# Notes python, numpy, matplotlib

*Bruno Pinçon* (Bruno.Pincon@univ-lorraine.fr)

Télécom Nancy (ex ESIAL) et Institut Elie Cartan Lorraine

Ces notes sont extraites de notes de cours d'un stage LIESSE effectué en mai 2014 dont le but était d'initier des professeurs de classes préparatoires au langage python.

# Table des matières

# Chapitre 1

# Introduction

Le langage python est utilisé depuis peu en classe préparatoire pour initier les élèves à l'informatique et sans doute leur servir à illustrer des notions de mathématiques, de physique, de chimie, etc... C'est sans doute un assez bon choix :

— C'est un langage très complet et multiplateformes [1] mais qui reste assez simple. Il permet, assez facilement, d'écrire des petits (et gros) programmes, de les mettre au point et de les utiliser sans avoir à être un spécialiste du langage (ce qui est le cas des 2 encadrants) et de l'informatique. Attention python reste quand même plus compliqué que scilab/matlab/octave mais il a un spectre applicatif beaucoup plus large que ces 3 autres langages qui sont plutôt dédiés au calcul scientifique.

— Il est très utilisé dans le monde académique mais commence aussi à prendre de l'importance dans les entreprises.

— Python, numpy, matplotlib et scipy sont libres sous licence BSD. On peut ainsi les utiliser gratuitement mais de plus, la licence BSD permet d'intégrer ces outils pour construire des logiciels commerciaux sans quasiment aucune contrainte autre que celle de préciser quelque part que vous utilisez ces composants logiciels dans votre code.

Actuellement il y a deux versions du langage en circulation, la "2.7" et la "3.x" et il est important de connaître les différences principales entre ces deux versions. La documentation officielle se trouve ici :

— pour la version 2.7 : `http://docs.python.org/2/index.html`

— pour la version 3.x : `http://docs.python.org/3/index.html`

La caractéristique essentielle de python est qu'il s'agit d'un langage interprété : il n'y a pas de transformation du code en langage machine. Il s'ensuit qu'un langage interprété est souvent très souple, que les variables ne sont pas (en général) déclarées, le type d'une variable est induit à l'initialisation (affectation) et peut changer de type via une autre affectation. Par contre un programme écrit en python sera généralement plus lent que s'il avait été écrit en C.

Ainsi pour faire du calcul scientifique de façon assez efficace (où l'algèbre linéaire appliquée joue un grand rôle), les langages interprétés se doivent d'avoir des "objets" tableaux munis d'opérations optimisées et c'est le cas de scilab/matlab/octave où cette préoccupation est à la base même de leur conception. En python c'est différent, les tableaux ne font pas partie des objets de base et sont ajoutés au langage via le module numpy. Malgré tout, muni des outils numpy, matplotlib (et scipy), on peut écrire en python des codes scientifiques presque aussi facilement qu'avec scilab/matlab/octave.

1. Le même code fonctionnera sur Linux, Windows, (Mac) OS X sans changements (ou parfois avec des changements mineurs) et ceci pour des codes qui vont afficher du graphique, interagir avec l'utilisateur, etc.

# Chapitre 2

# Premiers pas avec python

## 2.1 Principes de travail, quelques mots sur Spyder

Il existe plusieurs façons de travailler avec python :

— une possibilité, que l'on pourrait qualifier d'interactive, consiste à entrer directement des instructions dans (la fenêtre de commande de) l'interprète ;

— une autre consiste à écrire ces mêmes instructions dans un fichier (avec un éditeur de texte) puis de les exécuter par l'interprète python (ce qui peut se faire de diverses manières) ;

— on peut conjuguer les deux.

Un défaut de la première approche est qu'il n'est pas très pratique de ré-éditer et modifier les commandes que vous avez déjà entrées dans l'interprète et on ne l'utilisera qu'au tout début pour découvrir quelques types de base de python. Par contre elle reste intéressante conjuguée avec l'approche par fichiers :

— dans le cadre de la mise au point d'un programme car vous pouvez tester directement si tel ou tel bout de code fonctionne bien avant de l'écrire effectivement à la suite des instructions de votre fichier ;

— pour l'exploration de données : typiquement lorsque l'on a effectué certains calculs ou traitements en exécutant un fichier de commandes python (et que l'on a pas quitté l'interprète), on peut vouloir ensuite entrer des instructions directement dans l'interprète qui vont agir sur les variables/objets que l'on a obtenus précédemment.

Bref pour travailler avec python il suffit de l'interprète python et d'un éditeur de texte [1] et ces deux programmes peuvent être complètement indépendants l'un de l'autre. Cependant pour simplifier l'interaction entre l'édition, l'exécution et la mise au point de programmes python nous allons utiliser l'environnement de développement intégré Spyder qui est complet, multiplateforme et permet de progresser rapidement avec python en occultant les détails concernant l'exécution des fichiers de commande python.

Lancer Spyder peut se faire par un menu ou par une commande dans un terminal. Par défaut on obtient une nouvelle fenêtre munie d'une barre de menu en haut, d'une sous-fenêtre à gauche pour l'édition des fichiers et de deux sous-fenêtres à droite, celle du dessous correspondant à la fenêtre de commande python. Des explications orales seront données au fur et à mesure...

1. de préférence muni de facilités adaptées à l'écriture de codes python.

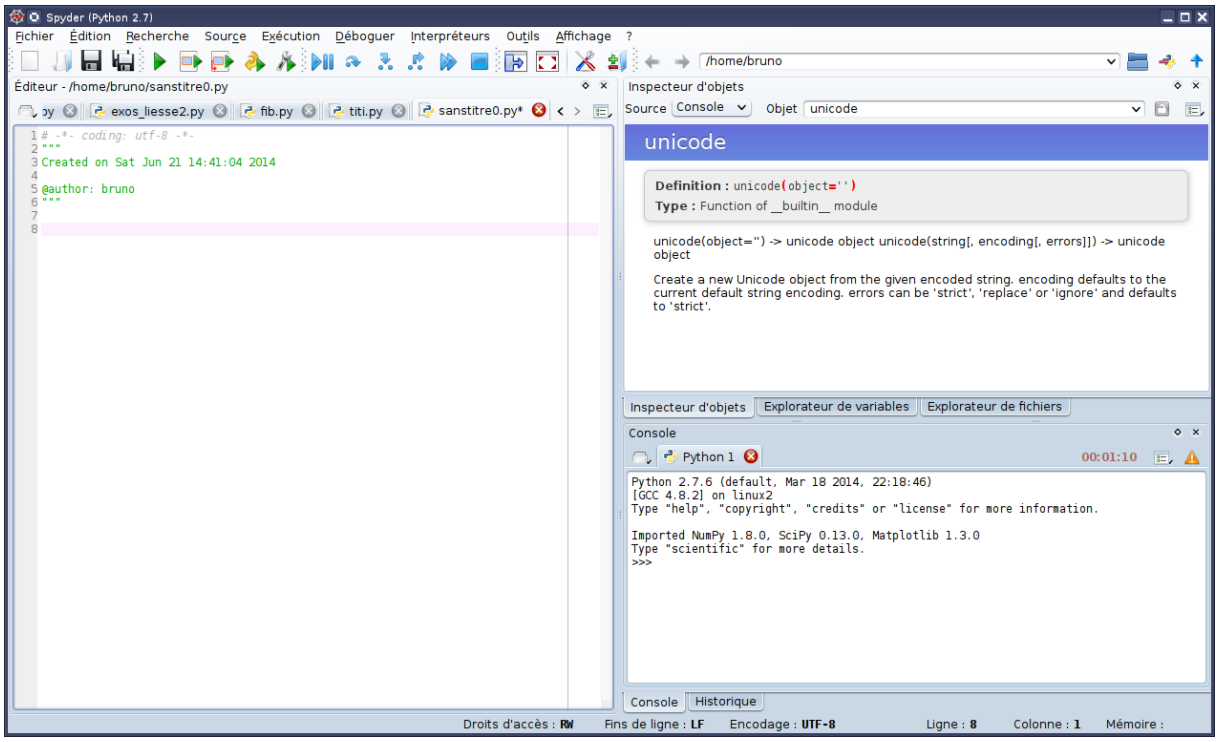



FIGURE 2.1 – La fenêtre Spyder.

## 2.2 Quelques types de base

Le but de cette partie est de vous faire découvrir quelques types de base de python en entrant directement des commandes dans l'interprète [2]. Vous devez rentrer les instructions proposées [3]. Quelques explications sont données au fur et à mesure, certaines en tant que commentaires python (tout texte après #) : inutile de les taper !

Avec python l'invite de commande prend la forme `>>>` vous entrez le texte à la suite **sans caractères blancs** supplémentaires entre l'invite et le début de la commande [4].

### 2.2.1 Les entiers

Les premiers objets que nous allons voir sont les entiers. Il existe deux types d'entiers dans python (version $< 3$), des entiers "usuels" avec un stockage sur 4 ou 8 octets et des entiers arbitrairement longs. Dans les versions 3.x ces deux types ont été transformés en 1 seul.

Néanmoins dans les versions 2.7.x, dès qu'une opération conduit à un résultat entier qui déborde des entiers "usuels" on obtient automatiquement un entier long. On peut ainsi ne pas se préoccuper de ces problèmes de débordement.

Notez que quand on évalue une expression python sans affecter l'objet résultat à une référence *dans la fenêtre de commande*, un affichage de l'objet est effectué. Ce mécanisme nous sert ci-après et évite d'utiliser systématiquement [5] `print`.

2. La fenêtre en bas à droite : vous pouvez la redimensionner en jouant avec la souris.
3. Vous pouvez bien sûr aussi entrer vos propres commandes pour vérifier que vous avez compris mais ne donnez pas les mêmes noms car certains objets sont réutilisés quelques lignes plus loin.
4. Ceci provient du fait que l'indentation est utilisée pour délimiter la portée de certaines constructions (fonction, boucle for, test if, etc...).
5. Néanmoins dans un script il faudra utiliser `print` pour afficher les variables.

```
>>> a = 5
>>> type(a)
<type 'int'>
>>> print(a)    # ou simplement a
5
>>> b = 11      # un autre entier
>>> a+b         # addition (pas d'affectation => affichage du résultat sans utiliser print)
16
>>> a-b         # soustraction
-6
>>> a*b         # multiplication
55
>>> a/b         # division entière. Attention avec python 3.x  il faut utiliser a//b
                # mais avec python 2.7 vous pouvez déjà utiliser //
0
>>> b/a
2
>>> b%a         # modulo
1
>>> q,r = divmod(b,a)  # à la fois le quotient et le reste (deux arguments sont retournés)
>>> q
2
>>> r
1
>>> c = 120     # encore un autre entier
>>> c**20       # conversion automatique en entier arbitrairement long
383375999244747512217600000000000000000000L
```

Différences sur la division entière entre python 2.7 et python 3.x : si `a` et `b` sont deux entiers alors :

- en python 2.7, `a/b` et `a//b` correspondent à la division entière, ainsi les deux expressions arihmétiques `2/5` et `2//5` donnent 2 ;
- en python 3.x, `a/b` est la division usuelle et `a//b` la division entière, `2/5` s'évalue à `2.5` et `2//5` à 2.

Pour éviter les pièges entre 2.7.x et 3.x pour la division avec les entiers :

- Pour une division entière toujours utiliser l'opérateur `//` (code portable entre les deux versions)
- En python 2.7.x, si vous voulez une division "réelle" entre deux opérandes entiers, deux solutions :
  - L'un des opérandes doit être converti en flottant au lieu d'écrire `a/b` on écrit `float(a)/b`. C'est pas beau mais le code est portable entre les deux versions.
  - Plus élégant vous utilisez l'instruction magique `from __future__ import division` qui donne le comportement de la division des versions 3.x dans les versions 2.7.x.

### 2.2.2 Les nombres flottants

Les nombres flottants permettent d'approcher les nombres réels sur ordinateur. Quelques explications seront données oralement (l'essentiel est résumé dans l'annexe A).

Python repère un flottant dès que le nombre comporte un point et/ou une partie avec un exposant (introduite sous la forme habituelle c'est à dire avec un `e` suivi de l'exposant comme dans `4.55e-6`). En fait python utilise ce que la tradition appelle des "doubles" même si le type s'appelle float.

```
>>> x=0.2
>>> type(x)
<type 'float'>
>>> print(x)  # format par défaut insuffisant...
0.2
>>> print(format(0.2,"24.17e")) # on découvre que 0.2 n'a pas été stocké exactement...
 2.00000000000000011e-01
>>> import sys    # import du module sys (on dispose maintenant de ses fonctionnalités)
>>> M=sys.float_info.max  # plus grand nombre flottant (mis dans la variable M)
>>> print(M)
1.79769313486e+308
>>> m=sys.float_info.min  # plus petit nombre flottant normalise (mis dans la variable m)
>>> print(m)
2.22507385851e-308
>>> eps = sys.float_info.epsilon # en fait deux fois le epsilon machine du cours
>>> print(eps)
2.22044604925e-16
>>> Inf = 2*M             # fabrication du nb flottant spécial Inf (mis dans la variable Inf)
>>> print(Inf)
inf
>>> Nan = Inf - Inf       # fabrication du nb flottant spécial Nan (mis dans la variable Nan)
>>> print(Nan)
nan
```

Ici pour obtenir les 3 nombres caractéristiques des flottants il faut importer le module `sys` ce qui s'obtient par la commande[6] `import sys`. On dispose alors des objets[7] définis dans ce module à condition de les préfixer[8] par `sys.`.

Dans les exemples ci-dessus, on utilise l'objet `float_info` du module `sys` et on s'intéresse aux "attributs" (propriétés) `max`, `min` et `epsilon` de cet objet `float_info`.

