...
2
3
4
```

Note : la fonction `range()` crée une liste de nombres qui va servir pour l'itération ; on peut s'en servir pour créer une liste sans que cela soit pour une boucle. Pour une boucle, il vaut en fait mieux utiliser la fonction `xrange()`, qui ne crée pas une 'vraie' liste en mémoire mais seulement un ***itérateur*** sur une liste d'entiers qui permettra à la boucle de se réaliser.

3.4.5 Exercice : décomposition en facteurs premiers

Ecrire une boucle qui, à partir d'un entier n, dresse la liste de ses facteurs premiers en les faisant défiler à l'écran. Optionnellement, les stocker dans une liste. Les opérateurs mathématique standards sont * (multiplication), / (division), % (modulo), et les opérateurs de comparaison sont : ==, <, <=, >, >=.

Note : contrairement au C et au C++, l'opération x%n renvoie bien une valeur dans l'intervalle [0 ;n-1], même si le nombre x est négatif...

3.5 Fonctions

Il est possible d'utiliser des fonctions en Python. Nous avons déjà utilisé `print()` et `type()`. De manière générale, une fonction s'utilise de la manière suivante : `NomDeLaFonction(paramètres)` ou (si la fonction renvoie une valeur) : `VariableRenvoyée=NomDeLaFonction(paramètres)`

Pour définir une nouvelle fonction, il faut utiliser le mot-clef "def" :

```
>>> def MaFonction(nom,age):
...     print nom,"a",age,"ans"        #noter les espaces en début de ligne
...
>>> MaFonction("Obélix",5)
Obélix a 5 ans
```

`nom` et `age` sont les *paramètres* de la fonction ; là encore il n'est pas nécessaire de déclarer le type de ces paramètres. En fait on peut très bien utiliser une chaine de caractère pour age :

```
>>> MaFonction("Obélix","cinq")
Obélix a cinq ans
```

Par contre si la déclaration de la fonction n'impose pas un contrôle sur les types de paramètres, il faut néanmoins qu'ils soient compatibles pour l'éxécution. Essayer :

```
>>> def Addition(a,b):
...     return a+b          # Renvoie une valeur avec ‘‘return’’
...
>>> print Addition(1,2)        #deux entiers
3
>>> print Addition(1,3.0)      #un entier plus un flottant= un flottant
4.0
>>> print Addition(1,"toto")   # entier+chaîne de caractères = erreur !
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")      # deux chaînes de caractères
tatatoto
```

Note : comme pour les boucles c'est l'***indentation*** (le nombre d'espaces en début de ligne) qui définit quelles sont les instructions qui font partie de la fonction. Cela remplace l'utilisation d'accolades {} dans d'autres langages.

Exercice : réutiliser le code écrit pour la décomposition en facteur premiers, et écrire une fonction qui prend comme argument un nombre entier, et renvoie une liste des facteurs premiers de cet entier.

3.6 Portée des objets : variables locales et globales

Comme dans d'autres langages il existe en Python des *variables globales* et des *variables locales*. Ces dernières n'existent qu'au niveau où elles ont été définies. Par défaut, toutes les variables sont locales. Exemple :

```
>>> x=12.3              # Première variable ‘‘x’’
>>> def fonction1():
...     x=1             # Deuxième variable ‘‘x’’
...     print x
...
```

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

Dans l'exemple ci-dessus, les 3 variables ont le même nom mais sont complètement indépendantes. Il est à noter qu'*un niveau d'indentation ne peut pas modifier les variables du niveau au-dessus* (contrairement à ce qui se passe en C/C++). Essayer par exemple :

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

Par contre, il aurait parfaitement été possible de *lire* la variable compte, sans la modifier.

Pour pouvoir utiliser une variable à différents endoits (autrement qu'en la passant comme paramètre à une fonction), il faut utiliser le mot-clef `global` :

