Introduction à la programmation python scientifique

lionel.bastard@grenoble-inp.fr
jean-christophe.toussaint@grenoble-inp.fr

Année 2022-2023

Ce document vient en support du cours de python. Il permet d'une part de conserver une trace écrite des notions vues en cours et d'autre part propose des exercices en surnombre de manière à ce que vous puissiez vous entraîner en dehors du cours sur les parties vous ayant posé problème. Par ailleurs, nous vous conseillons le site python-course.eu si vous souhaitez approfondir certains sujets ou même revoir des notions vues en cours avec un éclairage différent.

Le contenu du cours est précisé dans la table des matières ci-dessous :

Table des matières

1 Principaux concepts et instructions élémentaires

Dans ce chapitre, on se concentre sur les instructions de base du langage python, qui pourront être utilisée dans tout type de contexte, qu'il soit scientifique ou non. Le second chapitre se

concentrera sur la manipulation de données scientifiques.

1.1 Environnements de programmation

Python peut être utilisé de plusieurs manières. La plus simple consiste à taper les instructions ligne après ligne dans une console python. Chaque ligne est alors exécutée au moment ou l'utilisateur la valide avec la touche 'entrée'. Par exemple, si l'on tape le code :

```
2+2
```

Sur la ligne suivante s'affiche le résultat : 4. L'utilisation de la console est recommandée lorsque l'on a souhaite connaître le résultat de quelques instructions rapidement ou pour vérifier une syntaxe lorsque l'on développe un programme. Dès que l'on souhaite réaliser des programmes plus conséquents, il faudra utiliser une autre méthode.

La seconde façon d'utiliser python est d'écrire un programme, c'est à dire un ensemble de lignes qui seront toutes exécutées "d'un coup" lorsque l'on exécutera le programme. Pour ce faire, deux types d'environnements sont disponibles :

— l' "Integrated Development Environement (IDE)" est la solution classique. Il s'agit d'un éditeur de texte intégrant des commandes spécifiques pour l'exécution d'un programme et en général un outil de débuggage du code. Ces environnements sont particulièrement adaptés aux gros projets contenant plusieurs fichiers de code source. Les deux IDE que nous vous recommandons pour développer en python sont "spyder", fourni avec la distribution anaconda, et "Visual studio code", qu'il faut télécharger séparément.

— le "jupyter notebook "est une solution particulière à python. Il s'agit d'un environnement intégré dans un navigateur web, qui a l'avantage de permettre la visualisation interactive des données. Les "notebooks" sont constitués de différentes cellules qui peuvent être soit du code python soit du texte, ils sont donc intéressants pour produire un ensemble de petits programmes bien documentés et sont souvent utilisés pour l'enseignement du langage python. Note : on utilise "Ctrl+Entrée" pour exécuter le code de la cellule active dans un notebook.

Lorsque l'on passe par un programme, le résultat de chaque instruction n'est pas affiché. Si l'on souhaite connaître la valeur d'une variable, il faut utiliser la fonction `print()`. Par exemple : l'instruction suivante :

```
print(2+2)
```

déclenchera l'affichage du résultat suivant : 4. La fonction `print()` accepte plusieurs arguments [1]. Le résultat de chaque argument est affiché séparé par un espace (ce comportement par défaut peut être modifié). Par exemple :

1. Un argument est le terme que l'on fournit à la fonction, entre les parenthèses. En python, les différents arguments fournis à une fonction sont séparés par une virgule.

```
print('2+2=', 2+2)
```

Renvoie : 2+2=4. Pour finir, vous noterez que la fonction `print()` renvoie le résultat de l'argument (ou des arguments) de la fonction puis effectue un retour à la ligne. On peut s'en convaincre facilement en écrivant plusieurs instructions `print()` à la suite.

Pour l'instant nous avons vu comment afficher des résultats calculés par le programme, voyons à présent comment récupérer des informations d'un utilisateur. La fonction permettant ceci est `input()`. L'argument de cette fonction est le texte affiché pour que l'utilisateur sache ce qu'on lui demande. Par exemple, le programme suivant permet de dire bonjour de manière personnalisée :

```
nom = input('Quel est votre nom ?')
print('bonjour', nom)
```

Dans le programme ci-dessus, on a utilisé une variable : *nom*. Nous détaillerons l'utilisation de ces variables dans la section suivante.