Les attributs d'un objet sont référencés en utilisant la syntaxe[9] *ref_objet*`.`*nom_attribut* et sont eux-même des objets (les attributs précédents sont des flottants).

Les modules sont un aspect important du langage nous y reviendrons par la suite. Les fonctions usuelles ainsi que certaines constantes sont disponibles en utilisant le module `math` :

```
import math
dir(math)          # voir ce qui est disponible (juste les noms : pas de doc associée)
help(math)         # avec help vous avez une doc minimale des fonctionnalités du module
help(math.expm1)   # doc de la fonction expm1 du module math
math.pi            # approximation de Pi en double
math.sin(math.pi)  # n'est pas exactement 0 donc
```

### 2.2.3 Les chaînes de caractères

Python est particulièrement riche en ce domaine mais nous nous contenterons du minimum.

Avec les entiers les chaînes de caractères constituent l'une des différences importantes entre les versions "2.7" et "3.x" (sauf si on ne considère que des chaînes comportants des caractères ascii pur). En effet en "3.x" ce type correspond à l'unicode et vous permet d'écrire a priori des

6. Le fichier (module) `sys.py` est alors exécuté.

7. C'est à dire des variables (entières, flottantes, chaînes de caractères, listes, ...) mais aussi des fonctions.

8. Nous verrons ultérieurement différentes possibilités (changement du préfixe, importation de certains objets du module, etc...).

9. Nous verrons plus loin que l'on peut aussi appliquer des "méthodes" sur les objets avec une syntaxe assez voisine *ref_objet*`.`*nom_methode*`(`*arg1, arg2, ...*`)` qui correspond à un appel de fonction de la forme *nom_methode*`(`*ref_objet, arg1, arg2, ...*`)`.

caractères de toutes les langues de la terre. En "2.7" les chaînes de caractères sont des séquences d'octets : il est possible d'utiliser d'autres caractères que ceux du jeu de l'ascii pur (comme des caractères accentués) mais ceux-ci utiliseront en général plus d'un octet. Ainsi la fonction `len` qui compte le nombre d'élément d'une séquence (et donc ici du nombre d'octets en "2.7") n'a pas forcément le résultat attendu (le nombre d'octets pouvant alors être différent du nombre de caractères). De même pour la désignation du i ème élément : `mot[i]` donnera le $i+1$ ème octet de `mot` et pas forcément son $i+1$ ème caractère. Idem l'extraction de tranches se fait au sens des octets et pas des caractères. L'unicode est bien disponible avec la "2.7" mais c'est un autre type.

```
>>> s1 = "toto"           # on peut utiliser s1 = 'toto'
>>> s2 = "r"              # autre chaîne
>>> s1+s2                 # concaténation
'totor'
>>> s1[0]                 # extraction du premier caractère de s1
't'
>>> s1[0] = "z"           # erreur: les chaînes sont non modifiables
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s1*3                  # répétition
'tototototototo'
>>> len(s1)               # longueur de la chaîne
4
>>> s3 = "àéù"            # chaîne avec accents
>>> print(s3)
àéù
>>> len(s3)               # en 2.7 vous obtiendrez le nb d'octet (ici 6), en 3.x le nb de caractères.
6
```

Il existe d'autres types de chaînes de caractères (comme les chaînes "brutes") qui peuvent être utiles dans différents contextes.

### 2.2.4 Les listes

Nous terminons ce petit panorama des types de base avec une introduction aux listes.

Les listes servent à faire des collections ordonnées d'objets de type quelconque. Comme les chaînes, les listes sont indicées à partir de 0. Contrairement aux chaînes, les listes sont des objets modifiables.

1. Pour créer une liste il suffit d'écrire ses éléments entre crochets [10] et de les séparer par des virgules :

   ```
   >>> L = [ "a", 1, 3.1416, True ] # le dernier élément est un booléen
   >>> print(L)
   ['a', 1, 3.1416, True]
   ```

2. Tout comme les chaînes de caractères, les listes sont indicées à partir de 0 :

   ```
   >>> L[0]
   'a'
   >>> L[2]
   3.1416
   >>> len(L)                      # nb d'éléments de la liste
   4
   >>> LL = [0,"4"]                # une autre liste
   ```

10. Il y a aussi des objets appelés tuples analogues aux listes mais non modifiables ; un tuple est construit comme une liste en remplaçant les crochets par des parenthèses, exemple : `T = ( "a", 1, 3.1416, True )`.

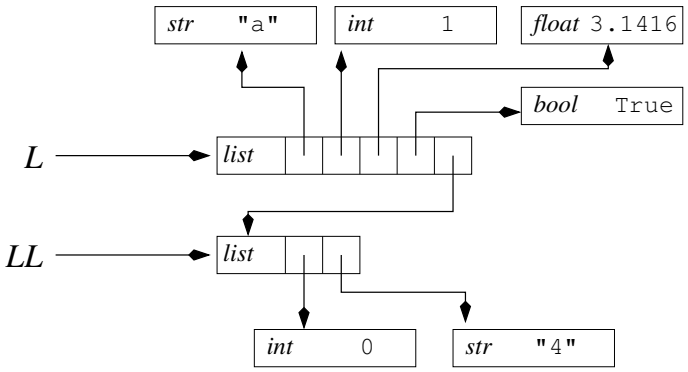
3. On peut ajouter un élément supplémentaire en fin de liste avec la méthode `append` :

```
>>> L.append(LL)            # ajout d'un nouvel élément (qui est une liste)
>>> print(L)
['a', 1, 3.1416, True, [0, '4']]
>>> L.append("toto")            # encore un nouvel élément
>>> print(L)
>>> L.append("titi")            # et encore un autre
>>> print(L)
```

4. et détruire un élément avec l'instruction `del` :

```
>>> del L[5]
>>> print(L)  # "toto" est détruit et "titi" devient l'élément en position 5
['a', 1, 3.1416, True, [0, '4'], 'titi']
>>> del L[5]
>>> print(L)
```

5. Attention les noms (ici `L` et `LL`) sont des références sur les objets et vont permettre de les manipuler et les modifier. Le schéma ci-dessous permet d'avoir une vue d'ensemble et de comprendre les instructions suivantes :




Ainsi pour modifier la liste référencée par `LL` on peut aussi utiliser `L[4]` :

```
>>> LL[0] = "gag"           # modification du 1 er élément de LL
>>> LL                      # OK
['gag', '4']
>>> L                       # L est aussi modifiée
['a', 1, 3.1416, True, ['gag', '4']]
>>> L[4][1] = 912           # autre modification
>>> LL                      # LL est aussi modifiée
['gag', 912]
```

6. La fonction `range` crée des listes particulières qui seront très utiles pour piloter les boucles `for` :

```
>>> c = range(4)       # equivalent à range(0,4,1)
>>> c
[0, 1, 2, 3]
>>> c = range(2,4)     # equivalent à range(2,4,1)
>>> c
[2, 3]
>>> c = range(2,8,2)
>>> c
[2, 4, 6]
>>> c = range(2,7,2)
>>> c
[2, 4, 6]
```

```
>>> c = range(4,0,-1)
>>> c
[4, 3, 2, 1]
>>> c = range(4,-1,-1)
>>> c
[4, 3, 2, 1, 0]
```

`range(deb,fin,inc)` construit donc une liste d'entiers partant de `deb` en incrémentant de `inc` tant que l'on arrive pas (et ne dépasse pas) [11] pas `fin`. Selon ces valeurs la liste peut être vide (`range(1,1,1)` retourne une liste vide).

Avec python 3.x `range` ne construit plus explicitement une liste [12] mais un objet de type "itérateur" permettant une meilleure efficacité pour les boucles par optimisation mémoire [13].

7. Les listes, les tuples, les chaînes de caractères (et d'autres) sont des objets appelés séquences sur lesquels un ensemble identique d'opérations, fonctions ou méthodes peuvent s'appliquer comme la fonction `len` ou la sélection d'un élément avec la syntaxe *ref_objet*`[`*indice*`]`. Sur les séquences il est possible de sélectionner un ensemble "régulier" d'indices appelé "tranche" en utilisant la syntaxe *ref_objet*`[`*deb*`:`*fin*`:`*inc*`]` qui sélectionne le même ensemble d'indices que la fonction `range(deb,fin,inc)` précédente. Il y a aussi quelques raccourcis, par exemple la partie `:`*inc* peut être omise et l'incrément est alors égal à 1. Quelques exemples :

```
>>> L = [ "a", 5, 2, "toto", True, [2, "b"]]  # une nouvelle liste
>>> L[2:6]     # création d'une copie de L avec les elts 2,3,4,5
[2, 'toto', True, [2, 'b']]
>>> L[2:]      # même effet (fin non précisé et inc(=1) > 0 => fin=len(L)=6)
[2, 'toto', True, [2, 'b']]
>>> L[0:4]     # création d'une copie de L avec les elts 0,1,2,3
['a', 5, 2, 'toto']
>>> L[:4]      # idem (deb non précisé et inc(=1) > 0 => deb=0)
['a', 5, 2, 'toto']
>>> L[0:6:2]   # inc donné different de 1 => indices 0, 2, 4
['a', 2, True]
>>> L[::2]     # idem
['a', 2, True]
>>> L[5:0:-1]  # indices 5, 4, 3, 2, 1 (fin n'est jamais atteint)
[[2, 'b'], True, 'toto', 2, 5]
```

Arrêtons nous sur le dernier exemple : si on voulait créer une copie "renversée" de `L` il semble qu'on puisse utiliser `L[5:-1:-1]` mais cela ne fonctionne pas car le bas ou le haut de tranche (cad deb et fin) peuvent être donnés à l'aide d'un entier négatif $k < 0$ qui désigne alors l'élément en position $n + k$ (pour une liste à $n$ éléments). Il faut utiliser :

```
>>> L[5::-1]     # ou encore L[::-1] car deb non précisé et inc < 0 => deb = n-1
[[2, 'b'], True, 'toto', 2, 5, 'a']
```

8. L'opérateur `in` permet de connaître l'appartenance d'un objet à une liste [14] :

```
>>> 'a' in L
True
>>> 'b' in L
```

11. Suivant le sens donné par `inc`.
12. Si vous voulez vraiment obtenir une liste avec python 3.x il faut utiliser `list(range(deb,fin,inc))`.
13. En gros l'objet itérateur ne contient que les 3 entiers et il sait comment générer l'ensemble des entiers sous-entendu.
14. Il permet aussi de savoir si une chaîne de caractère est une sous-chaîne d'une autre chaîne de caractères, par exemple `"oo" in "foo"` retournera vrai.

```
False
>>> [2,'b'] in L
True
```

9. Enfin pour connaître d'autres caractéristiques des listes :

```
>>> dir(list)   # juste les noms (des fcts, méthodes,..)
>>> help(list)  # une doc minimale pour chaque fct, méthode,...
```

**Complément sur les listes (à passer en première lecture)**

Pour copier la liste `L` on peut utiliser une extraction avec une "tranche" désignant l'ensemble des éléments (`L[:]`). Le même effet peut être obtenu avec la fonction `copy` du module `copy` :

```
>>> L = [ "a", 5, 2, "toto", True, [2, 'b']]  # si vous avez perdu L
>>> import copy
>>> LL = copy.copy(L)
>>> LL
['a', 5, 2, 'toto', True, [2, 'b']]
```

mais attention, avec `LL = L[:]` ou `LL = copy.copy(L)`, seuls les éléments non modifiables de la liste sont vraiment dupliqués : le sixième élement (i.e. en position 5) de `L` et de `LL` sont des références sur la même liste `[2, 'b']` :

```
>>> LL[0] = 'b'
>>> L    # L[0] est non modifié
['a', 5, 2, 'toto', True, [2, 'b']]
>>> LL[5][0] = 2.5     # L[5][0] EST AUSSI MODIFIÉ !
>>> LL
['b', 5, 2, 'toto', True, [2.5, 'b']]
>>> L
['a', 5, 2, 'toto', True, [2.5, 'b']]
```

En cas de besoin il est possible de dupliquer complètement un objet avec la fonction `deepcopy`.

## 2.3 Eléments de programmation : modules, fonctions, tests et boucles

### 2.3.1 Les modules et les fonctions

Les modules sont en quelque sorte les unités de programmation en python et permettent de structurer un code. Un module correspond à un fichier [15] dont le nom doit être suffixé en `.py` et qui contient des définitions de fonctions et/ou de variables (et/ou de classes). Nous allons nous initier aux modules et fonctions à l'aide d'un exemple où nous définissons deux fonctions :

- l'une appelée `quad` qui, étant donné les 4 nombres $a$, $b$, $c$ et $x$ calcule $ax^2 + bx + c$ ;
- et l'autre, `resoud_quad` qui, étant donné $a$, $b$, et $c$ calcule les deux racines de $ax^2+bx+c = 0$ (si le discriminant est positif ou nul).

puis nous utilisons ces fonctions sur deux exemples.

