M 2 Maths IMSD 2024-2025

Mise à niveau en python

Bruno Pinçon

séances 1 et 2

Pour cette mise à niveau en python j'avais pensé à travailler avec ce langage sans utiliser d'environnement de programmation (en utilisant juste un éditeur et une fenêtre de commande) mais cela s'avère peu pratique/facile dans la salle VG 321 (il y a au moins 3 versions de python sur ces machines). Des éléments seront peut-être donnés vers la fin de ce cours dans ce but mais pour les 5 séances à venir nous allons utiliser l'IDE spyder fourni par la distribution Anaconda de python disponible sur les machines de cette salle. Un des avantages (de cette distribution) est que tous les paquets utiles pour faire du calcul scientifique au sens large (i.e. y compris l'IA, l'analyse de données, etc.) sont disponibles (pas d'installation de modules python). Dans ces séances nous allons commencer tout au début à utiliser le débogueur python pdb (en mode texte de sorte que vous ne soyez pas dépendant de spyder qui possède une surcouche graphique au dessus). Une autre idée est l'autonomie des élèves, vous pourrez suivre en partie ce document à votre rythme, si vous n'avez jamais programmé dans ce langage signalez-vous, je vous apporterai plus d'attention. Enfin un site Arche contiendra les documents et sera complété au fur et à mesure, en particulier, j'essaierai d'écrire un test avec le feed-back nécessaire de manière à consolider vos connaissances. Ce test pourra être refait régulièrement dans l'année pour vos révisions. Il y aura aussi un forum dans lequel vous pourrez poster des questions concernant python, idem ceci tout au long de l'année.

Remarque : pour ces 2 premières séances, ce document est à récupérer sur la page :

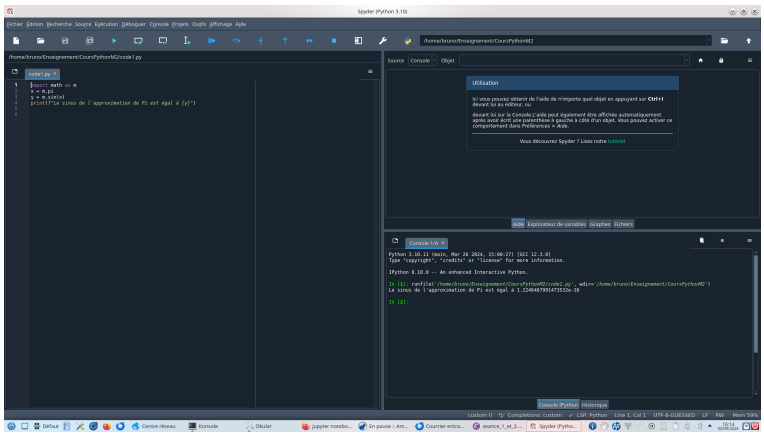
`https://iecl.univ-lorraine.fr/membre-iecl/pincon-bruno/`

(cliquer sur rubrique Enseignement). Vous verrez aussi un lien pour télécharger les deux fichiers python `exemple_debug.py` et `minilibquad.py` que vous déposerez dans un répertoire neuf dans lequel vous travaillerez pour les 5 séances prévues.

Table des matières

Préambule : lancer spyder anaconda sur une machine de la salle

Attention, il y a deux environnements (IDE) spyder disponibles, pour lancer le bon :
— dans la fenêtre de recherche (en bas au milieu de votre écran) rentrer spyder ;
— plusieurs réponses doivent apparaître, il faut choisir “spyder - anaconda”.
Vous devez alors voir apparaître une fenêtre qui ressemble à ceci :



avec :
— à gauche la fenêtre d’édition ;
— à droite en haut une fenêtre qui peut servir au choix, à afficher les graphiques (on fera autrement), à afficher la valeur des variables et d’autres choses encore ;
— à droite en bas une fenêtre avec une instance de l’interprète ipython (sur-couche sur l’interprète python) ;
— une barre avec divers menus tout en haut.
À l’aide de l’éditeur créer le fichier suivant (dans le nouveau répertoire que que vous avez dû créer) :

```
import math as m
x = m.pi
y = m.sin(x)
print(f"Le sinus de l'approximation de Pi est égal à {y}")
```

Pour exécuter le code, cliquer sur la petite flèche verte, une question vous sera posée (choisissez d’exécuter le code dans la fenêtre ipython courante). Dans la sous-fenêtre de l’interprète doit s’afficher la réponse “Le sinus de l’approximation de Pi est égal à 1.2246467991473532e-16”.