Exercice 1 :

a) Utilisez la console python pour faire quelques calculs.

b) Créez un jupyter notebook, que vous utiliserez pour la suite de ce cours. Écrivez un programme, dans lequel vous mettrez en oeuvre la fonction `print()`. Essayez notamment avec plusieurs arguments et plusieurs instructions `print()` à la suite.

c) Testez la fonction `input()` dans un programme calculant la somme de deux nombres fournis par l'utilisateur.

1.2 Variables et objets

Il est probable que vous ayez eu un résultat inattendu lors de la question c) de l'exercice 1. Cela est probablement lié au type d'objet retourné par la fonction `input()`. Pour comprendre cela, il faut d'abord expliquer la notion de variable.

Une variable est un élément présent dans le programme auquel on associe un nom et une valeur. Le nom [2] est donné à la variable lors de sa déclaration. Par exemple, le code :

2. Attention : les noms de variable sont sensibles aux majuscules : *A* et *a* sont deux variables différentes. Par ailleurs, il est impossible de créer une variable commençant par un chiffre. Les caractères spéciaux ne sont pas autorisés dans les noms de variable, à l'exception notable du tiret bas '_'.

```
a = 4
```

permet de créer une variable nommée *a*, à laquelle on associe la valeur 4. A chaque variable est également associé un *type*, que python "devine" en fonction de la valeur associée à la variable lors de sa création. Les principaux types sont les suivants :

- `int` (entiers), `float` (nombres réels), `complex` (nombres complexes)
- `str` (chaines de caractères)
- `bool` (booléens, qui peuvent prendre seulement 2 valeurs : `True` ou `False`)
- beaucoup d'autres que nous verrons par la suite au fur et à mesure.

Pour comprendre ce qu'il se passe lors de l'exercice 1c), il faut préciser que la fonction `input()` retourne toujours un objet de type `str`. Si l'on veut faire la somme de deux nombres écrits par l'utilisateur, il faut donc au préalable transformer les chaînes de caractères obtenues en nombres. Pour cela, on utilise une fonction ayant le nom du type désiré. Par exemple :

```
a = input("entrez un nombre : ")
b = input("entrez un nombre : ")
print('la somme vaut', float(a) + float(b))
```

Les types numériques (`int`, `float` et `complex`) s'utilisent de manière assez intuitive. Voici quelques détails supplémentaires concernant les chaînes de caractères :

- On crée une chaîne de caractères avec les guillemets simples (= apostrophe) : 'abc' ou doubles "abc". Il est intéressant d'avoir ces deux options au cas ou l'on veuille utiliser l'un de ces guillemets dans une chaîne de caractères, par exemple "l'oiseau" ou 'je lui ai dit "bonjour"'.
- l'opérateur '+' fonctionne comme opérateur de concaténation sur les chaînes de caractères. Par exemple, 'a'+'bc' = 'abc'. Ainsi, '2'+'3'='23' alors que 2+3=5.
- il existe des caractères spéciaux intéressant à connaître : '\n' correspond au retour à la ligne, '\t' à une tabulation.

Il est parfois utile de connaître le type d'une variable. Cela peut se faire avec la fonction `type()`, dont voici un exemple d'utilisation :

```
a = input("entrez un nombre : ")
print(type(a))
```

Pour finir notre rapide tour d'horizon des variables et objets en python, il faut comprendre qu'une variable ne correspond jamais uniquement à la simple association "nom de variable" <=> "valeur". En effet, une variable contient des données supplémentaires, auxquelles on peut accéder grâce à l'opérateur ".". Ces données sont appelées "attributs" de l'objet. Par exemple, les objets de type `complex` possèdent un attribut nommée `real` et un autre : `imag` qui correspondent respectivement à la partie réelle et la partie imaginaire du nombre complexe considéré. Par exemple :

```
a = 2+3j  # crée un nombre complexe nommé a
partie_reelle = a.real  # contient la partie réelle de a
partie_imag = a.imag    # contient la partie imaginaire de a
```

En plus des attributs, un objet possède également des fonctions qui lui sont associées. Ces fonctions sont accessibles grâce à l'opérateur ".", tout comme les attributs. Il est possible de lister les attributs et fonctions associés à un type d'objet grâce à la commande `help(nom_de_l_objet)` ou `help(type_d_objet)`.Puisque nous seront régulièrement amenés à manipuler des chaînes de caractères, voici à titre d'exemple quelques fonctions associées à ce type d'objet :

```
c = "Salut ça va ?"  # crée une chaine de caractères nommée c
c1 = c.replace('a','i')  # remplace les 'a' de c par des 'i'
liste_de_mots = c.split() # renvoie une liste des mots contenus dans c
position = c.find('va') # renvoie la position de la première occurrence
C = c.upper() # renvoie la chaine de caractères en majuscules
```