15. Il n'y a donc pas de mot clé spécial à écrire, le nom du module correspond au nom du fichier sans le suffixe `.py`.

### Edition du fichier avec Spyder

Si vous cliquez sur l'icône au haut à gauche pour créer un nouveau fichier vous allez voir que Spyder remplit le début du fichier avec une entête qui ressemble à :

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jun 21 14:54:01 2014

@author: bruno
"""
```

La première ligne est importante car elle permet à l'interprète python de savoir quel est l'encodage des caractères utilisé. En effet dès qu'on utilise une lettre accentuée [16] même dans un commentaire, l'interprète python refusera d'exécuter votre fichier si vous ne lui indiquez pas précisément le jeu de caractères utilisé par l'éditeur (utf-8, isolatin1, etc...) sous la forme de ce commentaire spécial mis en première ligne du fichier. Le problème est qu'il n'est pas forcément simple de savoir quel est le jeu de caractères utilisé mais l'éditeur de Spyder le sait et nous offre cette première ligne spéciale.

La deuxième partie de l'entête est une chaîne de caractères entourée de 3 guillemets appelée docstring permettant de documenter le fichier module. Une telle chaîne peut en fait tenir sur autant de lignes que vous voulez. Ces chaînes "docstring" peuvent être aussi utilisées pour chaque définition de fonction.

16. En fait tout caractère qui ne serait pas de l'ascii pur.

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jun 21 14:54:01 2014

@author: bruno

"""

# REMARQUE : il y a quelques commentaires explicatifs ne les rentrez pas !

from math import *   # pour utiliser sqrt (sans préfixe)

# définition de la fonction quad (pas de blancs avant le def)
# n'oubliez pas les : après le def
def quad(x,a,b,c):
    """retourne a x^2 + b x + c"""  # docstring (permet de documenter la fct)
    return a*x**2 + b*x + c

# définition de la fonction resoud_quad (qui retourne 2 arguments)
def resoud_quad(a,b,c):
    """résolution de a x^2 + b x + c = 0"""
    assert a != 0, "a = 0 : ce n'est pas une équation du second degré !"
    delta = b**2 - 4*a*c
    assert delta >= 0, "discriminant strictement négatif"
    x1 = (-b - sqrt(delta))/(2*a)
    x2 = (-b + sqrt(delta))/(2*a)
    return x1, x2  # les arguments retournés sont séparés par des virgules

# on utilise les fonctions
x1, x2 = resoud_quad(1.,-3.,1.)
print(x1)
print(x2)
print(quad(x1,1.,-3.,1.))
print(quad(x2,1.,-3.,1.))

x1, x2 = resoud_quad(1., 2.,4.)
```

**Quelques commentaires**

1. **Sur l'importation d'un module :** dans `resoud_quad` on a besoin de la fonction `sqrt` qui n'est pas directement disponible dans python. La solution est de l'"importer" du module `math` et cela peut se faire de multiples façons :
   - Ici nous avons choisi `from math import *` qui importe toutes les fonctionnalités (cad les fonctions, les variables, etc) définies par ce module sans leur donner de suffixe. Cela est pratique mais peu être problématique lorsqu'on importe un module qui définit beaucoup de choses et/ou si on importe plusieurs modules : on risque alors de rencontrer le problème qu'un même nom (identificateur) soit donné à différentes fonctions et/ou variables ! Par exemple si on définit notre propre fonction `sqrt` après la commande d'importation c'est cette dernière qui sera utilisée par `resoud_quad`.
   - On aurait pu utiliser `import math` qui rend disponible (cad "importe") toutes les fonctionnalités du module `math` avec le préfixe `math.` ce qui est lourd mais sûr !
   - On peut aussi vouloir changer de préfixe car il est peut être un peu long. Avec `import math as m` toutes les fonctionnalités de `math` sont accessibles avec le préfixe `m.` .
   - Il est possible de sélectionner les fonctions que l'on veut importer en utilisant `from math`

`import sqrt, sin, cos`. Dans ce cas seules ces 3 fonctions de `math` seraient utilisables dans notre module sans préfixe.

— Encore plus fort, il est possible de renommer les fonctions importées : avec `from math import sqrt as racine_carree, sin as sinus, cos as cosinus` ces trois fonctions sont importées et disponibles sous les nouveaux noms donnés.

2. **Sur les fonctions :** une définition de fonction s'introduit avec :

   `def` *nom_fonction*`(`*arg1, arg2, ...*`):`

   L'indentation est importante car elle définit la portée de la fonction [17]. L'indentation permet de ne pas utiliser de caractères ou mots clés délimitant les constructions algorithmiques (fonctions, tests, boucles, ...). Chaque instruction s'écrit en général sur une ligne [18] mais une instruction peut en utiliser plusieurs :

   — Par exemple toute construction utilisant des paires de `()`, `[]` ou {} peut tenir sur plusieurs lignes tant qu'on ne rencontre pas la parenthèse, le crochet ou l'accolade de fermeture. Ainsi vous pouvez définir une liste sur plusieurs lignes :

   ```
   L = [ 'toto', 1,
         'a',
         'titi']
   ```

   — en utilisant le caractère \ à la fin d'une ligne à continuer.

   Une fonction peut retourner 0 ou un ou plusieurs arguments en utilisant le mot clé `return` et il peut y avoir plusieurs `return` dans le corps de la fonction. Normalement quand une fonction `func` retourne 2 arguments ou plus, supposons par exemple que `func` retourne 3 arguments et demande 2 arguments d'entrée, alors un appel à `func` sera généralement de la forme :

   ```
   r1, r2, r3 = func(a1,a2)
   ```

   mais il est possible de l'appeler de cette façon :

   ```
   r = func(a1,a2)
   ```

   et dans ce cas `r` est un tuple de taille 3 contenant les arguments retournés par `func`.

   **Remarque importante :** tout argument d'entrée qui sera une référence sur un objet modifiable permet de modifier l'objet en question dans la fonction : certains arguments d'entrée comme les listes peuvent donc être modifiés par une fonction.

3. **L'instruction assert :**

   `assert` *expr_bool* `,` *expr_str*

   permet d'arrêter le déroulement d'un programme en affichant le message d'erreur *expr_str* si l'expression booléenne *expr_bool* s'évalue à faux. Noter que malgré nos deux `assert` la fonction n'est pas totalement "blindée" car on ne vérifie pas si les arguments $a$, $b$ et $c$ sont des entiers ou des flottants.

### Exécution du fichier

L'exécution d'un fichier (cad d'un module) python n'est pas si triviale... En gros pour exécuter un module il faut l'importer ! Cependant si on ne quitte pas l'interprète et que l'on modifie notre fichier, un deuxième `import` ne fonctionnera pas et il faut utiliser la fonction `reload`. Ces détails

17. L'éditeur de Spyder fait cela tout naturellement, c'est à dire que vous n'avez pas à entrer de tabulations ou blancs : il le fait à votre place.

18. Il est possible d'écrire plusieurs instructions sur une même ligne en les séparant par des points-virgules.

sont remis à plus tard car Spyder nous permet d'exécuter et ré-excuter un fichier sans se poser de questions. Pour cela il suffit de cliquer sur l'icône représentant un triangle vert ou encore d'entrer le raccourci clavier F5. Voici ce que vous devez obtenir dans la fenêtre de commande de l'interprète :

```
0.38196601125
2.61803398875
1.11022302463e-16
0.0

Traceback (most recent call last):
  File "/home/bruno/Enseignement/Python/titi.py", line 29, in <module>
    x1, x2 = resoud_quad(1., 2.,4.)
  File "/home/bruno/Enseignement/Python/titi.py", line 17, in resoud_quad
    assert delta >= 0, "discriminant strictement négatif"
AssertionError: discriminant strictement négatif
```

Commentaires :

1. on voit ici que les calculs avec les flottants ne sont pas exacts ;
2. conversion des entiers en float : on peut utiliser `resoud_quad(1,2,1)` et ça marche en partie parce que la racine carrée appliquée sur un entier donnera un float (il pourrait y avoir des problèmes en python 2.7 avec la division qui serait comprise comme entière...).
3. le deuxième exemple échoue (discriminant négatif) : on remarque que le message d'erreur "AssertionError : discriminant strictement négatif" est précédé de la trace des appels qui ont conduit à cette erreur.

**A vous de jouer**

Maintenant que l'on a écrit ce module, il est possible de changer les exemples à la fin du module, de sauvegarder le fichier puis le ré-exécuter avec F5. Mais on peut aussi (approche interactive) tester directement les fonctions dans l'interprète. Noter que de manière transparente F5 à permis d'importer le module sans préfixe, on a ainsi accès aux fonctions `quad` et `resoud_quad` directement avec leur nom sans préfixe :

```
>>> x1, x2 = resoud_quad(1,2,1)
>>> x1; x2                      # affichage des racines
-1.0
-1.0
>>> rac = resoud_quad(1,2,1)    # les 2 arguments de resoud_quad sont affectés à une seule variable
>>> rac                         # => rac est alors un t-uple
(-1.0, -1.0)
>>> help(quad)                  # aide sur quad (affiche la docstring)
Help on function quad in module __main__:

quad(x, a, b, c)
    retourne a x^2 + b x + c
```

### 2.3.2 tests if

Cette construction prend la forme suivante :

`if` *test*:

*instructions exécutées si test est vrai*

`else:`

*instructions exécutées si test est faux*

où l'indentation détermine la portée des instructions et *test* est une expression booléenne (pas besoin d'entourer l'expression booléenne avec des parenthèses). S'il n'y a rien à faire dans le cas où *test* s'évalue à faux, on n'écrit juste :

`if` *test*:

*instructions exécutées si test est vrai*

**les expressions booléennes, les opérateurs de comparaison**

On a pas encore parlé des expressions booléennes mais elles sont assez simple à former, les 3 opérateurs logiques usuels `et`, `ou` et `non` s'écrivent respectivement `and`, `or` et `not`. Noter que `not` est prioritaire sur le `and` qui lui-même est prioritaire sur le `or`. Cela veut dire que l'expression :

`not` *expr_bool1* `and` *expr_bool2* `or` *expr_bool3*

sera interprétée comme :

((`non` *expr_bool1*) `et` *expr_bool2*) `ou` *expr_bool3*

Exemple : supposons que $x$ soit un nombre (entier ou flottant) et que le test porte sur la non appartenance de $x$ à l'ensemble $[\![2, 6]\!] \cup [\![12, 17]\!]$, on peut écrire :

```
not ( (2 <= x and x <= 6) or (12 <= x and x <= 17) )
```

Comme le `and` est prioritaire sur le `or` on peut se passer des parenthèses et écrire :

```
not ( 2 <= x and x <= 6  or  12 <= x and x <= 17 )
```

mais on peut vouloir garder les parenthèses si on juge l'expression plus lisible. En fait on peut se passer des `and` et écrire :

```
not ( (2 <= x <= 6) or (12 <= x <= 17) )  # ou encore  not ( 2 <= x <= 6  or  12 <= x <= 17 )
```

car si `op1`, `op2` sont des opérateurs de comparaison :

| | |
|---|---|
| `==` | égalité |
| `!=` | non égalité |
| `<=` | inférieur ou égal |
| `<` | strictement inférieur |
| `>=` | supérieur ou égal |
| `>` | strictement supérieur |

et *expr1*, *expr2* et *expr3* des expressions (donnant des entiers, des flottants des chaînes de caractères ou tout objet pour lesquels les opérateurs précédents sont définis) alors une expression de la forme :

*expr1* `op1` *expr2* `op2` *expr3*

est interprété comme :

*expr1* `op1` *expr2* `and` *expr2* `op2` *expr3*

Noter qu'il faut bien les parenthèses entourant `2 <= x <= 6  or  12 <= x <= 17` car le `not` s'appliquerait uniquement sur `2 <= x <= 6` étant donné sa priorité sur le `or`.

Pour terminer sur les priorités, les opérateurs arithmétiques sont prioritaires sur les opérateurs de compaisons qui ont une priorité supérieure aux opérateurs logiques.

**Autres formes de if**

Parfois lorsque le test est faux la partie `else:` débouche sur un autre `if` et il est alors possible d'écrire le code différemment en utilisant le mot clé `elif` :

`if` *test1*:
    *instructions exécutées si test1 est vrai*
`elif` *test2*:
    *instructions exécutées si test1 est faux et test2 est vrai*
`else:`
    *instructions si exécutées si test1 et test2 sont faux*

Bien sûr vous pouvez utiliser autant de `elif` que nécessaires.

Exemple : parfois on a besoin d'une fonction signe qui renvoie trois valeurs $-1$ si la valeur testée est strictement négative, 0 si la valeur est 0 et sinon 1. Une telle fonction peut s'écrire :

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

Pour terminer sur les tests, noter qu'il peut être pratique d'utiliser l'opérateur `in`. Par exemple si on veut tester l'appartenance à un ensemble de valeurs entières non contiguës comme $\{-6, 1, 8, 9, 123\}$ on peut utiliser l'expression booléenne `x in [-6, 1, 8, 9, 123]`

### 2.3.3 boucles "tant que"

La forme la plus simple des boucles "tant que" est la suivante :

`while` *test*:
    *instructions*

où l'indentation détermine le corps de la boucle et *test* est une expression booléenne. Voici un exemple concret : pour rechercher si un élément `elem` est dans une liste `L` on pourrait utiliser :

```
trouve = False; i = 0
while not trouve and i < len(L):
    trouve = elem == L[i]
    i = i+1
```

mais c'est inutile puisqu'on dispose de l'opérateur `in`.

Il est possible de sortir autrement de la boucle en utilisant un `if` et un `break`. Exemple [19] :

`while` *test1*:
    *instructions*
    `if` *test2*:
        `break`
    *instructions2*

Donc lorsque *test2* est vrai on sort directement de ce `while` sans exécuter la deuxième partie des instructions (*instructions2*). En général, sauf cas particuliers, on essaie d'éviter ce style de programmation qui peut être difficile à lire.