1 Le débogueur python

En général quand on écrit un code, il ne fonctionne pas du premier coup et pour comprendre ce qui se passe on va souvent mettre des instructions “print” pour afficher le contenu de certaines variables. Cette méthode fonctionne mais il y a beaucoup mieux : utiliser un débogueur. Un tel outil permet d’exécuter votre code pas à pas, c’est à dire instruction après instruction et/ou de le faire exécuter par blocs en insérant (avec le débogueur) des points d’arrêt (appelés breakpoints), on peut ainsi aller directement au prochain point d’arrêt. Il permet aussi de regarder le contenu des variables très facilement. Une autre idée de cette mise à niveau est d’utiliser systématiquement le débogueur pour mettre au point vos programmes.

1.1 Comment utiliser le débogueur

Pour cela mettre dans votre fichier script l’instruction :

```
import pdb
```

puis plus loin (à l'endroit où vous souhaitez un premier point d'arrêt) :

```
pdb.set_trace()
```

Une fois ces deux instructions écrites, sauvegarder et relancer votre code. Lorsque l'interprète python va rencontrer `set_trace` il va passer en mode débogage, ce qui est signalé par l'invite :

```
(Pdb)
```

Voici quelques commandes (à entrer après l'invite [1]) :

- `l` : visualise quelques lignes de code autour de la ligne où s'est arrêté le débogueur (ligne signalée par une `->`) ;
- `ll` : idem à `l` mais plus de lignes sont affichées ;
- `n` : permet d'aller à la ligne suivante (si vous entrez de nouveau `l` vous verrez que la position de la `->` a changé) ;
- `p` *var_name* : affiche la variable *var_name* ;
- `b` *num_ligne* : ajoute un point d'arrêt (breakpoint) à la ligne *num_ligne* ;
- `c` : permet d'aller directement au point d'arrêt suivant (et de sortir du débogueur s'il n'y a plus de point d'arrêt) ;
- `b` : permet de visualiser tous vos points d'arrêt AINSI que le numéro (*num_bpt*) que le débogueur a attribué à chacun d'entre-eux (différent du numéro de ligne... en général) ;
- `cl` *num_bpt* : retire le point d'arrêt de numéro *num_bpt* ;
- `s` : idem à `n` sauf que le débogueur entre dans les appels de fonctions si votre ligne en comporte.
- `q` : sortie du débogueur (si vous ne voulez pas exécuter votre programme jusqu'au bout ou jusqu'à ce qu'il "plante").

Autre chose importante : après l'invite `(Pdb)` vous pouvez entrer presque n'importe quelle commande python, et modifier les variables actuelles.

> ***Attention :*** *comme on travaille avec* ipython *il faut faire précéder toutes les commandes de débogage par un point d'exclamation* `!n`, `!l`, *etc.* Parfois sans le `!` cela peut fonctionner mais autant assurer le coup !

1.2 Un exemple

Le fichier `exemple_debug.py` (que vous avez dû déjà télécharger) va nous servir pour tester quelques commandes du débogueur. Ce code est (normalement) correct, lire ce code très simple puis l'exécuter en choisissant une valeur élevée pour m (par exemple $m = 100000$). La lgn nous dit qu'on devrait avoir des proportions d'environ 0.2 (resp. 0.1) de nombres qui tombent dans le premier (resp. le deuxième) intervalle.

Ensuite décommenter la ligne `#pdb.set_trace()` (i.e. enlever le caractère dièse). Puis relancer le script. L'exécution s'arrête alors à l'instruction qui suit `pdb.set_trace()` (qui n'est pas (encore) exécutée). Vous pouvez entrer la commande `!n` pour aller à la ligne suivante. L'interprète exécute la ligne `m = int(input(" m = "))`, vous pouvez entrer 10 par exemple. L'exécution s'arrête après cette instruction. Vous pouvez alors :

- vérifier le contenu de la variable `m` avec : `!p m`
- voir une partie du code autour de la position actuelle avec : `!l`

1. Appuyer sur la touche entrée/return de votre clavier ensuite...

— si vous entrez : `!p int1`, l'erreur suivante est obtenue : `NameError: name 'int1' is not defined`, c'est normal (pourquoi ?).

De nouveau aller à la ligne suivante avec `!n`. Vous pouvez maintenant vérifier le contenu de la variable `int1` (avec `!p int1`). On peut aussi (par exemple) vérifier le type de cette varable avec `type(int1)` (rappellez vous qu'on a le droit d'entrer des commandes python).

Installation de deux points d'arrêt :

— Vérifier en tapant `!ll` si les lignes sont comptées à partir de 0 ou de 1. Je ne sais pas si c'est un bug mais (en tout cas sur ma machine sous linux) la première ligne est numérotée 0 mais le débogueur lui considère que c'est la ligne 1. Si vous êtes dans ce cas, il faut donc ajouter 1 au numéro de ligne qui apparait dans le terminal pour y obtenir un point d'arrêt. Installer un breakpoint sur la ligne `if test_intervalle(u, 0.55, 0.75):` puis un deuxième sur la ligne `print(f" proportion dans [0.55, 0.75] = nb_int1/m")`, soit :
  — `!b 26`
  — `!b 30`

  La commande `!b` vous donnera la liste constituée des (2) breakpoints.