```
>>> compte=0
>>> def compteur():
...     global compte
...     compte+=1
...     return compte
...
>>> compteur()
1
```

Note : ***en général, il vaut mieux éviter d'utiliser des variables globales, en passant des variables par paramètre aux fonctions.***

3.7 Exercice

Ecrire une fonction qui prend comme argument deux listes représentant des positions atomiques dans l'espace (e.g. ["C1", 0, 0.5 , 0.7]), et qui renvoie leur distance interatomique avec une ligne du type :

"La distance entre l'atome C1 et l'atome H3 est de 3.5 A"

Pour calculer une racine carrée importer le module de math avec "`import math`", et utiliser "`help(math)`" pour avoir la liste des fonctions disponibles.

Remarque : pour importer un module (tel que `math`), il y a deux manières de le faire, avec (a) "`import math`" ou (b) "`from math import *`". Dans le cas (a), les fonctions du module math s'utiliseront en les faisant précéder de "`math.`" (exemple : "`math.sin(math.pi/3.0)`"). Dans le cas (b), il n'est pas nécessaire d'utiliser le préfixe, i.e. python comprendra directement "`sin(pi/3.0)`". A priori, la première

notation n'est à utiliser que si il y a un risque de conflit de noms entre fonctions, si par exemple vous souhaitez redéfinir la fonction sin()...

4 Classes et Objets

4.1 Présentation

Historiquement se sont succédées plusieurs manières de programmer :

Dans l'approche ***procédurale***, un seul programme exécute séquentiellement une suite d'instructions. Ce fonctionnement convenait bien à des programmes courts, écrits par une seule personne .

Avec des programmes plus longs (et l'intervention de plusieurs personnes, en parallèle ou se suivant dans le temps), il est devenu nécessaire de séparer les différentes tâches du programme avec plusieurs fonctions : c'est l'approche ***structurée***.

L'approche la plus moderne est la ***Programmation Orientée Objet*** (POO) : les données traitées dans le programme sont stockée dans des "objets", qui comprennent à la fois les données ainsi que des fonctions qui servent à utiliser l'objet. Par exemple un objet Molécule pourra contenir une liste d'atome, leurs positions, mais également des fonctions pour calculer l'énergie, la charge totale, le moment dipolaire, ... de la molécule. En outre on peut créer des hiérarchies d'objets, qui "héritent" des propriétés de leur ancêtres, etc...

Attention *: bien que la programmation Objet soit la plus moderne des approches, les deux autres restent parfaitement valables, en particulier pour de petits projets. Rien ne sert de créer des objets pour des calculs ne prenant que quelques lignes ! ! !*

4.2 Exemple

Construisons par exemple une classe d'objets de type "polygone" :

```
>>> class Polygone:             # Déclaration de la classe
...    circ=1.0                   # Un attribut de la classe, pour stocker la circonférence
...    def Circonference(self):  # fonction renvoyant la valeur de la circonférence
...         return self.circ
...
>>> a=Polygone()                  # Création d'un objet de type Polygone
>>> print a.circ                  # Que vaut sa circonférence (accès direct) ?
1.0
>>> print a.Circonference()       # Que vaut sa circonférence (par la fonction) ?
1.0
>>> a.circ=12.3                   # Modification de la circonférence
>>> print a.Circonference()       # Vérification de la modification
12.3
```

Dans la déclaration ci-dessus, on voit aparaître le mot-clef "`self`" : c'est ce qui désigne, dans le corps de la classe, l'objet lui-même. Par exemple, "`self.circ`" veut dire "*la variable circ dans l'objet Polygone où le code est exécuté*" (si on est dans le Polygone "a", cela sera a.circ).

4.3 Attributs

Un ***attribut*** est une variable (ou donnée) membre stockée dans un objet, comme la variable "`circ`" ci-dessus. Il peut y en avoir autant que l'on veut. Il est même possible (contrairement au C++) d'***ajouter des attributs après avoir déclaré une classe*** : à la suite du code ci-dessus, essayer :

```
>>> Polygone.surface=45.6   # Ajout de la variable membre 'surface'
>>> b=Polygone()
```

```
>>> print b.circ,b.surface,a.circ,a.surface
1.0 45.6 12.3 45.6
```

L'attribut ajouté est accessible dans tous les objets Polygone, même "a" qui a été créé avant.

4.4 Fonctions membres

Une fonction membre est une fonction qui est définie dans la déclaration de la classe, et a accès aux attributs de la classe, comme la fonction `Circonférence(self)` définie dans l'objet `Polygone`. Elle prend au moins un argument, "`self`", qui désigne l'objet lui-même (c'est l'équivalent du pointeur `this` en C++) et permet à la fonction d'accéder aux attributs. Là encore, on peut utiliser autant de fonctions membre que nécessaire.

Il est en général recommandé (en POO) de ne pas accéder directement aux attributs de l'extérieur de la classe, mais plutôt d'utiliser des fonctions membre pour accéder à ces attributs. L'idée est que la manière dont sont stockées les données peut varier d'une classe à l'autre, alors que les fonctions membres resteront les mêmes.

Par exemple dans la classe `Polygone`, la circonférence est stockée sous la forme d'un attribut `circ`, mais dans la classe `Carre` (voir plus loin), il n'y a pas d'attribut `circ`, puisque la circonférence peut être calculée direcement à partir du coté... Par contre les deux classes ont la même fonction `Circonference()`, mais qui procèdent de deux manières différentes pour renvoyer le résultat.

Dans une fonction membre, il est très important de distinguer les variables membre (celles précédées de "`self.`") des variables locales. Les variables membres sont les seules qui ne sont pas détruites à la fin de l'éxécution d'une fonction membre et qui permettent de 'stocker' des informations dans l'objet, et donc de les passer de fonction en fonction.