Exercice 2 :

a) Reprenez l'exercice 1c. Regardez ce qu'il se passe lorsque l'utilisateur entre autre chose qu'un nombre.

b) Regardez comment python procède lors de la conversion d'une variable de type `float` en `int`. Testez également d'autres conversions de type même si elles ont l'air improbables (que se passe-t-il si l'on transforme un entier en booléen ?).

c) Créez différentes chaînes de caractères utilisant des caractères spéciaux et des guillemets.

d) Testez la fonction `type()` et montrez que le type d'une variable peut changer au cours de l'exécution d'un programme.

e) Regardez les attributs associés aux objets de type `complex` grâce à la fonction `help()`

1.3 Fonctions

Les fonctions sont des morceaux de programmes que l'on peut exécuter par la suite autant de fois que nécessaire. L'écriture de fonction permet de produire un programme plus concis donc plus lisible, ainsi que d'éviter des erreurs dans la recopie du code. En règle générale, dès que l'on souhaite utiliser un morceau de code plusieurs fois, même s'il s'agit d'une seule ligne, il est préférable de créer une fonction. La déclaration d'une fonction se fait à l'aide du mot-clé `def` selon la syntaxe suivante :

```
def nom_fonction(arg1, arg2):
    """ squelette d'une fonction prenant deux arguments """
    instruction 1
    instruction 2
    return valeur_retour
```

Après le nom de la fonction, les arguments sont listés entre parenthèse et séparés par une virgule. Ces arguments devront être présents lors de l'appel de cette fonction.

Notez que si la fonction n'a pas besoin d'arguments, on la déclare sans rien mettre entre les parenthèses : "`def nom_fonction():`". Dans tous les cas, le symbole ' :' termine obligatoirement la ligne. La ligne suivante est optionnelle mais fortement conseillée. Il s'agit d'une chaîne de caractère spéciale, nommée 'docstring'. Elle est délimitée par un triple guillemet, qui peut s'étaler sur plusieurs lignes et permet de décrire ce que fait la fonction. Il s'agit d'une bonne pratique qui favorise la lisibilité de votre code et qui permet à une personne n'ayant pas accès au code d'avoir des informations sur votre fonction via la commande `help(nom_fonction)`. A la fin de la fonction, la commande `return` indique quelle valeur est retournée par la fonction. Si l'instruction `return` n'est pas présente, la fonction renverra la valeur `None`, qui est un mot-clé du langage (notez le N majuscule). Pour finir, notez que les instructions à l'intérieur de la fonction sont décalées (généralement d'une tabulation ou 4 espaces). Ce décalage (ou indentation) n'est *pas* optionnel et doit être le même pour toutes les instructions, sinon votre programme ne pourra pas s'exécuter. Si vous revenez au même niveau d'indentation que le `def`, le code sera considéré comme ne faisant plus partie de la fonction.

Pour faire appel à la fonction, on indique son nom et les arguments que l'on souhaite passer à la fonction entre parenthèse (ou une parenthèse vide s'il n'y a pas d'argument). Si on est intéressé par la valeur de retour, on assigne une variable. Dans l'exemple ci-dessous, on appelle la fonction avec les arguments 2 et 3, la valeur de retour est stockée dans la variable f.

```
f = nom_fonction(2,3)
```

Lors de l'appel d'une fonction, il est possible d'utiliser le nom des arguments. L'utilisateur pourra ainsi utiliser la fonction en appelant les arguments dans l'ordre qu'il souhaite. Par exemple :

```
def division(numerateur, denominateur)
    return numerateur/denominateur

d1 = division(4,2)
d2 = division(2,4)
d3 = division(numerateur=4, denominateur=2)
d4 = division(denominateur=2, numerateur=4)
```

Dans cette exemple, on a donné des noms d'arguments explicites, de manière à ce que l'appel à la fonction soit plus facile à écrire et à lire. L'écriture correspondant à d3 et d4 est en effet

compréhensible même si l'on ne se souvient plus des détails de la fonction "division". Ici, d1, d3 et d4 donnent le même résultat (2), alors que d2 donne 0,5.