19. Noter qu'on peut mettre une instruction juste après les `:` des `if`, donc ici on aurait pu écrire `if` *test2*`: break`

### 2.3.4 boucles “pour”

Les boucles `for` permettent d'itérer sur toutes séquences :

`for` *k* `in` *seq* :
    *instructions*

où *seq* est donc un objet qui a la propriété d'être une séquence comme les listes, les chaînes, les tuples, les tableaux numpy ou d'autres. Le nombre d'itérations de la boucle est égal au nombre d'éléments de la séquence [20] et à la $k$-ième itération, la variable de boucle `k` est égale au $k$-ième élément de *seq*. Voici un exemple où l'on affiche les éléments d'une séquence :

```
for elem in L:
    print(elem)
```

Pour avoir des boucles qui ressemblent à celles d'autres langages on peut utiliser les listes obtenues avec la construction `range`. Exemples :

```
for i in range(n):           # i sera égal successivement à 0, 1, ..., n-1

for i in range(n-1,-1,-1):   # i sera égal successivement à n-1, n-2, ..., 0

for i in range(0,n,2):       # i sera égal successivement à 0, 2, 4, ...(on s'arrête avant n)

for i in range(1,n,2):       # i sera égal successivement à 1, 3, 5, ...(on s'arrête avant n)
```

Ainsi si on veut collecter (dans une liste `ind`) tous les indices d'une liste `L` pour lesquels on a égalité avec un objet `elem` on peut écrire :

```
ind = []  # liste vide
for i in range(len(L)):
    if elem == L[i]:
       ind.append(i)
```

Comme pour les boucles `while` il est possible de sortir prématurément avec un `break`.

### 2.3.5 print

Pour le moment nous avons utilisé la forme fonction de print (`print(obj)`) qui affiche l'objet `obj` sans trop de contrôle. Il existe une forme “instruction” de print qui permet facilement d'ajouter des directives de formatage mais elle n'est plus supportée avec les versions 3.x. Pour se débrouiller uniquement avec la fonction `print` (de sorte à écrire un code portable entre version 2.7 et 3.3), on peut convertir toutes les variables numériques en chaînes de caractères avec la fonction `str` et former une grande chaîne de caractères par concaténation (avec l'opérateur `+`) et finalement afficher cette grande chaîne avec la fonction `print`. Voici un exemple :

```
x = 1.72    # flottant
k = 981     # entier
s = "toto"  # chaîne
print( "Notre ami " + s + " a " + str(k) + " ans et mesure " + str(x) +" m" )
```

Cependant avec `str` on ne contrôle pas le format précis de la conversion. Par contre, la fonction `format(obj,fmt)` peut faire ce travail ; voici un exemple déjà utilisé dans la partie sur les flottants :

20. Au nombre de caractères dans le cas d'une chaîne, au nombre de lignes dans le cas d'un tableau 2d.

```
>>> format(0.2,"24.17e")
' 2.00000000000000011e-01'
```

Le deuxième argument de `format` est une chaîne de caractère précisant comment la conversion en chaîne de caractères du premier argument doit s'effectuer. Ici on veut une sortie sous forme scientifique (le `e` comme exposant) tenant sur 24 caractères avec une mantisse comportant 17 chiffres après le point décimal. En dehors de `e` il y a les possibilités suivantes :

- `f` pour afficher un flottant sous forme décimale ;
- `g` pour afficher un flottant sous forme décimale s'il n'est pas de magnitude trop grande ou trop petite et sinon sous forme scientifique.

On utilisera `format` à la place de `str` lorsque le formattage de cette dernière ne convient pas.

Enfin si on veut une petite interactivité (avec la fenêtre de commande) pour demander de rentrer une valeur, on pourra utiliser la fonction `raw_input(`*message*`)`. Elle affiche son argument dans la fenêtre de commande (*message* doit être une chaîne de caractères) puis attend que vous entriez des caractères au clavier. Lorsque vous avez appuyé sur la touche Entrée, elle retourne alors la chaîne de caractères brute entrée au clavier. Si la réponse attendue est un nombre, il faut transformer la chaîne en utilisant l'une des fonctions de conversions `int`, `float` ou `complex`. Ecrivez l'exemple ci-dessous dans un fichier, tester le et modifier le.

```
str_x = raw_input(" x = ")  # str_x est une chaîne de caractères
x = float(str_x)
print " on a entré x = %g" % x
# entrée d'un entier :
k = int(raw_input(" k = "))
print(k)
c = complex(raw_input(" entrer le nb complexe c = "))
print(c)
```

**Attention :** avec python 3.3 il faut utiliser `input(`*message*`)` qui a le comportement de la fonction `raw_input` de python 2.7.

Cette fonction `input` existe déjà en python 2.7 mais fait toute seule le travail de conversion dans le cas où la chaîne entrée en un nombre.

## 2.4 Exercices

Tous les exercices consistent à écrire des fonctions. Celles correspondant aux exercices 1 à 8 peuvent être codées dans le même fichier module. Dès que vous avez écrit ou corrigé une nouvelle fonction vous pourrez re-exécuter le fichier avec F5 et ainsi tester rapidement celle-ci directement dans la fenêtre de commande.

Les fonctions correspondant aux exercices 9 et 10 (méthode de Newton et introduction aux complexes) peuvent être écrite dans un autre fichier module ainsi que celles du dernier exercice (Variation sur Fibonacci).

**Exercice 1** *calcul de $n$!*

Ecrire une fonction d'entête `def fact(n):` qui calcule $n!$. Comme il est possible d'utiliser la récursivité en python, écrire une seconde version `def factbis(n):` récursive. Tester vos fonctions et apprécier les entiers longs de python !

Remarque : une fonction python est un objet au même titre qu'un entier, un flottant, une liste, etc... Vous pouvez ainsi mettre vos 2 fonctions dans une liste et les utiliser via cette liste :

```
>>> type(fact)
<type 'function'>
>>> L = [fact, factbis]
>>> L[0](6)   # appel à fact
720
>>> L[1](6)   # appel à factbis
720
```

et une fonction peut être donc passée comme argument d'une autre fonction.

**Exercice 2** *suite de Fibonacci*

On rappelle la suite de Fibonacci $F_0 = 0, F_1 = 1$ et pour $n \geq 2$, $F_n = F_{n-2} + F_{n-1}$. Ecrire une fonction d'entête `def fib(n):` qui calcule $F_n$ sans utiliser la récursivité et sans mémoriser tous les termes de la suite dans une liste. Ecrire ensuite une version récursive simple `def fibbis(n):` qui s'avèrera peu efficace.

**Exercice 3** *former une liste au hasard*

Ecrire une fonction d'entête `def randlist(n,a,b):` qui retourne une liste de longueur $n$ dont les éléments sont des entiers "aléatoires" compris entre $a$ (inclus) et $b$ (exclus). Pour obtenir un tel entier aléatoire, on importera le sous-module `random` du module `numpy` en utilisant au début du fichier module :

```
import numpy.random as alea
```

et les fonctions de ce sous-module seront accessibles avec le préfixe `alea`. Celle qui nous intéresse est `randint` et `alea.randint(a,b)` retournera un tel entier aléatoire.

Aide : utiliser `L = []` pour créer une liste vide et la méthode `append` pour rajouter successivement les $n$ entiers aléatoires.

**Exercice 4** *calcul du minimum ou du maximum ou des deux, d'une liste d'entiers*

Il existe déjà les fonctions `min` et `max` qui font ce travail mais écrire une telle fonction reste un exercice simple et la dernière question est un peu plus intéressante.

1. Ecrire une fonction d'entête `def mini(L):` permettant de calculer le minimum d'une liste d'entiers. On peut remarquer que sur une liste de taille $n$ l'algorithme utilise $n-1$ comparaisons. Tester votre fonction avec des listes obtenues avec la fonction précédente. Lorsque la liste est vide, on retournera l'objet `None`.
2. De même, on pourrait écrire une fonction qui calcule le maximum et appeler successivement les 2 fonctions pour avoir le minimum et le maximum d'une liste d'entiers ce qui nous demanderait $2(n-1)$ comparaisons. Or il est possible de faire mieux avec l'idée suivante : appelons $u_0, u_1, \ldots, u_{n-1}$ nos entiers et posons :

$$m_k := \min_{j\in[\![0,k]\!]} u_j, \quad M_k := \min_{j\in[\![0,k]\!]} u_j$$

les min et max courants. Pour économiser des tests, on compare d'abord $u_{k+1}$ et $u_{k+2}$ entre eux, puis on compare le plus grand de ces deux nombres à $M_k$ (ce qui permet de déterminer $M_{k+2}$) et le plus petit à $m_k$ (ce qui permet de déterminer $m_{k+2}$). Pour avoir toujours un nombre pair de nombres à tester, il suffit de jouer sur l'initialisation (min et max initiaux obtenus avec $u_0$ ou bien avec $u_0$ et $u_1$). Ecrire une fonction d'entête `def minmax(L):` qui retourne le minimum et le maximum d'une liste d'entiers à l'aide de cet algorithme.

**Exercice 5** *Recherche d'un motif dans une chaîne de caractères*

Ecrire une fonction d'entête `def recherche(pattern, str):` qui donne les indices de toutes les occurrences de la chaîne `pattern` dans la chaîne `str` (par exemple si la chaîne `pattern` (de taille `np = len(pattern)`) est égale à la sous-chaîne `str[i:i+np]` alors `i` est un tel indice).

La fonction retournera une liste (vide si il y a aucune occurence de `pattern` dans `str`). Exemples :

— avec la chaîne `"tttt"` et le pattern `"tt"` la fonction doit retourner les indices 0, 1 et 2 ;
— avec la chaîne `"totititi toto tititi"` et le pattern `"titi"` la fonction doit retourner les indices 4, 14 et 16.

**Exercice 6** *Décomposition d'une chaîne de caractères en mots*

Ecrire une fonction d'entête `def decompose(phrase,sep):` permettant de décomposer la chaîne de caractères `phrase` en mots que l'on mettra dans une liste retournée par la fonction. La liste (ou la chaîne) `sep` contient les caractères considérés comme des "séparateurs" (en général le point, la virgule, le point-virgule, le caractère blanc,...), un "mot" étant alors défini comme une sous-chaîne non vide contenant n'importe quels caractères autres que ceux contenus dans `sep`. Exemple :

```
>>> phr = "Les sanglots longs des violons de l'automne."
>>> decompose(phr,[" ",".",",",";"])
['Les', 'sanglots', 'longs', 'des', 'violons', 'de', "l'automne"]
```

Si on ajoute l'apostrophe dans les caractères séparateurs, on obtiendra le mot supplémentaire `l`.

**Exercice 7** *calcul approché de $e^x$*

Ecrire une fonction d'entête `def expo(x):` qui approche l'exponentielle en ajoutant un à un les termes de la série :

$$e^x = \sum_{n=0}^{\infty} t_n = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \ldots$$

jusqu'à ce que l'addition d'un nouveau terme ne modifie plus la somme. Utiliser la relation $t_n = (x/n) \times t_{n-1},\ n \geq 1$ pour calculer les termes de la somme au fur et à mesure.

Tester votre fonction sur plusieurs exemples en la comparant à la fonction `exp` du module `math`.

Attention au piège suivant : avec python 2.7 si votre argument est un entier python (par exemple si on veut calculer `expo(3)`) la division $x/n$ sera considérée comme entière ! Il faut donc utiliser `expo(3.0)` ou utiliser la conversion `x = float(x)` au début de votre fonction encore insérer l'instruction magique `from__future__ import division` au tout début du module (après le commentaire spécial précisant le jeu de caractère utilisé). Ceci est un argument en faveur du nouveau comportement de `/` sur les entiers avec python 3.x.

**Exercice 8** *enlever les éléments redondants d'une liste*

Ecrire une fonction d'entête `def unique(L):` qui, enlève les éléments redondants d'une liste quelconque. Cette fonction doit modifier la liste et ne retourne aucun argument. On utilisera l'algorithme évident [21] suivant : lorsque les $k$ premiers éléments de la liste sont uniques, on teste [22] le $k+1$ contre les $k$ premiers. S'il existe déjà on le détruit (fonction `del`) et sinon on le garde et on incrémente $k$, etc...

Tester votre fonction à l'aide des listes obtenues avec la fonction `randlist`.

**Exercice 9** *méthode de Newton, introduction aux arguments optionnels nommés*

Avec python on dispose d'arguments optionnels nommés dont les valeurs par défaut sont initialisées dans l'entête de la fonction. L'entête d'une telle fonction ressemblera à :

```
def func(arg_1, .., arg_n, opt_1 = val_1, ..., opt_m = val_m):
```

Lors de l'appel à une telle fonction, les $n$ arguments d'entrée `arg_1`, ..., `arg_n` sont à fournir obligatoirement et chacun des $m$ arguments optionnels peut être fourni ou pas et ceci dans n'importe quel ordre. Par exemple avec :

```
func(arg1, .., argn, opt_3 = 3, opt_1 = "toto")
```

on précise seulement la valeur des arguments optionnels 1 et 3 (les autres arguments optionnels prennant leur valeur par défaut). Ce mécanisme est très pratique pour écrire des fonctions qui prennent certains paramètres "annexes" comme des critères de convergence numérique.

Méthode de newton : pour une fonction $f$ assez régulière et un réel $x_0$ suffisamment proche d'une racine $r$ de $f$, cette méthode consiste en les itérés :

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

et converge assez rapidement vers $r$ sous certaines conditions.

Dans cet exercice, on demande de coder cette méthode avec comme test d'arrêt :

$$\frac{|x_k - x_{k-1}|}{\max\{|x_k|, 1\}} \leq \epsilon$$

(auquel cas $x_k$ est considéré comme l'approximation de $r$ cherchée). Néanmoins si on dépasse un certain nombre d'itérations *itermax* on terminera l'exécution de la fonction. Lorsqu'elle converge la méthode de Newton est (en général) très rapide, on peut donc se contenter d'une valeur par défaut de l'ordre de 10 pour *itermax*. La valeur par défaut du paramètre $\epsilon$ sera par exemple de $2\ 10^{-16}$ valeur très proche du epsilon machine.