```
>>> class Objet:
...    var1=1.0       # variable membre déclarée à la base de la classe
...    def __init__(self): # Constructeur
...       varlocale=1.0    # Variable locale
...       self.var1=2.4    # Changement de la valeur de la variable membre self.var1
...       self.var2="toto" # Ajout d'une seconde variable membre self.var2
...    def Fonction1(self):
...       print self.var1,self.var2
...       varlocale="tata" # Cette variable est locale et n'a donc rien à voir avec
celle
...
>>> a=Objet()
>>> a.var1
2.3999999999999999
>>> a.var2
'toto'
>>> a.Fonction1()
2.4 toto
>>> a.varlocale     # erreur car varlocale n'est pas une variable membre
Traceback (most recent call last):
 File "<stdin>", line 1, in?
AttributeError: Objet instance has no attribute 'varlocale'
```

Pour chaque objet, il faut avoir le réflexe "`self`.nomVariable", de manière à systématiquement distinguer les variables membre des variables locales.

4.5 Fonctions membres spéciales *[Facultatif]*

Certaines fonctions membre sont prédéfinies dans python car elles permettent d'effectuer des opérations prédéfinies. Le nom de ces fonctions *commence et finit par* ***deux*** *traits soulignés*.

4.5.1 Constructeur `__init__`

Un constructeur est une fonction membre d'un type spécial, qui sert à ***initialiser un nouvel objet***, en général en utilisant des arguments :

```
>>> class Polygone:
...     circ=1.0
...     def __init__(self,circ0):     # Prend comme argument la circonférence initiale
...         self.circ=circ0             # Initialise la circonférence
...     def Circonference(self):
...         return self.circ
...
>>> a=Polygone(10.5)                   # Crée un polygone de circonférence 10.5
>>> print a.Circonference()
10.5
```

4.5.2 Destructeur `__del__`

Le destructeur est la fonction qui est appelée lors de la destruction de l'objet. Il ne prend qu'un argument (self). Il peut servir à 'nettoyer' des données créées par l'objet.

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

Ce sont les opérateurs (+, -, *, /, **) qui permettent des opérations arithmétiques. Ils prennent deux arguments, self et un argument désignat le second membre de l'opération :

```
>>> class Polygone:
...    circ=1.0
...    def __add__(self,poly):   # addition
...       somme=Polygone()       # objet à renvoyer
...       somme.circ=self.circ+poly.circ
...       return somme
...
>>> a=Polygone()
>>> b=Polygone()
>>> b.circ=3.5
>>> print (a+b).circ      # Circonférence de la somme de a et b
4.5
```

Redéfinir ces opérateurs standard peut être intéressant, mais il ne faut jamais le faire en modifiant la signification originale de l'opérateur (par exemple en calculant une différence en ré-écrivant une fonction `__add__`).

4.6 Héritage *[Facultatif]*

L'héritage permet de créer des classes qui héritent tout ou partie des propriétés de la ou les classes parentes. On peut ainsi dériver une classe rectangle de la classe Polygone :

```
>>> class Polygone:
...     def __init__(self,circ0=2.1):     # On utilise ici un argument par défaut
...         self.circ=circ0
...     def Circonference(self):
...         return self.circ
```

```
...
>>> class Rectangle(Polygone):        # Hérite de la classe Polygone
...     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.Circonference()  # La fonction Circonference() est héritée de Polygone
14.0
```

Il est possible de dériver une autre classe, et également de remplacer la fonction Circonference() de la classe de base par une nouvelle fonction :