— Entrer maintenant la commande `!c` : le débogueur exécute le code jusqu'à cette ligne. Là regarder le contenu de la variable `u` (commande `!p u`).

— Pour aller dans la fonction `test_intervalle` il faut entrer la commande `!s` (comme step into) ; dans cette fonction vous pouvez avancer avec des commandes `!n` pour voir si elle s'exécute comme elle le doit (n'hésitez pas à utiliser des `!l` si vous êtes perdu).

— Lorsque l'exécution sort de la fonction entrer de nouveau la commande `!c`, que se passe-til ?

— Enlever le premier breakpoint avec la commande `!cl 1`. De nouveau vérifier la liste des breakpoints (commande `!b`) il ne doit en rester qu'un !

— Maintenant entrer la commande `!c` que se passe-t-il ?

Voilà pour l'initiation au débogueur, vous verrez par la suite que l'investissement en vaut le coup (un petit effort à faire au début mais énormément de temps gagné par la suite).

> *Pour la suite de cette séance, les élèves qui ont déjà de bonnes notions du langage peuvent se rendre directement à la section suivante "Exercices" (14), sinon vous pouvez réviser et apprendre les bases dans la section suivante puis passer aux exercices. Il est aussi possible de faire des allers et retours entre les exercices et les rappels.*

2 Rappels et compléments sur Python

2.1 Quelques types de base

Voici une session interactive (inutile d'écrire cela dans un fichier mais directement dans la console ipython) concernant quelques types de base et des opérations sur ces types. Rmq : inutile d'entrer les commentaires !

```
# les entiers
a = 5
type(a)   # doit retourner int
b = 11
a + b # addition
a - b # soustraction
a*b   # multiplication
b/a   # division => on obtient un flottant
b//a  # division euclidienne
b%a   # modulo
q, r = divmod(b,a) # division euclidienne complète
q, r  # équivalent à print(q,r)
120**20  # puissance (les entiers de python n'ont pas de limite)
```

```
# les flottants (approximations des nombres réels sur ordinateur)
x=0.2
type(x)  # doit retourner float
print(x)
print(format(0.2,"24.17e")) # on découvre que 0.2 n'a pas été stocké exactement...
# les 4 opérations usuelles et la puissance sont les mêmes que pour les entiers

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)
m=sys.float_info.min  # plus petit nombre flottant normalisé (mis dans la variable m)
print(m)
eps = sys.float_info.epsilon # deux fois la précision relative max du codage d'un réel par un flottant
print(eps)
Inf = 2*M  # on obtient le nb flottant spécial inf qu'on l'on référence Inf
Inf - Inf  # on obtient le nb flottant spécial Nan

# les chaînes de caractères
x = "x"
type(x)  # doit retourner str
y = "y"
x+y  # concaténation
x*5  # répétition
toto = "éêèà"
print(toto)
toto[0]  # première lettre de la chaîne
toto[-1] # dernière lettre de la chaîne
toto[1] = '4'  # erreur les chaînes de caractères sont non modifiables

# les booléens
b = 4 < 6
type(b)   # doit retourner bool
True or False  # ou logique
True and False # et logique
not False      # non logique

# les fonctions usuelles sont dans le module math
import math as m  # pour avoir un préfixe plus court
dir(m)            # voir ce qui est disponible dans le module math
help(m)           # doc minimale des fonctionnalités du module
help(math.expm1)  # doc de la fonction expm1 du module math
m.sin(m.pi)       # m.pi est l'approcimation flottante de pi
m.sqrt(2)
m.exp(-1)
m.log(1.0001)  # doit être proche de 0.0001
m.factorial(5) # 5! = 120
```

2.2 Quelques types "conteneurs" : listes, tuple

Les listes servent à faire des collections ordonnées d'objets de type quelconque. Comme les chaînes, les listes sont indicées à partir 0 mais contrairement aux chaînes, les listes sont des objets modifiables et peuvent contenir des objets de types tous différents. Les tuples sont comme des listes non modifiables. Enfin si les listes sont très versatiles, elles ne sont pas très efficaces pour faire du calcul numérique d'où l'utilisation des tableaux numpy dans ce cas.