21. On peut faire mieux en utilisant la méthode `sort`.
22. On pourra écrire une version qui utilise `in` et une autre où on écrit le code pour ce test.

L'entête de la fonction peut être `def newton(x0, f, fp, eps = 2e-16, itermax = 10):` et elle devra retourner deux arguments, le dernier élément de la suite obtenue et l'une des deux chaîne de caractère, `'succes'` ou `'echec'` permettant de préciser si le test de convergence numérique est vrai ou faux.

Attention pour avoir la valeur absolue sur les nombres flottants il faut importer le module `math` et la fonction s'appelle `fabs` (et pas `abs`). Si dans le fichier vous importez `math` sans préfixe, vous aurez accès à toutes les fonctions de ce module. Et comme avec F5 on importe le module sans préfixe vous pourrez tester votre fonction dans l'interprète avec :

```
>>> newton(6,sin,cos)
(6.283185307179586, 'succes')
```

Cette fonction pourrait être perfectionnée :

— en utilisant le test suivant :

$$\frac{|x_k - x_{k-1}|}{\max\{|x_k|, 1\}} \leq \epsilon \quad \textbf{et} \quad |f(x_k)| \leq \epsilon_f$$

puisque le but est de trouver une racine de $f$. On aurait donc un troisième argument optionnel nommé.

— il existe une façon d'augmenter le domaine de convergence de la fonction et de rendre ainsi plus robuste cette méthode de Newton.

**Exercice 10** *introduction aux nombres complexes : résolution de l'équation du second degré*

On aimerait que la fonction `resoud_quad` (cf le module `toto`) puisse traiter le cas $\Delta < 0$ et aussi qu'elle puisse admettre des arguments complexes. On voudrait aussi vérifier que les arguments d'entrée de la fonction sont bien des nombres (entiers et/ou flottants et/ou complexes).

Cela va nous permettre de travailler avec les nombres complexes et sur la vérification des types des objets en python. Voici les compléments de python nécessaires :

1. Nous avons déjà vu l'utilisation de la fonction `type` voici de nouveau quelques exemples :

```
>>> t = type(1)
>>> t
<type 'int'>
>>> type(t)
<type 'type'>
>>> type(1.)
<type 'float'>
>>> type(1.+4j)    # un complexe s'introduit avec a + bj ou a + bJ ou complex(a,b)
<type 'complex'>
>>> type("c")
<type 'str'>
```

Appliqué sur n'importe quel objet python, cette fonction retourne le type (la classe) de l'objet sous la forme d'un objet python de type `type` (et pas sous la forme d'une chaîne de caractères). Pour tester si un objet `o` est un entier on peut donc utiliser l'expression booléenne `type(o) == type(1)`. Néanmoins comme python fournit par défaut toutes les variables de type `type` dont on a besoin : `int`, `float`, `complex`, `str`, etc, il est plus lisible d'écrire `type(o) == int` pour tester si `o` est un entier. Attention ces variables ne sont pas protégées et donc le code suivant ne fonctionnera pas :

```
int = 5   # int n'est plus une ref sur l'objet "type entier" mais sur l'objet "entier 5"
type(1) == int # renvoie False
```

Pour tester si un objet `o` est un nombre entier, flottant ou complexe (flottant) on peut donc utiliser l'expression booléenne :

```
type(o) == int  or  type(o) == float  or  type(o) == complex
```

2. On peut aussi utiliser la fonction `isinstance` :

```
>>> c = 1 + 2j          # 3 façons de définir un complexe
>>> c = 1 + 2J
>>> c = complex(1,2)
>>> isinstance(c,complex)
True
```

Un avantage de `isinstance` sur `type` c'est que son deuxième argument peut être un tuple de types et la fonction renvoie vrai si le type de l'objet correspond à l'un des types du tuple. Ainsi pour tester si l'objet `o` est un nombre on peut utiliser l'expression :

```
isinstance(o, (int,float,complex) )
```

3. Les fonctions mathématiques sur les complexes sont dans le module `cmath` il faut donc importer ce module pour avoir la bonne fonction `sqrt`. C'est donc le bon moment pour re parler des problèmes de noms... Si on importe les deux modules `math` et `cmath` de cette façon :

```
from math import *
from cmath import *
```

alors les fonctions de `math` portant des noms identiques aux fonctions de `cmath` (comme `sqrt`) seront "masquées" par ces dernières. Ceci n'est pas forcément dommageable puisque les fonctions qui attendent des arguments complexes fonctionnent aussi pour les arguments réels (flottants). Cependant les arguments retournés seront considérés comme complexes. Par exemple la fonction complexe `sqrt` renvoie $2 + 0j$ si on lui donne l'entier 4 à manger :

```
>>> import cmath
>>> cmath.sqrt(4)
(2+0j)
>>> import math
>>> math.sqrt(4)
2.0
```

Il faudrait donc appliquer la fonction `float` si on veut un résultat réel.

Bref il est sans doute plus approprié d'importer `math` et `cmath` de cette façon :

```
from math import *
import cmath         # ou encore from cmath import sqrt as csqrt
```

Maintenant vous disposez des éléments pour écrire une fonction `resoud_quad(a,b,c)` plus sûre fonctionnant sur des nombres entiers et/ou flottants et/ou complexes (flottants) (générer une exception avec `assert` si l'un des 3 arguments n'est pas un nombre) et capable de traiter le cas réel lorsque $\Delta < 0$ et le cas complexe. Pour cela vous pouvez écrire au préalable une fonction `is_number` qui renvoie vrai si son argument est `int`, `float` ou `complex` et une fonction `is_real` qui renvoie vrai si son argument est `int` ou `float`.

**Exercice 11** *Variations sur Fibonacci, introduction à l'instruction global*

Un exercice qui peut être intéressant pour les élèves est de montrer pourquoi la fonction `fibbis` est peu efficace. Pour cela on peut tracer l'arbre des appels récursifs et se rendre compte que l'on calcule souvent la même chose. Plus précisément on peut déterminer le nombre d'appels $A(n)$ à la fonction `fibbis` pour calculer $F_n$ :

— On a $A(0) = A(1) = 1$ : en effet un seul appel à `fibbis` dans ce cas puisqu'on tombe sur la récursivité terminale donnant immédiatement le résultat.
— Pour $n > 1$ on a $A(n) = 1 + A(n-1) + A(n-2)$ (1 pour l'appel `fibbis(n)` mais ensuite le code demande de calcul de `fibbis(n-1)` et de `fibbis(n-2)`).

La solution de cette récurrence (algèbre linéaire de sup ou de spé) nous donne :

$$A(n) = \left(1 + \frac{1}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(1 - \frac{1}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n - 1$$

et l'on voit donc que le nombre d'appels, asymptotiquement donné par le premier terme (l'autre tend vers 0 avec $n$) sera proche de $1.44...(1.61...)^n$. Pour $n = 30$ on aura plus de 2 millions d'appels à `fibbis` !

Ainsi on peut vouloir expérimenter tout cela de la façon suivante. Dans un nouveau fichier module :

— on définit une variable globale nommée `nb_appels` qu'il faudra réinitialiser à 0 avant un appel et incrémenter à chaque appel de `fibbisc` ;
— on réécrit une nouvelle version de la fonction `fibbis` appelée par exemple `fibbisc` avec en plus l'incrémentation de la variable globale `nb_appels` : attention si les variables définies dans un module à l'extérieur de toutes les fonctions, sont utilisable en lecture dans les fonctions, il faut impérativement les déclarer comme globale si on veut pouvoir les modifier dans une fonction. Ainsi `fibbisc` s'écrit :

```
def fibbisc(n):
    """n ème terme de la suite de Fibonacci version récursive naïve
       cette version incrémente le compteur nb_appel pour compter
       le nb d'appels"""
    global nb_appels
    nb_appels = nb_appels + 1
    if n <= 1:
        return n
    else:
        return fibbisc(n-2)+fibbisc(n-1)
```

— On écrit une fonction `nb_appels_fibbisc` permettant de calculer le nombre d'appels $A(n)$ à l'aide de la formule précédente. Elle donne théoriquement un entier mais comme les calculs se font en flottant, il peut y avoir une légère erreur. Pour obtenir un entier (au sens informatique du terme) vous pouvez utiliser la fonction `int` sur le résultat de la formule.
— On peut aussi écrire une fonction permettant de réinitialiser le compteur (ou bien il faudra le faire plus manuellement).

Ensuite il faut comparer la théorie à la pratique ; après avoir chargé votre fichier dans l'interprète avec F5 :

```
>>> nb_appels = 0    # mise à 0 du compteur
>>> fibbisc(18)
2584
>>> nb_appels        # on obtient en pratique 8361 appels
8361
>>> nb_appels_fibbisc(18)  # et 8361 appels en théorie !
8361
```

On peut aussi réfléchir à l'écriture d'une version récursive `fibter` efficace. L'idée est que votre fonction (pour $n \geq 1$) retourne les deux derniers termes de la suite sous la forme d'une liste ou d'un t-uple : si `fibter(n)` renvoie $(F_{n-1}, F_n)$ alors un seul appel récursif (à `fibter(n-1)`) lui est nécessaire au lieu de 2.

## 2.5 Correction des exercices

Premier fichier module :

```
# *-* coding: utf-8 *-*

# pour obtenir le comportement de la division de la version 3.x :
from __future__ import division

import numpy.random as alea

def fact(n):
    """fonction factorielle version itérative"""
    fact_n = 1
    for k in range(1,n+1):
        fact_n = fact_n * k
    return fact_n

def factbis(n):
    """fonction factorielle version récursive"""
    if n <= 1:
        return 1
    else:
        return n*factbis(n-1)

def fib(n):
    """n ième terme de la suite de Fibonacci version itérative"""
    if n <= 1:
        return 1
    else:
        fp = 0
        f = 1
        for i in range(2,n+1):
            temp = f+fp
            fp = f
            f = temp
        return f

def fibbis(n):
    """n ème terme de la suite de Fibonacci version récursive naïve"""
    if n <= 1:
        return n
    else:
        return fibbis(n-2)+fibbis(n-1)

def randlist(n,a,b):
    """retourne une liste de n entiers tirés uniformément dans [a,b-1]"""
    L = []
    for i in range(n):
        L.append(alea.randint(a,b))
    return L
```

```
def mini(L):
    """retourne le minimum d'une liste de nombres"""
    n = len(L)
    if n == 0:
        return None
    m = L[0]
    for k in range(1,n):
        if L[k] < m:
            m = L[k]
    return m

def minmax(L):
    """retourne à la fois le min et le max d'une liste de nombres"""
    n = len(L)
    if n == 0:
        return None,None
    # initialisation selon la parité de n
    if n % 2 == 0:  # n pair
        if L[0] < L[1]:
            m = L[0]; M = L[1]
        else:
            m = L[1]; M = L[0]
        start = 2
    else:            # n impair
        m = L[0]; M = L[0]
        start = 1
    # boucle principale
    for k in range(start,n,2):
        if L[k] < L[k+1]:  # L[k] est comparé à m et L[k+1] à M
            if L[k] < m:
                m = L[k]
            if L[k+1] > M:
                M = L[k+1]
        else:              # L[k] est comparé à M et L[k+1] à m
            if L[k] > M:
                M = L[k]
            if L[k+1] < m:
                m = L[k+1]
    return m,M


def recherche(pattern, str):
    """retourne les occurences de pattern dans la chaîne str"""
    np = len(pattern)
    ns = len(str)
    L = []
    for i in range(ns-np+1):
        if str[i:i+np] == pattern:
            L.append(i)
    return L
```

```
def decompose(str,sep):
    """décompose la phrase str en une liste de mots"""
    mots = []
    mot = ""
    for k in range(len(str)):
        if str[k] in sep:
            if mot != "" :
                mots.append(mot)
                mot = ""
        else:
            mot = mot+str[k]
    if mot != "":
        mots.append(mot)
    return mots

def expo(x):
    """calcul de la fct exponentielle en sommant la série exponentielle"""
    S = 1
    n = 0
    term_n = 1
    while True:
        n = n + 1
        term_n = term_n*(x/n)
        Snew = S + term_n
        if Snew == S:
            break
        S = Snew
    return S

def unique(L):
    """enlève les doublons de la liste L"""
    k = 1
    while k < len(L):
        trouve = False
        j = 0
        while not trouve and j < k:
            trouve = L[k] == L[j]
            j = j+1
        if trouve:
            del L[k]
        else:
            k = k+1

def unique_bis(L):
    """enlève les doublons de la liste L
       (cette version utilise in pour le test d'appartenance)"""
    k = 1
    while k < len(L):
        if L[k] in L[:k]:
            del L[k]
        else:
            k = k+1
```

Deuxième fichier module :

```
# *-* coding: utf-8 *-*

# pour obtenir le comportement de la division de la version 3.x :
from __future__ import division

from math import *
import cmath

def newton(x0, f, fp, eps=2e-16, itermax=10):
    iter = 0
    x = x0
    while True:
        iter = iter + 1
        xn = x - f(x)/fp(x)
        if fabs(xn-x)/max(1,fabs(xn)) <= eps:
            return xn, "succes"
        elif iter >= itermax:
            return xn, "echec"
        x = xn

def is_number(numb):
    return isinstance(numb,(int,float,complex))

def is_real(numb):
    return isinstance(numb,(int, float))

def resoud_quad(a,b,c):
    """résolution de a x^2 + b x + c = 0"""
    assert is_number(a) and is_number(b) and is_number(c), "pas des nombres"
    assert a != 0, "a = 0 : pas une équation du second degré !"

    delta = b**2 - 4*a*c
    if is_real(delta):
        # cas réel
        if delta >= 0:
            x1 = (-b - sqrt(delta))/(2*a)
            x2 = (-b + sqrt(delta))/(2*a)
        else:
            x1 = complex(-b,-sqrt(-delta))/(2*a)
            x2 = complex(-b, sqrt(-delta))/(2*a)
    else:
        x1 = (-b - cmath.sqrt(delta))/(2*a)
        x2 = (-b + cmath.sqrt(delta))/(2*a)

    return x1, x2
```