```
>>> class Carre(Rectangle):     # Hérite de la classe Rectangle
...     def __init__(self,x0):
...         self.x=x0
...     def Circonference(self):       # On réécrit la fonction Circonference()
...         return 4*self.x
...
>>> a=Carre(4.0)                   # Carré de coté 4.0
>>> print a.Circonference()
16.0
```

L'héritage peut être très utile lorsque l'on génère de multiples classes ayant des propriétés similaires. Par contre il ne faut pas en abuser, cela peut amener à une programmation inutilement complexe ! Ce n'est en général utile que pour des projets de taille suffisament grande (> 1000 lignes), mais cela peut être aussi utile pour ré-utiliser du code (provenant par exemple d'une librairie développée par quelqu'un d'autre), en ajoutant quelques fonctions à une classe pré-existante.

4.7 Attention : opérateur "==", différence entre égalité et identité, copie et référence... *[Facultatif]*

Utilisons les classes définies précédemment, et essayons de comparer des objets :

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

Que s'est-il passé ? Clairement les trois carrés sont égaux et on s'attendrait à lire `True True True` comme réponse... En fait l'opérateur == n'a pas comparé les *valeurs* (est-ce que les carrés sont *égaux*), mais les *adresses*, pour tester *l'identité* des carrés. Ce qui se comprend mieux si on teste le code suivant :

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

Ce qui se passe donc lorsqu'on écrit "`c=a`" n'est pas la création d'un nouveau carré distinct de `a`, mais un carré dont l'attribut "`x`" pointe vers la même case mémoire que celle de `a`... Ce qui fait que `c` et `a` représentent rigoureusement le même objet.

L'idée derrière ce comportement qui peut souvent induire en erreur est de minimiser l'espace mémoire en faisant par défaut un *passage par référence* (i.e. en pointant la nouvelle variable vers la même case que la variable originelle) plutôt qu'une copie.

Afin d'éviter cela, il est possible d'effectuer une véritable copie avec **la fonction** `copy.copy()` **ou** `copy.deepcopy()` :

```
>>> import copy
>>> a=Carre(4.0)
>>> b=Carre(4.0)
>>> c=copy.copy(a) # en fait il vaut mieux utiliser c=copy.deepcopy(a)
>>> print a==b, a==c, b==c
False False False
```

4.8 Exercice

Créer une classe Atome qui a 4 attributs : le nom, et les 3 coordonnées x,y,z, avec des fonctions permettant de créer la classe, et d'accéder aux 4 attributs (Nom(), X(), Y() Z()).

Créer une fonction calculant la distance entre deux atomes.

Ensuite créer une classe Molécule, qui a comme attribut une liste d'atomes, et qui a des fonctions membre permettant de :

— Ajouter un atome
— Accéder à un des atomes
— Afficher la liste de toutes les distances interatomiques dans la molécule.

5 Lecture et écriture de fichiers

On crée un objet fichier Python avec la fonction open() :

```
>>> MonFichier = open(''NomDuFichier'','r')
```

Le mode peut être 'r' (read, en lecture seule), 'w' (en écriture seule)... D'autres modes (mixte lecture/écriture, etc...) existent.

La lecture des fichiers texte (i.e. non binaires) se fait avec les fonctions read(), readline() et readlines() :

```
>>> MonFichier.read()     # Lit la totalité du fichier sous forme d'un chaîne de caractères
>>> MonFichier.readline()    # Lit la ligne suivante du fichier, sous forme d'une chaîne de caractères
>>> MonFichier.readlines()    # Lit toutes les lignes du fichier sous forme d'une liste de chaînes de caractères
```

Pour lire une liste de nombres contenu dans une ligne, il suffit de lire cette ligne avec la fonction readline(), puis de séparer les nombres avec la fonction split(). Par exemple :

```
>>> f=open("tmp0",'r')
>>> ligne=f.readline()       # lecture d'une ligne
>>> print ligne             # Ecriture de la ligne sous forme d'une chaîne de caractères
Toto 1.8 9.2
>>> print ligne.split()    # On sépare les éléments -> liste
['Toto', '0.2', '9.2']
>>> print float(ligne.split()[1]) # Conversion en nombre réel
0.2
>>> f.close()      # referme le fichier
```

`help(file)` permet de lister l'aide sur l'utilisation des fichiers.

5.1 Exercice

Créez un fichier avec une liste d'atomes (un nom et 3 coordonnées par ligne). Rajoutez une fonction LireFichier(self, NomFichier) dans votre classe Molecule, qui lit la liste des atomes. Et ensuite listez toutes les distances interatomiques...

Ensuite, ajouter une fonction qui peut écrire dans un fichier la liste des atomes... et vérifiez que vous arrivez à la relire ensuite !