2.2.1 Une session interactive sur les listes

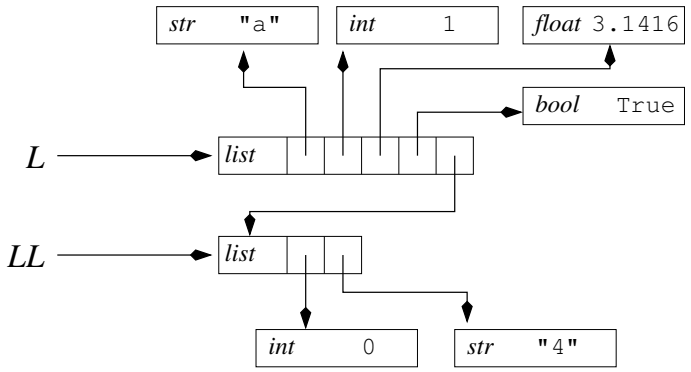
```
# création d'une liste
```

```
L = [ "a", 1, 3.1416, True ]
type(L)
print(L)
L[0]
L[2]
len(L)  # retourne le nb d'éléments de la liste
LL = [0,"4"]        # une autre liste

# ajout d'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)
L.append("toto")     # encore un nouvel élément
print(L)
L.append("titi")     # et encore un autre
print(L)
len(L)

# on peut détruire un élément avec l'instruction {\tt del}
del L[5]
print(L)  # "toto" est détruit et "titi" devient l'élément en position 5
del L[5]
print(L)
```

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
L                       # L est aussi modifiée
L[4][1] = 912           # autre modification
LL                      # LL est aussi modifiée
```

Pour voir les fonctions et/ou méthodes qui s'appliquent aux listes, entrer `help(list)` dans la console ipython.

2.2.2 Une session interactive sur les tuples

Quelques mots maintenant sur les tuples. Les tuples sont comme des listes non modifiables. Ils sont assez utilisés comme argument de fonctions mais ils jouent un grand rôle comme valeur de retour des fonctions (on verra cela un peu plus tard). Nous verrons aussi qu'ils permettent de donner les dimensions (le profil) d'un tableau dans les fonctions numpy.

```
# création d'un tuple
T = ( "a", 1, 3.1416, True ) # on met des ( ) à la place des [ ]
type(T) # doit retourner tuple
```

```
print(T)
# mais en fait on a pas besoin des parenthèses
U = 4,  "toto", 2.71, False
type(U)
print(U)
# tentative de modification du 2 ème élément de U => python génère une erreur
U[1] = "titi"
# attention quand même si vous voulez un tuple ne comportant qu'un seul élément
V = (5)
type(V) # doit retourner int
V = (5,) # là ça doit être OK
type(V)
print(V)
W = True,  # idem
type(W)

# trucs utiles avec les tuples
# Imaginez que ces 3 nombres correspondent aux paramètres d'un algorithme
# par exemple une erreur relative max, une erreur absolue max, un nombre d'itérations max
param = (1e-6, 1e-3, 200)
# comment sortir les paramètres du tuple ?
# 1/ solution évidente :
epsr = param[0]
epsr  # en session interactive ceci est équivalent à print(epsr)
epsa = param[1]
epsa
itermax = param[2]
itermax
# 2/ il y a mieux avec la solution ''pythonique'' :
epsr, epsa, itermax = param
# la ligne suivante génère une erreur
epsr, autres_parametres = param
# par contre en ajoutant une * autres_parametres collecte les autres éléments du tuple de droite
epsr, *autres_parametres = param
print(autres_parametres)
# échange du contenu des 2 variables (il faut dire échange des 2 références plutôt)
a = "Vladimir"
b = "Bruno"
# la solution classique
temp = a
a = b
b = temp
print(a,b)
# la solution en utilisant les tuples
a, b = b, a   # trop facile
print(a,b) # attention la référence a ''repointe'' maintenant sur l'objet chaîne  "Vladimir"
           # et la référence b ''repointe'' sur l'objet chaîne "Bruno"
```

2.3 Éléments de programmation : tests, boucles, fonctions et modules.

2.3.1 Les modules et les fonctions

Chaque fichier python est ce qu'on appelle un module python (de même qu'une session interactive qui est un module spécial). Il s'agit donc de l'unité de programmation de ce langage. Un module peut définir des fonctions, des variables, des nouveaux types et méthodes (programmation objet), faire des calculs et peut utiliser les fonctionnalités d'autres modules grâce à l'instruction `import`. Supposons que vous écriviez un nouveau module dans un fichier dont le nom est `toto.py` (le suffixe `.py` est obligatoire) alors ce module a pour nom `toto`. Et supposons aussi que ce module définisse les fonctions `fun1`, `fun2`. Vous écrivez maintenant dans un autre fichier dont le nom est `titi.py` et qui se situe dans le même

répertoire [2] que `toto.py` et vous voulez utiliser vos deux fonctions `fun1` et `fun2` dans le module `titi`. Comment faire ? Il y a plusieurs solutions qui consistent toutes à écrire une instruction d'importation (placée en général au début dans le fichier `titi.py`) :

```
# solution 1
import toto
# par la suite pour utiliser fun1, il faut écrire toto.fun1

# solution 2
import toto as tot #
# par la suite pour utiliser fun1, il faut écrire tot.fun1

# solution 3 (j'aime pas les préfixes)
from toto import fun1, fun2
# par la suite pour utiliser fun1, il faut juste écrire fun1