Une dernière possibilité existe pour personnaliser les arguments d'une fonction : on peut leur donner une valeur par défaut. Cela se fait lors de la déclaration de la fonction comme ceci :

```
def puissance(base, exposant=2)
    return base**exposant

p1 = puissance(4,2)
p2 = puissance(4)
p3 = puissance(base=4)
p4 = puissance(4,3)
```

S'il est très utile de créer des fonctions indépendantes, telles que celles que nous venons de présenter, il existe une autre manière d'utiliser les fonctions, en lien avec un objet. Nous avons déjà vu qu'il existe des attributs associés à un objet, il existe également des fonctions associées, qui sont appelées "méthode". Par exemple, les objets de type `complex` possèdent une méthode nommée `conjugate()` qui retourne le conjugué du nombre complexe considéré. Comme pour les attributs, on accède aux méthodes d'un objet via l'opérateur ".". Par exemple :

```
a = 2+3j  # crée un nomrbe complexe nommé a
partie_reelle = a.real    # utilisation de l'attribut 'real'
conj = a.conjugate()      # utilisation de la méthode 'conjugate'
```

Exercice 3 :

a) Reprenez le code de la fonction "puissance" ci-dessus, en ajoutant un docstring. Vérifiez le résultat retourné par les appels p1..p4. Vérifiez que le docstring joue bien son rôle en utilisant la fonction `help()`.

b) Utilisez la fonction `help()` pour obtenir des informations sur la méthode `replace()` associée aux chaînes de caractères (`str`).

c) Créez une fonction qui prend en argument une chaîne de caractères et retourne cette même chaîne de caractères dans laquelle on a supprimé tous les "e".

d) Modifiez votre fonction pour que l'utilisateur puisse spécifier la lettre qui est supprimée de la chaîne de caractères (par défaut c'est toujours le "e" qui est supprimé).

1.4 utilisation de modules - le module `math`

le langage python est contenu dans un ensemble de fichiers appelés "librairies" ou "modules". Un certain nombre de modules sont chargé automatiquement. Il est ainsi possible d'appeler la fonction `print()` directement car elle fait partie d'un module de base de python. Cependant, python peut être utilisé pour des applications très diverses (calcul scientifique, analyse de données, pilotage d'instruments de mesure et de caméras, création de sites web, pilotage de bases de données, programmation de micro-contrôleurs, ...). Pour ces applications spécifiques, l'utilisateur doit lui-même charger puis utiliser les modules adaptés. La recherche du ou des modules adaptés ainsi que la lecture de leur documentation est ainsi une partie importante du travail d'un développeur souhaitant créer une nouvelle application python.

Pour charger un module et pouvoir l'utiliser, il faut dans un premier temps l'installer dans votre distribution python. Dans le cadre de ce cours, python a été installé avec un grand nombre de modules dédiés au calcul scientifique, et vous n'aurez donc rien à installer [3]. Par contre, il faudra toujours charger en mémoire les modules dont vous avez besoin dans votre code. La commande pour ce faire est `import`. Par exemple, pour importer le module `math`, on écrit :

```
import math      # importe le module math
```

Lorsque l'import du module a été effectué, il est possible d'accéder aux fonctions contenues dans le module en utilisant l'opérateur ".". Pour utiliser la fonction `sin()`, qui renvoie le sinus de l'angle passé en arguement, il faut ainsi écrire :

```
import math          # importe le module math
a = math.sin(0.3)    # utilise la fonction sin() du module math
b = math.cos(0.3)     # utilise la fonction cos() du module math
```

Le fait d'utiliser `math.sin()` au lieu de `sin()` pour l'appel de la fonction peut sembler peu pratique, mais cela permet d'autoriser à ce que plusieurs fonctions dans différents modules aient le même nom. Si l'on sait que l'on n'utilisera que la fonction `sin()` du module `math` dans un programme donné, il est possible de modifier l'import comme suit :

```
from math import sin    # importe uniquement la fonction sin
a = sin(0.3)  # fonctionne
b = cos(0.3)  # ne fonctionne pas !!
c = math.cos(0.3)  # ne fonctionne pas non plus
```

Dans ce cas, l'appel à la fonction est plus simple, mais seule la fonction `sin()` est importée, les autres fonctions du module `math` ne seront donc pas accessibles.
Pour finir, notons qu'il est possible de renommer un module ou une fonction d'un module lors de

3. La méthode standard pour installer un module python est de se placer dans une console sur laquelle vous avez les droits administrateur puis de taper la commande `pip install nom_du_module`. Le module en question sera téléchargé depuis internet et installé dans votre distribution python.

l'appel à la fonction `export`. Cela est généralement utilisé pour avoir un nom plus court lors de l'appel du module ou de la fonction. Par exemple :

```
import math as ma       # import du module math sous le nom 'ma'
from math import factorial as fac  # import de math.factorial
a = ma.cos(0.2)
f = fac(6)
```

Exercice 4 :

a) importer le module `math`. Regarder les fonctions contenues dans ce module sur le site de référence https://docs.python.org/fr/3.5/library/math.html.

b) utiliser quelques fonctions du module `math` avec la première et la seconde méthode d'import présentées ci-dessus.

c) renommez une fonction du module math de votre choix lors de son import et utilisez la.