Variations sur Fibonacci :

```
# -*- coding: utf-8 -*-
# pour obtenir le comportement de la division de la version 3.x :
from __future__ import division

from math import sqrt

"""
Created on Sun Jun 22 12:57:07 2014

Variations sur Fibonacci

@author: bruno
"""

nb_appels = 0

def init_nb_appels():
    """permet de remettre nb_appels à 0"""
    global nb_appels
    nb_appels = 0


def fibbisc(n):
    """n ème terme de la suite de Fibonacci version récursive naïve
       cette version incrémente le compteur nb_appel pour compter
       le nb d'appels"""
    global nb_appels
    nb_appels = nb_appels + 1
    if n <= 1:
        return n
    else:
        return fibbisc(n-2)+fibbisc(n-1)

def fibter(n):
    """pour n > 0, retourne le n-1 ème et le n ème terme
    de la suite de Fibonacci (version récursive efficace)"""
    if n == 0:
        return 0
    elif n == 1:
        return (0,1)
    else:
        L = fibter(n-1)
        return (L[1],L[0]+L[1])

def nb_appels_fibbisc(n):
   r5 = sqrt(5)
   return int(((1+1/r5)*((1+r5)/2)**n + (1-1/r5)*((1-r5)/2)**n) - 1)
```

# Chapitre 3

# Les tableaux et les graphiques

## 3.1 Les tableaux numpy

Les tableaux (arrays) sont des objets fournis par le module numpy. Contrairement aux listes ils contiennent des objets de types identiques, comme des flottants, des complexes flottants ou des entiers. **Attention** pour utiliser le module `numpy` il faut a priori l'importer, de même si on veut seulement tester des exemples simples dans la console python (qui est elle-même un "fichier" module particulier). Cependant Spyder fait ce travail à votre place. Si je lis le début de ma console python de mon environnement Spyder je vois les choses suivantes :

```
Python 2.7.6 (default, Mar 18 2014, 22:18:46)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.

Imported NumPy 1.8.0, SciPy 0.13.0, Matplotlib 1.3.0
Type "scientific" for more details.
```

Les modules numpy, scipy et matplotlib sont déjà importés dans la console et les noms sont utilisables sans préfixe.

### 3.1.1 Construire des tableaux

Il est possible de construire des tableaux à partir de listes avec la fonction `array`. Pour une liste simple (dont chaque élément est un entier ou un flottant) on obtiendra un tableau à une dimension, pour une liste dont les $m$ éléments sont eux même des listes "simples" à $n$ éléments on obtiendra un tableau bidimensionnel $m \times n$. Exemples :

```
>>> # création d'un tableau 1d à l'aide d'une liste
... T = array([0.1, 3, -1, 4, 6.7])  # on mélange flottants et entiers => tableau de flottants
>>> print(T)
[ 0.1  3.  -1.   4.   6.7]
>>> # création d'un tableau 2d à l'aide d'une liste (de listes)
... U = array([[0.4, 0.9, 4],[-1.5, 2, 1./3]])
>>> print(U)
[[ 0.4         0.9         4.        ]
 [-1.5         2.          0.33333333]]
>>> type(T)
<type 'numpy.ndarray'>
>>> type(U)
<type 'numpy.ndarray'>
```

Il y a plusieurs fonctions[1] qui permettent de construire des tableaux types :

1. `linspace(a,b,n)` permet de créer un tableau 1d dont les $n$ composantes sont uniformément réparties entre $a$ et $b$ :

```
>>> linspace(0,1,5)  # vecteur à 5 composantes équiréparties entre 0 et 1
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

2. `arange(a,b,inc)` permet de créer un tableau 1d et fonctionne comme la fonction `range(a,b,inc)` sauf que les arguments peuvent être des flottants :

```
>>> arange(0,1,0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

3. les fonctions `zeros` et `ones` permettent de créer des tableaux contenants des 0 ou des 1. Attention pour créer des tableaux 2d et plus, les dimensions doivent être données sous la forme d'un seul argument tuple (contenant ces dimensions). Quelques exemples :

```
>>> zeros(5)  # tableau 1d de 0
array([ 0.,  0.,  0.,  0.,  0.])
>>> ones((2,4))  # tableau 2d de taille 2 x 4 contenant des 1
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> zeros((2,2,2)) # tableau 3d de taille 2 x 2 x 2 contenant des 0
array([[[ 0.,  0.],
        [ 0.,  0.]],

       [[ 0.,  0.],
        [ 0.,  0.]]])
```

   Les fonctions `zeros_like(T)` et `ones_like(T)` sont très utile pour initialiser des tableaux de 0 ou de 1 ayant les mêmes dimensions qu'un autre tableau `T` (évitant ainsi d'avoir à récupérer les dimensions de ce tableau si on devait utiliser `zeros` ou `ones`).

4. La fonction `eye` permet d'obtenir des tableaux "identités" :

```
>>> eye(3)         # matrice identité 3 x 3 equivalent à eye(3,3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> eye(3,2)       # matrice de la surjection canonique de R^2 -> R^3
array([[ 1.,  0.],
       [ 0.,  1.],
       [ 0.,  0.]])
```

5. La fonction `diag` permet de construire des tableaux 2d en spécifiant une diagonale sous forme d'un tableau 1d ou bien d'extraire une certaine diagonale d'un tableau 2d sous la forme d'un tableau 1d :

```
>>> diag(ones(3))         # en entrée un tableau 1d de 1 => création d'un tableau 2d
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> diag(ones(3),k=1)  # ici on précise la diagonale (par défaut k=0)
array([[ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  0.]])
```

1. Elles ont souvent le même nom que les fonctions équivalentes de scilab ou matlab.

```
>>> A = array([[1,2,3],[4,5,6],[7,8,9]])  # un tableau 2d
>>> print(A)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> diag(A)       # extraction de la diagonale principale
array([1, 5, 9])
>>> diag(A,k=-1)  # extraction de la sous-diagonale
array([4, 8])
```

6. On a déjà vu à l'exercice 3 que le sous-module `random` de `numpy` permet d'obtenir des nombres aléatoires, `random.rand` pour générer des nombres aléatoires de loi uniforme $U(0,1)$, `random.randn` pour la loi normale centrée réduite ($N(0,1)$) et `random.randint` pour une loi uniforme sur un intervalle d'entiers. **Attention** pour que Spyder ressemble un peu plus à Matlab, l'importation du module numpy dans la console python par Spyder modifie en fait certains noms, et le sous-module random n'existe plus, les fonctions `random.rand`, `random.randn`, `random.randint` étant directement accessibles sans le préfixe `random.`.

   Exemples :

```
>>> rand(4)   # pour un tableau 1d à 4 éléments
array([ 0.90538856,  0.23389902,  0.17182099,  0.81463087])
>>> randn(3,3) # pour un tableau 2d 3 x 3
array([[-0.15114084,  0.09718998, -0.81857774],
       [-0.05252272,  0.02244689,  1.01026781],
       [ 0.45428138,  0.64223423,  0.58867134]])
>>> randint(0,9,80)   # loi uniforme sur les entiers de [0,9[ (donc [0,8])
array([4, 2, 5, 4, 5, 2, 8, 8, 1, 7, 0, 1, 1, 5, 3, 7, 5, 2, 3, 2, 6, 5, 3,
       2, 1, 2, 6, 8, 0, 1, 4, 5, 0, 2, 6, 7, 5, 6, 6, 3, 5, 5, 6, 8, 0, 4,
       2, 6, 4, 3, 8, 2, 2, 8, 3, 8, 8, 3, 5, 2, 0, 4, 2, 6, 2, 6, 4, 4, 5,
       3, 1, 0, 5, 7, 3, 4, 4, 0, 2, 8])
>>> randint(0,9,(3,3))  # pour obtenir une matrice 3 x 3
array([[3, 6, 8],
       [6, 4, 0],
       [5, 3, 4]])
```

### 3.1.2 Interrogation des tableaux

Les attributs `ndim`, `shape` et `dtype` permettent de connaître respectivement le nombre de dimensions d'un tableau, le nombre d'éléments dans chaque dimension et le type des éléments du tableau. Exemples :

```
>>> T.ndim    # ou np.ndim(T) pour la version fonction
1
>>> U.ndim
2
>>> T.shape   # ou np.shape(T) pour la version fonction
(5,)
>>> U.shape
(2, 3)
>>> T.dtype
dtype('float64')
```

Noter aussi que la fonction `len` retourne le nombre d'éléments de la première dimension d'un tableau ce qui permet d'écrire des codes qui peuvent fonctionner aussi bien sur une liste que sur un tableau 1d.

### 3.1.3 Modification du profil d'un tableau

Les dimensions d'un tableau et/ou le nombre de dimensions (passage d'un tableau 1d à une tableau 2d, etc...) peuvent être changées avec la fonction ou la méthode `reshape` **si** le nombre total de composantes ne change pas. Exemples :

```
>>> T = arange(12)   # tableau 1d à 12 composantes
>>> T
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> T.shape
(12,)
>>> TT = reshape(T, (3,4) )  # TT sera un tableau 2d (3,4)
>>> TT
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> TT.shape
(3, 4)
>>> TT = reshape(T, (2,3,2) )  ## TT sera un tableau 3d (2,3,2)
>>> TT
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]]])
>>> TT.shape
(2, 3, 2)
```

**Attention :** le tableau `TT` obtenu par cette fonction partage en général ses valeurs avec le tableau initial `T` :

```
>>> T[1] = -1  # modif
>>> T
array([ 0, -1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> TT         # gag !
array([[[ 0, -1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]]])
```

Bon cela peut paraître bizarre mais dans la pratique ce n'est pas génant et si on veut vraiment un autre tableau on utilise la méthode copy suivante. Noter que `reshape` est pratique pour transformer un tableau 1d en tableau 2d à une seule colonne.

### 3.1.4 Copie d'un tableau

La copie d'un tableau s'obtient grâce à la méthode `copy`, exemple `A = U.copy()`.

### 3.1.5 Référencer les éléments d'un tableau

Pour les tableaux 1d, la syntaxe est la même que pour les listes, i.e. `T[1]` désigne le 2 ème élément du tableau, `T[0:2]` (ou `T[:2]`) désigne le sous-tableau formé par les éléments d'indice 0 et 1, etc...

Pour les tableaux 2d et plus, on utilise la syntaxe suivante [2] `U[1,2]`. Si on veut désigner toute la colonne d'indice 1 on écrit `U[:,1]` et si on veut référencer la ligne d'indice 0 à partir de la colonne d'indice 1 on écrit `U[0,1:]`, etc...

Il est possible d'obtenir des sous-tableaux en utilisant des vecteurs d'indices entiers ou des vecteurs de booléens [3].

**Attention :** un tableau 1d n'est pas un tableau 2d particulier comme avec matlab ou scilab (vecteur ligne ou colonne) et lorsque vous utilisez une expression comme `x = U[:,1]`, l'objet référencé par `x` est un tableau 1d [4]. Ces remarques sont importantes pour la suite.

### 3.1.6 les différentes multiplications

1. L'avantage des tableaux c'est qu'on peut utiliser des expressions comme en algèbre linéaire du genre $\alpha u + \beta v$ où $\alpha$ et $\beta$ sont des scalaires et $u$ et $v$ des vecteurs ou matrices. Si `alpha` et `beta` sont des scalaires et `u` et `v` des tableaux de même forme (mêmes dimensions) alors `alpha*u + beta*v` a l'effet attendu.
2. Multiplication de deux tableaux au sens élément par élément : l'opérateur `*` correspond à cette multiplication et donc pas à la multiplication matricielle ! Ainsi lorsque `A` et `B` sont deux tableaux de même forme, `C = A*B` est le tableau dont les éléments sont égaux à $C_i = A_i B_i$ dans le cas 1d et $C_{i,j} = A_{i,j} B_{i,j}$ dans le cas 2d, etc... Cela fonctionne aussi avec la division et la puissance `**`. Et bien sûr on dispose de l'addition et la soustraction entre tableaux !

   **Attention :** lorsque `A` et `B` n'ont pas la même forme une erreur n'est pas forcément obtenue car l'opération peut être "étendue" (broadcasting) dans certains cas comme par exemple lorsque $A$ est un tableau 1d de taille 3 et $B$ un tableau 2d de taille $2 \times 3$ alors l'opération consiste à multiplier chaque ligne de $B$ par $A$ (au sens élément par élément).
3. la multiplication matricielle : elle s'obtient avec la fonction `dot` :
   (a) Pour deux tableaux 2d `A` et `B`, `np.dot(A,B)` fonctionne donc si le nombre de colonnes de `A` est égal au nombre de lignes de `B`.
   (b) Si `x` est un tableau 1d de taille $n$, et `A` un tableau 2d de taille $m \times n$, `np.dot(A,x)` effectue le produit $Ax$, le résultat étant un tableau 1d (donc pas un vecteur colonne). Et si `y` est un tableau 1d de taille $m$, `np.dot(y,A)` effectue le produit $y^\top A$ le résultat étant un tableau 1d (et pas un vecteur ligne).
   (c) Si `x` et `y` sont deux tableaux 1d de même taille `np.dot(x,y)` effectue le produit scalaire $(x|y)$.
4. produit extérieur : Si `x` est un tableau 1d de taille $m$ et `y` un tableau 1d de taille $n$, `A = np.outer(x,y)` renvoie un tableau 2d de taille $m \times n$ avec $A_{i,j} = x_i y_j$.