# solution 4 (j'aime pas les préfixes mais je n'aime pas les noms fun1 et fun2
from toto import fun1 as f1, fun2 as f2
# par la suite pour utiliser fun1 il faut juste écrire f1

# solution 5 (j'aime pas les préfixes)
from toto import *
# par la suite on utilise fun1 pour fun1
```

La dernière solution n'est pas recommandée sauf éventuellement pour une session interactive ou lorsque le module en question (ici toto) ne contient pas grand chose.

Reprenons le premier exemple :

```
import math as m
x = m.pi
y = m.sin(x)
print(f"Le sinus de l'approximation de Pi est égal à {y}")
```

Si vous l'avez écrit dans un fichier dont le nom est `essai1.py`, il définit donc un module dont le nom est `essai1`. On peut voir que ce module ne fait que définir quelques objets (les scalaires flottants x et y). Un tel module ne définissant ni fonctions, ni nouveaux types, etc, sera généralement appelé un script. Dans un projet de programmation usuel, on est amené à travailler avec plusieurs fichiers :

— Une partie des fichiers vont définir des fonctions (et/ou de définir la valeur de certaines variables et/ou de définir de nouveaux types d'objets) permettant de faire des "calculs" utiles. Ces fichiers, on pourrait les qualifier de "vrais" modules (et dont l'ensemble pourrait former une bibliothèque ou ce qu'on appelle un package python).

— Les autres fichiers (qui seront appelés plutôt des script), font faire les calculs effectifs en construisant les données dans le script (peut-être via la lecture de fichiers de données) puis appeler les bonnes fonctions (ou certaines de leurs fonctionnalités) définies dans les modules précédents et afficher des résultats (ou les écrire dans un fichier), des graphiques, etc. Dans un vrai projet il y a aussi des fichiers scripts pour vérifier le code de la bibliothèque (tests unitaires, etc.).

Par exemple on peut imaginer écrire un module avec une ou des fonctions capables de résoudre un type de problème d'optimisation $\mathcal{P}$ comme résoudre un PL en forme standard :

$$\max_{x\in\mathcal{C}} f(x) = c^\top x, \mathcal{C} := \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$$

c'est la partie "bibliothèque réutilsable" (dans cette bibliothèque on peut donc imaginer d'autres fonctions permettant d'écrire tout PL en forme standard, s'il ne l'est pas) et à côté on peut écrire des fichiers scripts qui vont définir chacun un tel problème, par exemple définir ici les vecteurs c, b et la matrice A, puis appeler la fonction de la "bibliothèque" (et donc l'importer) qui résout un tel problème puis affiche les

2. Ce n'est pas obligé mais c'est le cas le plus simple.

résultats. Pour les fichiers de test, on peut s'appuyer sur des problèmes dont on connait la solution et examiner la différence entre le résultat connu et celui calculé par votre fonction de bibliothèque.

Sur les fonctions : dans un module, une définition de fonction s'introduit avec :

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

Voici un exemple d'un module contenant deux fonctions (cf fichier `minibibquad.py` que vous avez dû déjà télécharger) :

```
from math import sqrt   # 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 resout_quad (qui retourne 2 arguments)
def resout_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 (c'est un tuple en fait)
```

Quelques remarques :

— Chaque ligne qui suit un `def` est affublé d'une chaîne de caractères entre triple guillements (avec autant de lignes que vous voulez) et qui va permettre de documenter chaque fonction, c'est ce qu'on appelle une docstring (voir plus loin).

— L'indentation est importante car elle définit la portée de la fonction [3]. 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 [4] 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`.

— L'instruction assert :

3. L'éditeur de Spyder fait cela plus ou moins naturellement, c'est à dire qu'en général, vous n'avez pas à entrer de tabulations ou blancs : il le fait à votre place.

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

`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.

— 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 ou les tableaux numpy peuvent donc être modifiés par une fonction (mais pas un tuple).

Testons ce module dans une session interactive :

```
import minibibquad as mbq   # importation avec modif du nom (qui est un peu long)
dir(mbq) # donne le contenu succinct de ce module
help(mbq) # la doc fournie grâce aux docstrings
x1, x2 = mbq.resout_quad(1,2,1)
print(x1,x2)                # affichage des racines
rac = mbq.resout_quad(1,2,1)    # les 2 arguments de resout_quad sont affectés à une seule variable
print(rac)                  # => rac est alors un t-uple
help(mbq.quad)              # aide sur quad (affiche la docstring)
# tester d'autres exemples d'appels
```

La documentation est un peu spartiate mais il y a des outils pour obtenir du documentation html avec lien, etc.

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

2.3.3 les expressions booléennes, les opérateurs de comparaison

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.

2.3.4 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` (voir l'exercice 4 pour des explications). 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]`.

2.3.5 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` (complément donné dans l'exercice 4). Il est possible de sortir autrement de la boucle en utilisant un `if` et un `break`. Exemple [5] :

`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.

2.3.6 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 [6] 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 la fonction `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`.

5. Noter qu'on peut mettre une instruction juste après les `:` des `if`, donc ici on aurait pu écrire `if` *test2*`:` `break`
6. Au nombre de caractères dans le cas d'une chaîne, au nombre de lignes dans le cas d'un tableau 2d.

Compléments : Selon vos habitudes précédentes de programmation, on peut avoir tendance à utiliser systématiquement `range` pour piloter une boucle. Par exemple pour afficher les éléments d'une liste on pourrait utiliser :