1.5 Listes, tuples et dictionnaires

Tous les langages de programmation récents fournissent des outils permettant de traiter des ensembles de données (des "structures de données"). En python, les principaux outils sont les listes, les tuples et les dictionnaires, qui font partie du python standard. Pour des applications scientifiques, nous aurons également besoin de tableaux numériques, qui sont fournis avec le module `numpy` et abordés au chapitre 2.1.

Les listes sont des ensembles de données hétérogènes (les objets contenus dans une liste peuvent avoir différents types). Le type de donnée associé aux listes est `list`. Pour créer une liste, on place les objets entre crochets [], chaque élément étant séparé par une virgule. Ainsi :

```
a = [1.3, "hello world", 3+2j]
```

crée une liste appelée 'a', qui contient 3 objets (un réel, une chaîne de caractère et un complexe). Pour accéder aux objets d'une liste, on utilise l'opérateur []. `a[1]` renverra ainsi la valeur de l'élément 1 de la liste. *Attention, la numérotation des objets commence à 0 en python.* Pour modifier la valeur de l'un des objets de la liste, on utilise également l'opérateur []. Les objets de type `list` possèdent beaucoup de fonctions permettant leur manipulation. Nous ne dresserons pas ici une liste exhaustive (voir la documentation pour cela), mais voici quelques exemples :

```
a = [1.3, "hello", 3+2j]  # création d'une liste
b = []  # création d'une liste vide
c = a[1]  # la variable c contient "hello"
a[2] = "coucou" # remplace le 3ème élement de a
a.append("bonjour") # ajoute "bonjour" à la fin de a
a.reverse() # inverse l'ordre des éléments de la liste
b.append(['a','b','c']) # ajoute une liste dans b
ab = a+b  # concaténation des listes a et b
```

Il est possible d'utiliser un entier négatif avec l'opérateur [], qui permet d'accéder à l'élément en partant de la fin de la liste. Par exemple `a[-1]` correspond au dernier élément de la liste `a`. Pour finir, on peut noter que les chaînes de caractères sont considérées comme des listes de caractères. Il est donc possible avec l'opérateur [] d'extraire un caractère d'un objet de type `str`.

Une structure de données très semblable à aux listes est le type `range`. Il s'agit d'une liste de nombres espacés d'une valeur constante. La spécificité du `range` est qu'il ne stocke pas tous les éléments en mémoire et permet donc de générer de très longues suites de chiffres sans encombrer la mémoire lors de l'exécution du programme. La création d'un `range` est assurée par la fonction `range()`, qui prend trois arguments, dont les deux derniers sont optionnels. Le premier argument correspond au point de départ (inclus), le second au point d'arrivée (exclus) et le troisième au pas (un pas négatif fait commencer le `range` par son point d'arrivée). Si on omet un argument, le pas vaut 1 par défaut et si on omet deux arguments, le point de départ vaut 0 et le pas vaut 1.

```
r1 = range(10)          # de 0 à 9 par pas de 1
r2 = range(2, 8)        # de 2 à 7 par pas de 1
r3 = range(2, 8, 2)     # de 2 à 7 par pas de 2
```

Une seconde structure de donnée proposée par python est le `tuple`. Cette structure est proche d'une liste : c'est également une collection hétérogène d'éléments. La principale différence réside dans le fait que les éléments d'un tuple ne peuvent pas être modifiés. Il faut donc nécessairement initialiser un tuple lors de sa création, car il n'est pas possible si l'on crée un tuple vide d'y ajouter des éléments. Un tuple est créé par l'opérateur parenthèses (), mais on accède à ses éléments avec l'opérateur [], comme pour les listes.

```
a = (1.3, "hello", 3+2j)
c = a[1]  # la variable c contient "hello"
a[2] = "coucou" # renvoie une erreur : impossible de modifier un tuple
```