2. Différente d'une liste de listes où on écrirait `U[1][2]`. Attention cette syntaxe peut fonctionner mais peut désigner aussi bien $U_{1,2}$ que $U_{2,1}$ en fonction du stockage du tableau en mémoire ; en effet un tableau 2d `numpy` peut être "rangé" aussi bien colonne par colonne que ligne par ligne : bref il faut toujours utiliser la syntaxe `U[1,2]` sauf si vous savez exactement ce que vous faites !

3. Un exemple : `U[U>0.5] = 0.5` affectera la valeur 0.5 à la place de tout coefficient du tableau `U` qui serait strictement supérieur à 0.5.

4. Si jamais vous vouliez disposer d'un tableau 2d avec une seule colonne il faudrait écrire `x = U[:,1:2]`.

### 3.1.7 autres fonctions ou méthodes qui pourront nous servir

En vrac :

1. Concaténation de tableaux, on utilise la fonction `concatenate`. Attention les tableaux à concaténer doivent être mis dans un tuple qui est alors le premier argument la fonction. D'autre part on ne peut que concaténer des tableaux ayant le même nombre de dimensions (on ne peut pas directement concaténer un tableau 1d avec un tableau 2d).

```
>>> A = np.ones((2,2))  # l'argument de ones est un tuple avec les dimensions du tableau
>>> A
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> B = np.zeros((2,2))
>>> B
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> np.concatenate((A,B))   # 1 er arg (ici le seul) = tuple formé par les tableaux
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> np.concatenate((A,B),axis=0)  # 2 eme argument direction de concatenation
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> np.concatenate((A,B),axis=1)
array([[ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  0.,  0.]])
```

2. Transposition : on utilise la méthode `transpose` ou encore l'attribut `T` :

```
>>> A = rand(2,3)
>>> A
array([[ 0.19865153,  0.01653091,  0.44841961],
       [ 0.33224291,  0.24958435,  0.68242376]])
>>> B = A.transpose()   # A n'est pas modifié
>>> B
array([[ 0.19865153,  0.33224291],
       [ 0.01653091,  0.24958435],
       [ 0.44841961,  0.68242376]])
>>> C = A.T             # même effet
```

   Attention de même qu'avec `reshape` le tableau transposé partage ses éléments avec l'ancien tableau et il faut utiliser la méthode `copy` si cela ne convient pas.

3. comme avec matlab ou scilab toutes les fonctions élémentaires usuelles (`exp, log, sin, cos, ...`) s'appliquent sur les tableaux et renvoient un tableau de même taille que l'argument, la fonction ayant été appliquée élément par élément. Vous disposez aussi des constantes $\pi$ et $e$. D'autre part ces fonctions élémentaires de `numpy` s'appliquent aussi sur les scalaires, **il est donc inutile d'importer le module** `math` dès que vous importez `numpy`.

4. quelques méthodes : `sum, prod, mean, var, std, min, max, argmin, argmax, sort, clip, unique`.

## 3.2 Graphiques avec matplotlib

Spyder permet d'obtenir une bonne interaction entre les graphiques matplotlib et l'interprète python [5]. En gros lorsque vous affichez des graphiques, la console est toujours disponible et permet de modifier les graphiques déjà affichés.

Voici une session de commande pour vous faire découvrir la fonction `plot`. Noter que comme pour le module `numpy`, Spyder importe automatiquement `matplotlib` si bien que vous n'avez pas à entrer d'instruction d'importation de module.

La même chose pourrait être réalisée sous la forme d'un fichier module à "quelques détails près" (qui seront rappelés ultérieurement) :

1. il faudrait écrire l'instruction d'importation du module `pylab` (on dispose à la fois de `pyplot` et `numpy`) [a] ;
2. et, en fonction de la façon dont vous avez importé ces modules, de penser à mettre l'éventuel bon préfixe sur les fonctions numpy et pyplot (si vous utilisez "`import pylab`" il faudra rajouter le préfixe `pylab.` sur les fonctions de dessin et les fonctions de `numpy` et si vous utilisez "`from pylab import *`" il n'y pas de préfixe à rajouter.).

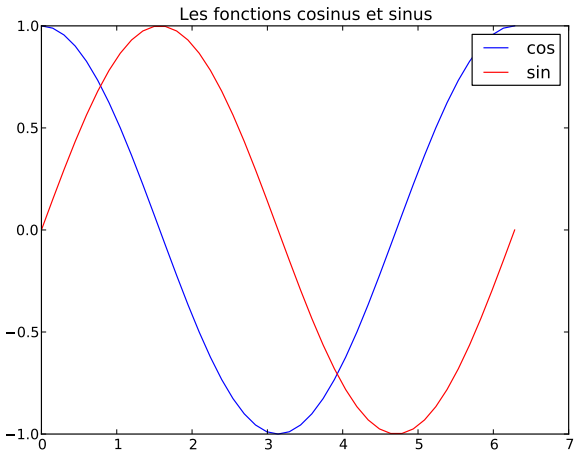
*a.* **matplotlib, pyplot, pylab, numpy ?** Pour faire court `pyplot` est l'interface "à la matlab" de **matplotlib** et c'est elle que nous utiliserons principalement. Le module `pylab` apporte les fonctionnalités de `pyplot` et `numpy` ce qui est pratique.

### 3.2.1 La fonction `plot`

Entrez les commandes suivantes :

```
x = linspace(0,2*pi,43)        # discrétisation de [0,2pi] avec 43 points
y = cos(x); yb = sin(x)        #
l1,l2 = plot(x,y,"b",x,yb,"r") # l1 et l2 sont des objets "lignes"
title("Les fonctions cosinus et sinus")  # title retourne un objet Text
<matplotlib.text.Text object at 0x386fdd0>
legend(("cos","sin"))    # legend retourne un objet Legend
<matplotlib.legend.Legend object at 0x3129090>
```

Vous devriez obtenir ceci :




5. Nous n'en dirons pas plus mais ce n'est pas forcément le cas avec d'autres environnements python, par exemple avec IDLE.

Quelques commentaires :

— vous devez voir apparaître les courbes dès que vous avez entré `plot(x,y,"b",x,yb,"r")` ; par la suite un titre et une légende s'affichent [6] ;

— la fonction `plot` retourne deux objets de type "ligne 2D" :

```
type(l1)
<class 'matplotlib.lines.Line2D'>
```

qui peuvent être utile pour modifier par la suite les propriétés des lignes tracées avec la fonction `setp` , exemples :

```
setp(l1, linewidth=3, color = "g")
[None, None]    # retour de la fct setp (ne pas entrer)
setp(l2, linestyle="--")
[None]
```

néanmoins la légende n'est pas mise à jour automatiquement (au moins avec la version que j'utilise), il est donc préférable d'afficher la légende une fois que toutes les courbes ont leur aspect définitif.

— la fonction `legend` admet de nombreux paramètres... Par exemple il est possible de la positionner sur le graphique à l'aide de l'argument optionnel nommé `loc=`, voir `help(legend)`.

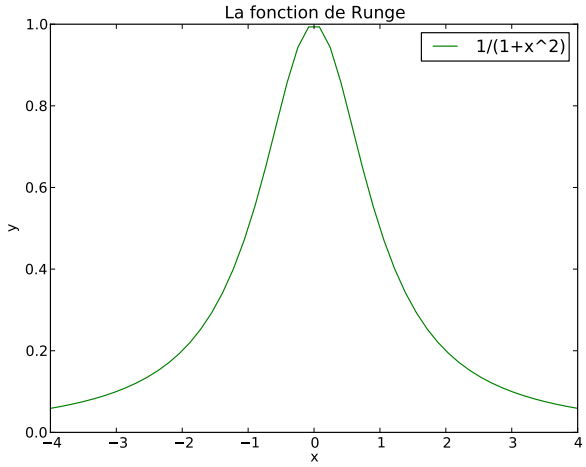
### 3.2.2 Jouer avec plusieurs fenêtres graphiques

Par défaut le graphique s'est affiché dans la fenêtre 1 mais il est possible d'avoir plusieurs fenêtres graphiques en utilisant la fonction `figure` :

```
figure(2)   # autre figure
<matplotlib.figure.Figure object at 0x27bb1d0>
xx = linspace(-4,4,50)
yy = 1 / (1 + xx**2)
plot(xx,yy,"g")
[<matplotlib.lines.Line2D object at 0x27ea5d0>]
legend(["1/(1+x^2)"])
<matplotlib.legend.Legend object at 0x27bb950>
title("La fonction de Runge")
<matplotlib.text.Text object at 0x27d3bd0>
xlabel("x")
<matplotlib.text.Text object at 0x27c8850>
ylabel("y")
<matplotlib.text.Text object at 0x27cc790>
```

Vous devriez obtenir ceci :

6. Le comportement serait légèrement différent si les instructions avaient été écrites dans un fichier puis exécutées car le dessin aurait eu lieu à la fin de l'exécution et le titre, la légende, les courbes, ... auraient été dessinés "en même temps".




Avant d'aller plus loin on peut s'arrêter sur l'instruction `yy = 1 / (1 + xx**2)` qui contient de nombreux raccourcis d'écriture tolérés par `numpy`. Suite à `xx = linspace(-4,4,50)`, `xx` est un tableau 1d. Le passage à la puissance `xx**2` fonctionne comme on s'y attend, c'est à dire qu'on obtient un nouveau tableau où chaque élément de `xx` est élevé au carré. Plus surprenant est l'opération suivante : `1 + xx**2` où on voit l'addition d'un scalaire et d'un tableau. Il s'agit d'un raccourci où on obtient un nouveau tableau 1d (le scalaire est additionné à chaque élément de `xx**2`). Finalement dans la division, on a aussi le raccourci *scalaire* `/` *tableau 1d* qui est valide.

On peut revenir à la première fenêtre et rajouter un graphique avec :

```
figure(1)   # le fenêtre graphique 1 redevient la fenêtre graphique courante
<matplotlib.figure.Figure object at 0x3cc83d0>
plot(x,sin(x)+cos(x),"g")  # ajout d'une courbe
[<matplotlib.lines.Line2D object at 0x3e832d0>]
```

puis effacer son contenu et finalement la détruire avec :

```
clf()       # effacement de la fenêtre graphique courante
close()     # destruction de la fenêtre graphique courante
```

### 3.2.3 Choisir le style des tracés

Nous avons utilisé la fonction `plot` avec la syntaxe suivante :

`plot(`$x_1, y_1, style_1, x_2, y_2, style_2, \ldots$`)`

où les couples $(x_i, y_i)$ sont deux tableaux 1d donnant les abscisses et ordonnées de la courbe $i$ et $style_i$ est une chaîne de caractère qui permet de choisir une couleur et/ou un style de trait et/ou un style de marqueur. Exemples :

- avec `"ob"` on aura des ronds bleus non reliés entre-eux par des traits ;
- avec `"-r^"` on aura des triangles rouges reliés par des traits pleins rouges ;
- avec `"--g"` on aura des traits tiretés verts ;
- avec `"..y"` on aura des traits pointillés jaunes ;

Vous pourrez voir toutes les possibilités avec un `help(plot)`. Une autre syntaxe très intéressante est celle-ci :

`plot(`*x,y,style, clé1= arg1, clé2=arg2, ...*`)`

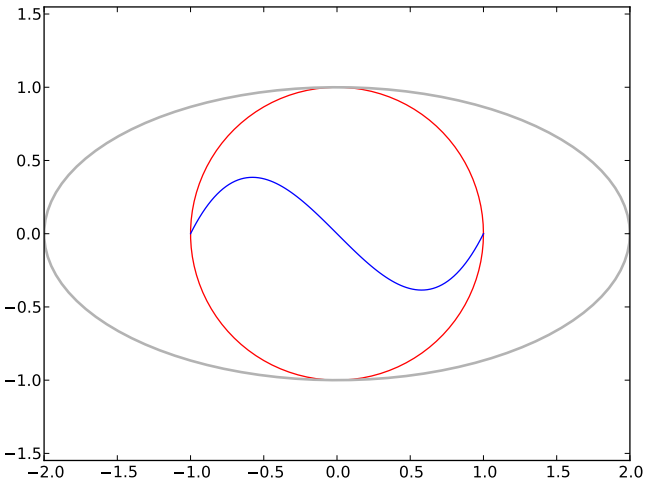
car on dispose alors d'un contrôle accru du style de la courbe en précisant différentes propriétés avec clé=arg, en particulier vous pouvez choisir n'importe quelle couleur avec la clé `color=`, préciser l'épaisseur du trait avec la clé `linewidth=`, et le style de la ligne avec `linestyle=`, voir `help(plot)`. Cette syntaxe évite d'avoir à utiliser la fonction `setp`. Pour les couleurs introduites avec `color=` il y a de nombreuses possibilités :

- avec une chaîne de caractères : `"b"` pour le bleu, `"r"` pour le rouge, etc... mais les noms de couleurs html sont aussi supportés.
- avec une chaîne de caractères de la forme `"0.75"` qui indique un niveau de gris ;
- avec un tuple de flottants `(r,g,b)` où r, g et b sont les intensités de rouge, vert et bleu à prendre dans $[0, 1]$.
- et il y a d'autres possibilités.