```
for k in range(len(L)):
    print(L[k])
```

mais cette solution ajoute une indirection en plus et s'exécutera donc un peu plus lentement. Donc si vous n'avez pas besoin de connaître l'indice d'un élément d'une séquence (liste, tuple,...), inutile de piloter votre boucle via `range`. Il y a un outil qui permet d'itérer directement sur plusieurs séquences : `zip`. Admettons que vous vouliez itérer sur les séquences `S` et `T` en même temps, on peut utiliser :

```
for s,t in zip(S,T):
    # s (resp. t)  est l'élément courant de la séquence S (resp. T)
    # on peut les utiliser dans une expression
```

Plutôt que d'écrire :

```
for k in range(len(S)):
    s = S[k]  # 2 indirections supplémentaires pour disposer de s et t
    t = T[k]
```

Et ceci fonctionne avec autant de séquences que vous voulez `zip(S1, S2, S3, ...)`. Les séquences n'ont pas à avoir autant d'éléments chacune, la boucle comportera autant d'itérations que le nombre d'éléments de la plus petite séquence. Néanmoins, on peut avoir besoin de connaître le numéro de l'itération. Dans ce cas il y a `enumerate`. Supposons qu'on parcoure les éléments d'une liste `L` et que l'on cherche à repérer la position (l'indice) des éléments vérifiant une condition (ici on suppose qu'on appelle une fonction `test_cond` qui prend un élément et retourne un booléen), ces indices étant collectés dans une autre liste (appelée `OK_test`, on peut utiliser :

```
OKtest = []
for index, elem in enumerate(L):
    if test_cond(elem):
       OKtest.append(index)
```

Les indices obtenus sont les classiques, $0, 1, ...$ jusqu'à la longueur de la séquence moins 1 mais `enumerate` peut commencer à partir d'un indice de départ spécifié par l'argument optionnel `start=`, `enumerate(L,start=1)` comptera à partir de 1 (au lieu de 0 par défaut). Et bien sûr on peut "encapsuler" un `zip` avec un `enumerate`, en reprenant l'exemple précédent de `zip` :

```
for index, (s,t) in enumerate(zip(S,T)):
    # s (resp. t)  est l'élément courant de la séquence S (resp. T)
    # et index est le numéro de l'itération
```

Attention de bien mettre les parenthèses autour des éléments de la séquence !

```
for index, s,t in enumerate(zip(S,T)):  # erreur !
```

3 Exercices

Chaque exercice (ou presque) liste les points des rappels et compléments précédents dont vous pourriez avoir besoin, et permet d'y accéder rapidement (cliquer sur le lien ce qui devrait fonctionner avec la plupart des lecteurs pdf). Certains exercices exposent des compléments du langage qui ne sont pas inclus dans les rappels précédents mais sont importants à connaître. ***Pour la mise au point de vos fonctions utilisez le débogueur !*** *La plupart des exercices consistent à écrire une fonction, toutes ces fonctions seront à écrire dans le même fichier (pour la suite je suppose que son nom est* `fonctions.py`*). Prière de faire vos essais, soit directement dans la console ipython, soit dans un autre fichier pour vous habituer à utiliser* `import`*.*

Exercice 1 *Suite de Fibonacci*

On rappelle la suite de Fibonacci $F_0 = 0, F_1 = 1$ et pour $n \geq 2$, $F_n = F_{n-2} + F_{n-1}$. Écrire 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. Aide :

- Utiliser assert pour s'assurer que l'argument est bien un entier positif ou nul (voir l'exemple d'utilisation d'assert de la fonction `resout_quad` au paragraphe des rappels sur la notion de fonction python (p. 9)).
- Pour tester si l'argument est bien un entier positif ou nul on peut utiliser l'expression booléenne :

```
type(n) is int and n >= 0
```

- Pour mettre à jour les variables qui contiendront les valeurs courantes de F_{n-2} et F_{n-1} vous pouvez utiliser une assignation de tuple (plutôt que de passer par une classique variable temporaire), cf les rappels sur les tuples p. 6.
- Les rappels sur les boucles **pour** sont donnés à partir de la p. 12.
- Pour vérifier votre fonction, la page wikipedia (`https://fr.wikipedia.org/wiki/Suite_de_Fibonacci`) nous dit (entre autres) que $F_{10} = 55, F_{15} = 610$. Comme expliqué précédemment, faites vos tests directement dans la console ipython ou écrivez les dans un autre fichier. Par exemple dans un fichier `test_fib.py` on pourrait trouver :

```
""" qq tests pour la fonction fib du module fonctions """
from fonctions import fib
#import pdb       # à décommenter pour déboguer
#pdb.set_trace()  # à décommenter pour déboguer
f10 = fib(10)
f15 = fib(15)
print(f"fib(10) = {f10}")
print(f"fib(15) = {f15}")
# un peu mieux
if f10 == 55 and f15 == 610:
    print("La fonction fib passe le test")
else:
    print("La fonction fib ne passe pas le test")
```

Exercice 2 *Former une liste au hasard*

Écrire 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 et b (inclus tous les deux). Aide :

- Pour obtenir un tel entier aléatoire, on importera le module `random` :
  `import 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 [7].

7. Attention le sous-module random de numpy possède la même fonction mais où l'entier de la borne droite (b) est lui exclus.

— Utiliser `L = []` pour créer une liste vide et la méthode `append` pour rajouter successivement les n entiers aléatoires (les rappels sur les listes sont donnés à partir de la p. 5).
— Les rappels sur les boucles **pour** sont donnés à partir de la p. 12.
— Il est plus difficile de tester cette fonction (si on veut vérifier que le hasard est bien uniforme), par contre on peut vérifier facilement si la liste construite a le bon nombre d'élements, et s'ils sont bien tous des entiers compris entre a et b. Par exemple, on pourrait écrire un fichier `test_randlist.py` avec :

```
""" test (un seul) pour la fonction randlist du module fonctions """
from fonctions import randlist
#import pdb       # à décommenter pour déboguer
#pdb.set_trace()  # à décommenter pour déboguer
n = 21
a = -6
b = 9
L = randlist(n, a, b)
if len(L) == n:
    print(f"La liste a le bon nombre d'éléments ({n})")
else:
    print("Erreur : la liste n'a le bon nombre d'éléments")

OK = True
for e in L:
    if not type(e) == int and (a <= e <= b):
        OK = False
if OK:
    print(f"Tous les éléments de la liste sont des entiers compris entre {a} et {b}")
else:
    print(f"Erreur : au moins un élément de la liste n'est pas un int ou n'est pas compris entre {a} et {b}")
```

N'écrivez pas (pour ne pas perdre trop de temps) un tel fichier, faites un test rapide directement dans la console ipython. On voit que ce test pourrait être écrit dans une fonction et être appelé plusieurs fois avec des instances différentes, ce qui permettrait de faire beaucoup plus de tests très facilement.

Exercice 3 *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. Écrire 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 :
$$m_k := \min_{k\in[\![0,k]\!]} u_k, \quad M_k := \min_{k\in[\![0,k]\!]} u_k$$
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). Écrire 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.

Aide :
— La parité d'un entier peut se tester avec l'expression booléenne `n % 2 == 0`.
— Les rappels sur les boucles **pour** sont donnés à partir de la p. 12.
— Les rappels sur les tests **if** sont donnés à partir de la p. 10.

Exercice 4 *Enlever les éléments redondants d'une liste*

Écrire une fonction d'entête `def unique(L):` qui construit une nouvelle liste avec les éléments non redondants de la liste `L`. Cette fonction retourne cette nouvelle liste et ne modifie pas la liste `L`. Aide :
— On utilisera l'algorithme évident [8] suivant :
1. on initalise la nouvelle liste (appelons là `LL`) par une liste vide `LL = []` ;
2. on parcourt tous les éléments de la liste d'entrée `L` pour chaque élément, on teste (voir ci-après l'opérateur `in`) s'il est déjà dans la nouvelle liste `LL`, si oui, on ne fait rien et sinon, on ajoute cet élément à `LL` avec la méthode `append` ;
3. il existe en python l'opérateur très pratique `in` [9] qui permet de savoir si un objet python est déjà dans une liste, un tuple, un tableau numpy, etc. en fait dans tout objet python qui est une séquence (un "iterable" en python) ; soit `e` un objet et `S` une séquence alors `e in S` est une expression booléenne qui renvoie vrai si l'objet `e` est dans la séquence `S` et faux sinon.

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

Exercice 5 *Calcul approché de e^x*

Écrire une fonction d'entête `def expo(x, tol=1e-8):` qui approche l'exponentielle avec l'algorithme suivant :

1. Si $x \geq 0$, on calcule 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 quasiment plus la somme. Pour cela si on appelle S_k la somme calculée avec les $k+1$ premiers termes, on arrêtera le calcul lorsque :

$$\frac{S_{k+1} - S_k}{S_k} \left(= \frac{t_{k+1}}{S_k} \right) \leq tol$$

2. Si $x < 0$, on utilisera l'identité $e^x = 1/e^{-x}$ avec un appel à cette même fonction.

Aide :
— Il faut utiliser une boucle tant que, voir p. 11. Vous êtes autorisé à écrire une boucle "infinie" "tant que Vrai" et faire le test de convergence à l'intérieur (et sortir de la boucle avec un `break` s'il est positif).
— 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`. L'idée est de calculer l'erreur relative entre ces deux fonctions et de l'afficher pour diverses valeurs de x.

8. On peut faire mieux en utilisant la méthode `sort`.
9. Á ne pas confondre avec le `in` d'une instruction `for`.

Exercice 6 *Découvrir les cycles d'une permutation*

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 :

$$\begin{array}{l} 0 \to 2 \to 6 (\to 0) \\ 1 (\to 1) \\ 3 \to 5 \to 4 (\to 3) \end{array}$$

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. 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"). Aide :

— Pour "marquer" les entiers de $[\![0, n-1]\!]$ on peut utiliser une liste de booléens initialisés à faux, ce qui peut s'obtenir de manière efficace avec :