### 3.2.4 Echelle isométrique

Parfois on a besoin d'une échelle isométrique, celle-ci s'obtient en utilisant `axis("equal")`. Exemple si on veut tracer sur une même figure, le cercle $x^2+y^2 = 1$ en rouge, l'ellipse $(x/2)^2+y^2 = 1$ en gris clair avec une épaisseur de 2 et la fonction $f(x) = (x-1)x(x+1)$ pour $x \in [-1, 1]$ en bleu, on peut procéder ainsi :

```
clf()
t = linspace(0,2*pi,100)
x1 = cos(t); y1 = sin(t)
x2 = 2*cos(t); y2 = sin(t)
x3 = linspace(-1,1,100); y3 = (x3-1)*x3*(x3+1)
plot(x1,y1,"r",x3,y3,"b")
plot(x2,y2,color="0.7",linewidth=2)
axis("equal")
```




### 3.2.5 Exemple d'affichage de courbes à partir d'un script

Cet exemple reprend des tracés précédents et vous montre une possibilité d'afficher des expressions mathématiques "à la latex" dans vos graphiques. Par exemple pour écrire $\frac{1}{1+x^2}$ j'utilise `$\frac{1}{1+x^2}$`. Dans les fonctions comme `leg`, `title`, `xlabel`, etc... il faut entrer ces expressions latex comme des chaînes de caractères "brute" [7] ce qui s'obtient en faisant précéder les

7. C'est à dire dont aucun caractère ne doit être interprété de manière spéciale.

chaînes en questions par un `r`.

```
from pylab import *

figure(1)
clf()
xx = linspace(-4,4,200)
yy = 1 / (1 + xx**2)
plot(xx,yy,"g")
leg=legend([r"$\frac{1}{1+x^2}$"],fontsize=24)  # utilise une chaine brute
title("La fonction de Runge")
xlab = xlabel(r"$x$",fontsize=24)
ylab = ylabel(r"$y$",fontsize=24)

figure(2)
clf()
t = linspace(0,2*pi,100)
x1 = cos(t); y1 = sin(t)
x2 = 2*cos(t); y2 = sin(t)
x3 = linspace(-1,1,100); y3 = (x3-1)*x3*(x3+1)
plot(x1,y1,"r",x3,y3,"b")
plot(x2,y2,color=(0.7,0.7,0.7,1),linewidth=2)
title("L'oeil")
axis("equal")
```

## 3.3 Exercices sur les tableaux

**Exercice 12** *un tableau trié*

Ecrire une fonction `partition(n)` qui construit un tableau 1d $x$ avec $n$ éléments tels que $x_0 = 0 \leq x_1 \leq x_2 \leq \cdots \leq x_{n-1} = 1$ et où les $x_i$, $i \in [\![1, n-2]\!]$ sont obtenus au hasard uniforme sur $[0, 1]$.

*Aide :* vous pouvez utiliser les fonctions `random.rand`, `concatenate`[8] et la méthode `sort` de `numpy`.

**Exercice 13** *intégration par la méthode des rectangles et par la méthode des trapèzes*

Pour approcher l'intégrale :

$$I = \int_a^b f(x)dx$$

on peut discrétiser l'intervalle $[a, b]$ en $n$ sous-intervalles $[x_k, x_{k+1}]$, $k = 0, \ldots, n-1$ avec $x_k = a+kh$ et $h = (b-a)/n$ et utiliser la formule des rectangles à gauche :

$$I_R(n) = \sum_{k=0}^{n-1} hf(x_k) = h\sum_{k=0}^{n-1} f(x_k) = \frac{b-a}{n}\sum_{k=0}^{n-1} f(x_k)$$

ou la formule des trapèzes :

$$I_T(n) = \sum_{k=0}^{n-1} h\frac{f(x_k)+f(x_{k+1})}{2} = h\left(\frac{1}{2}f(x_0) + \sum_{k=1}^{n-1} f(x_k) + \frac{1}{2}f(x_n)\right)$$

ou beaucoup d'autres pour approcher $I$. Sous des conditions pas trop restrictives on montre que la formule des rectangles est d'ordre 1 ($I_R(n) = I + C_R h + O(h^2)$) et que celle des trapèzes est d'ordre 2 ($I_T(n) = I + C_T h^2 + O(h^4)$). Le but de l'exercice est d'illustrer numériquement ce comportement.

Il faut donc :

- Ecrire deux fonctions, `rectangle(a,b,n,f)` et `trapeze(a,b,n,f)` qui implémentent ces deux méthodes d'intégration. On génèrera les abscisses $x_k$ avec `linspace`. On supposera que la fonction python `f` peut s'évaluer sur un vecteur `numpy` et on utilisera la fonction `sum` de `numpy`. Ces ingrédient permettent d'écrire ces fonctions en quelques lignes sans écrire de boucle.
- Puis on écrira à la suite de vos deux fonctions, des instructions permettant de calculer[9] les erreurs $e_R(n) = |I - I_R(n)|$ et $e_T(n) = |I - I_T(n)|$ pour plusieurs valeurs de $n$ (cad de $h$). En échelle log-log (utiliser `loglog` à la place de `plot`) on devrait alors voir la "courbe" d'erreur pour $I_R$ s'aligner sur une droite de pente 1 (si on trace $\log(e_R)$ en fonction de $\log(h)$) et celle pour $e_T$ s'aligner une droite de pente 2.

**Exercice 14** *permutation au hasard*

Ecrire une fonction `rand_perm(n)` qui retourne une permutation de $[\![0, n-1]\!]$.

*Aide :* cette fonction s'écrit en 4 lignes en utilisant la fonction `range` et la fonction `random.shuffle` de `numpy`.

**Exercice 15** *découvrir les cycles d'une permutation*

8. pour rajouter le tableau `array([0,1])` aux $n-2$ nombres aléatoires.
9. Sur un exemple dont on connait l'intégrale exacte !

Soit $p$ permutation de $[\![0, n-1]\!]$. $p$ est connue sous la forme d'un tableau, $p[k]$ donnant l'image de $k$. Souvent une permutation est aussi représentée de cette manière :

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 1 & 6 & 5 & 3 & 4 & 0 \end{pmatrix}$$

d'un tableau à 2 lignes où chaque colonne représente $\begin{bmatrix} k \\ p[k] \end{bmatrix}$. Les cycles de cette permutation sont :

$$0 \to 2 \to 6 (\to 0)$$
$$1 (\to 1)$$
$$3 \to 5 \to 4 (\to 3)$$

et forment une partition de $[\![0, 6]\!]$, les sous-ensembles formant cette partition $\{0, 2, 6\}, \{1\}, \{3, 4, 5\}$ étant invariants par la permutation.

Le but de l'exercice est d'écrire une fonction `cycles_perm(p)` qui retourne la liste des cycles d'une permutation, chaque cycle sera lui même donné dans une liste (plutôt qu'un tableau 1d). Le principe de l'algorithme est assez simple : étant donné un entier $k \in [\![0, n-1]\!]$ qui ne fait pas encore partie d'un cycle (cad non marqué) :

1. on crée une liste dont le premier élément est $k$ (et on "marque" $k$)
2. on ajoute dans cette liste (et on marque) les entiers $p[k]$, $p[p[k]]$, ...jusqu'à ce que l'on retombe sur $k$. On ajoute alors cette liste dans la liste des cycles.

Il faut recommencer l'opération tant que tous les cycles n'ont pas été trouvés (cad tant qu'il existe un $k \in [\![0, n-1]\!]$ non "marqué"). Pour "marquer" les entiers de $[\![0, n-1]\!]$ on peut utiliser un tableau de booléen initialisé à faux, ce qui s'obtient en utilisant la fonction `zeros` de `numpy` avec un deuxième argument qui permet de donner un type au tableau (ici `bool`). Marquer un entier $k$ s'écrit alors `marque[k] = True`.

## 3.4 Corrections

Fichier module des fonctions des exercices 12, 14, 15.

```
# -*- coding: utf-8 -*-
import numpy as np

def partition(n):
    x = np.concatenate((np.array([0,1]),np.random.rand(n-2)))
    x.sort()
    return x

def rand_perm(n):
    p = range(n)
    np.random.shuffle(p)
    return p

def cycles_perm(p):
    n = len(p)
    marque = np.zeros(n,bool)
    les_cycles = []
    while 1:
        # recherche du point de départ du prochain cycle
        k = 0
        while k < n and marque[k]: k = k+1
        if k >= n: break
        # decouverte du cycle partant de k
        cycle = [k]
        marque[k] = True
        j = p[k]
        while j != k:
            cycle.append(j)
            marque[j] = True
            j = p[j]
        les_cycles.append(cycle)
    return les_cycles
```

Fichier module pour l'exercice sur l'intégration.

```
# -*- coding: utf-8 -*-
from pylab import *

def rectangle(a,b,n,f):
    x = linspace(a,b,n+1)
    return (b-a)*sum(f(x[:-1]))/n

def trapeze(a,b,n,f):
    x = linspace(a,b,n+1)
    return (b-a)*( 0.5*f(x[0]) + sum(f(x[1:-1])) + 0.5*f(x[n]) )/n

# on intégre exp(x) sur [0,pi] d'où Ie = exp(1)-1

a = 0; b = 1; Ie = exp(1)-1
nn = array([10, 20, 40, 80, 160, 320, 640, 1280])
m = len(nn)
er = zeros((2,m))

for k in range(m):
    er[0,k] = fabs(rectangle(a,b,nn[k],exp) - Ie)
    er[1,k] = fabs(trapeze(a,b,nn[k],exp) - Ie)

clf()
loglog(nn,er[0,:],'ob',nn,er[1,:],'xr')
legend(('erreur rectange','erreur trapèze'))
```

## 3.5 Compléments sur les modules

### 3.5.1 Exécution d'un module depuis l'interprète

> Spyder simplifie beaucoup l'exécution de programmes python. Cette partie pourra être lue ultérieurement.

Jusqu'à présent nous avons utilisé la touche F5 de Spyder, il s'agit maintenant d'en savoir un peu plus [10]. Pour exécuter un module à partir de l'interprète (c'est de la fenêtre de commande) on peut utiliser une instruction `import` :

— avec :

```
import nom_du_module  # attention ne pas mettre le suffixe .py
```

le module est exécuté la première fois (s'il ne comporte pas d'erreur de syntaxe). A la suite vous pouvez utiliser, dans la fenêtre de commande, les objets définis par ce module avec le préfixe `nom_du_module.`

— et avec :

```
from nom_du_module import *
```

le module est exécuté la première fois (s'il ne comporte pas d'erreur de syntaxe) mais dans ce cas on peut utiliser les objets définis sans le préfixe `nom_du_module.`.

**Question :** comment python peut savoir où se trouve votre module ? Lorsque l'interprète rencontre une commande `import toto` il cherche donc le fichier `toto.py` en premier dans le répertoire "courant" et sinon il utilise les répertoires listés dans la variable d'environnement PYTHONPATH.

Pour faire simple nous utiliserons uniquement la première possibilité et si le fichier module que vous voulez exécuter n'est pas dans le répertoire "pointé" par l'interprète, on changera ce dernier de la façon suivante :

```
import os
os.getcwd()  # renvoie le répertoire courant
os.chdir(nouveau_repertoire) # change le répertoire courant pour nouveau_repertoire
```

**Rechargement d'un module :** bon vous avez modifié votre fichier module et vous voulez le ré-exécuter sans sortir de l'interprète. Or lors d'une même session python un module n'est exécuté qu'une seule fois avec `import` (le deuxième, le troisième, etc, import ne font rien !), et, en cours de développement on a pas forcément envie de quitter la session python puis d'en ouvrir une nouvelle. Pour cela il faut utiliser [11] `reload(`*nom_module*`)`. Attention :

— si vous avez importé le module en lui donnant un autre nom, comme par exemple avec `import toto as titi`, il faudra utiliser `reload(titi)` ;

— si vous avez importé le module avec `from toto import *`, il faudra utiliser :

```
reload(toto)
from toto import *
```

Attention `reload` n'est plus une "built-in" de python 3.x il faut l'importer du module imp (`from imp import reload`).

10. Spyder utilise en fait une fonction qu'il définit lui-même, à savoir `runfile` mais elle n'est pas définie directement par python.

11. `reload` est une fonction, il faut entourer son argument par des parenthèses.

### 3.5.2 Portée des objets dans un module

Un objet défini dans un module en dehors d'une fonction peut être utilisé en lecture dans les fonctions mais ne peut être modifié par une fonction. Exemple :

```
# module truc
A = 1

def func1(x):
   y = A + x     # utilise le A précédent
   return y

def func2(x):
   A = 3         # défini un objet A local à la fonction
   y = A + x     # utilise cet objet
   return y

def func3(x):
   y = A + x     # génèrera une erreur à l'exécution (cf explications)
   A = 3
   return y

print(func1(2))  # on obtient 3
print(func2(2))  # on obtient 5
print(A)         # toujours 1
print(func3(2))  # génère une erreur
```

Voici ce que donne l'exécution de ce module :

```
3
5
1

Traceback (most recent call last):
  File "/home/bruno/Enseignement/Python/truc.py", line 22, in <module>
    print(func3(2))  # génère une erreur
  File "/home/bruno/Enseignement/Python/truc.py", line 15, in func3
    y = A + x     # génèrera une erreur à l'exécution (cf explications)
UnboundLocalError: local variable 'A' referenced before assignment
```

En effet l'instruction d'affectation `A = 3` dans `func3` rend cette variable locale à la fonction. Et, lorsque la deuxième instruction de cette fonction est exécutée cette variable locale n'a pas encore été initialisée.

Si on veut pouvoir modifier des variables "globales" du module dans une fonction, il faut utiliser :

```
def func3(x):
   global A      # A globale
   y = A + x     # génèrera une erreur à l'exécution (cf explications)
   A = 3         # modifie la valeur de A
   return y
```

Avec cette modification l'exécution du module affichera :

```
3
5
1
3
```

et si on rajoute à la fin une impression de `A` on verra que sa valeur est modifiée.