```
marque = [False]*n  # créé une liste de taille n où chaque élément est le booléen Faux
```

Marquer un entier k s'écrira alors `marque[k] = True`.

— Pour cet exercice il faut utiliser des boucles tant que, voir p. 11.

— Et vous aurez besoin de `append` pour construire les listes utilisées.

— Tester votre fonction avec l'exemple donné dans l'énoncé ; vous pourrez aussi construire des permutations avec la fonction suivante :

```
import random
def rand_perm(n):
    """ Retourne une permutation aléatoire de [0,1, ..., n-1]"""
    p = list(range(n))
    random.shuffle(p)
    return p
```

Exercice 7 *Méthode de Newton robuste, 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. *Remarque :* on peut aussi se passer des clés pour les arguments optionnels mais il faut alors les donner dans l'ordre.

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 (de dérivabilité et généralement un point de départ x_0 suffisamment proche de r). On peut écrire cette itération sous la forme :

$$x_k = x_{k-1} + \tau_k d_k, \text{ avec } \tau_k = 1 \text{ et } d_k = -\frac{f(x_{k-1})}{f'(x_{k-1})}.$$

Augmentation du rayon de convergence : on peut remarquer que trouver les racines de f revient à minimiser $g := f^2$ et si on trouve un minimum x^* de g tel que $g(x^*) = 0$ alors x^* est bien une racine (un zéro) de f. Pour minimiser g, on peut utiliser une méthode du type :

$$x_k = x_{k-1} + \tau_k d_k$$

où d_k est une direction de descente de g en x_{k-1}, c'est à dire telle que $\langle d_k, g'(x_{k-1})\rangle < 0$ (cf cours d'optimisation de M1). Ici $g'(x_{k-1}) = 2f'(x_{k-1})f(x_{k-1})$. Avec $d_k = \frac{f(x_{k-1})}{f'(x_{k-1})}$, on a donc :

$$\langle d_k, g'(x_k)\rangle = -\frac{f(x_{k-1})}{f'(x_{k-1})} 2f'(x_{k-1})f(x_{k-1}) = -2f(x_{k-1})^2 < 0$$

si x_{k-1} n'est pas déjà une racine. Ainsi la méthode de Newton est une méthode de descente sur f^2 avec pas égal à 1. Si le pas est suffisamment petit on a alors nécesairement $g(x_k) < g(x_{k-1})$ ce qui est équivalent à $|f(x_k)| < |f(x_{k-1})|$. Ceci nous permet d'améliorer l'itération de Newton selon l'algorithme :

$\tau \leftarrow 1$ *(on commence par le pas de Newton)*
$d \leftarrow -f(x_{k-1})/f'(x_{k-1})$
Tant que "Vrai"
  $y \leftarrow x_{k-1} + \tau d$
  si $|f(y)| < |f(x_{k-1})|$
    fin de l'itération (sortie de la boucle tant que)
  sinon
    $\tau \leftarrow \tau/2$
$x_k \leftarrow y$

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

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

(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*. On prendra (à éventuellement modifier) $\epsilon_f = 10^{-8}$ et $\epsilon_x = 10^{-15}$.

L'entête de la fonction peut être :

```
def newton(x0, f, fp, epsf = 1e-8, epsx = 1e-15, itermax = 10):
```

et elle devra retourner les arguments suivants :

— le dernier itéré x_k obtenu ;
— une des deux chaîne de caractère, `'succes'` ou `'echec'` permettant de préciser si le test de convergence numérique est vrai ou faux ;
— $f(x_k)$
— le rapport $\frac{|x_k - x_{k-1}|}{\max\{|x_k|,1\}}$

Mini-test possible :

```
from fonctions import newton
from math import sin, cos
rep = newton(6,sin,cos)
print(rep)
```

Exercice 8 *Introduction aux nombres complexes : résolution de l'équation du second degré*

On aimerait que la fonction `resoud_quad` (cf le module `minibibquad`) 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`.