Si le type `tuple` est moins flexible que le type `list`, les tuples sont néanmoins utilisés dans un certain nombre de situation où l'on exploite le fait que leurs éléments ne peuvent pas être modifiés et que leur nombre d'éléments est fixe. Le passage de variables séparées à `tuple` est par exemple très simple :

```
t1 = ('a', 'b', 'c')    # création d'un tuple contenant 3 éléments
i,j,k = t1  # création de 3 variables i,j,k valant t1[0], t1[1] et t1[2]
m,n,p = (1, 1.5, 4) # assignation directe, évite d'écrire 3 lignes
t2 = (i,j,p) # utilisation de variables pour créer un tuple
```

Les tuples sont également utilisés lorsque l'on souhaite qu'une fonction retourne plusieurs valeurs. Dans ce cas, la fonction retourne alors un tuple, même si l'on omet les parenthèse dans la clause `return`. Pour illustrer cela, le code ci-dessous décrit une fonction qui renvoie le premier et le dernier caractère d'une chaîne de caractères passée en argument.

```
def debut_fin(chaine):
    debut = chaine[0]
    fin = chaine[-1]
    return debut, fin   # équivalent à return (debut, fin)

a = debut_fin('abcdefgh')  # a est un tuple qui vaut ('a', 'h')
deb,fin = debut_fin('abcdefgh') # deb='a' et fin = 'h'
```

Dans la dernière ligne de l'exemple ci-dessus, on a utilisé le fait qu'une fonction qui renvoie plusieurs éléments produit un tuple, combiné avec l'assignation de plusieurs variables aux éléments d'un tuple.

Pour finir, une dernière structure de donnée importante en python est le dictionnaire, de type `dict`. Il s'agit d'une collection d'éléments contenant deux parties : une clé et une valeur. La clé et la valeur peuvent être de n'importe quel type. Les dictionnaires sont créés avec les accolades , la clé et la valeur étant séparés par le symbole " :". Comme les listes, les éléments d'un dictionnaires peuvent être modifiés et des éléments peuvent être ajoutés ou retirés après la création du dictionnaire. Contrairement à la liste, les éléments ne sont pas ordonnés : on ne peut pas y accéder via leur numéro, mais à l'aide de leur clé. Voici un exemple d'utilisation :

```
d = {"nom":"John", "age":23, "profession":"étudiant"}
print(d["nom"])   # renvoie la valeur de l'élément "nom"
print(d[0])     # ne fonctionne pas : éléments non ordonnés
d["age"] = 25  # modifie la valeur de l'élément "age"
d["taille"] = 1.85 # ajoute une élément au dictionnaire d
print(d)   # affiche tous les éléments du dictionnaire
"age" in d  # True
```

Dans un dictionnaire, la clé doit être unique. Si vous essayez de créer un dictionnaire avec deux clés identiques, une erreur est générée. Il est possible de savoir si une clé fait partie d'un dictionnaire grâce au mot-clé `in`. Ainsi, la dernière ligne du code ci-dessus renverra "True", car la clé "age" fait partie du dictionnaire d.

Pour finir, sachez qu'il est possible de transformer une structure de donnée en une autre grâce aux fonctions `list()`, `tuple()` et `dict()`. Par exemple :

```
a = [1.3, "hello", 3+2j]  # création d'une liste
t = tuple(a)  # t est un tuple avec les mêmes éléments que a
```

Exercice 5 :

a) Écrivez une fonction qui crée un liste à partir des 3 arguments fournis. La fonction devra retourner la liste ainsi créée.

b) Utilisez l'opérateur [] pour lire et modifier les éléments d'une chaîne de caractères.

c) testez la fonction range avec des valeurs de pas négatives. Déterminez si le point de départ et le point d'arrivée sont inclus ou exclus dans ce cas.

d) Créez une fonction 'som_dif', qui renvoie la somme et la différence des deux premiers nombres d'une liste fournie en arguments. Testez le bon fonctionnement de votre fonction. Si l'argument est un tuple, votre fonction est-elle toujours opérationnelle ? Expliquer le résultat donné par `som_dif(som_dif([1,3]))`.

e) Vous voulez créer une application d'annuaire téléphonique. Vous décidez de stocker les données de votre annuaire dans une liste de dictionnaires, chaque dictionnaire de la liste contenant les informations d'un contact donné. Créez une liste vide (l'annuaire), puis une fonction pour ajouter un contact à l'annuaire.

1.6 Boucles

L'écriture de programmes implique souvent d'effectuer une même opération plusieurs fois, en faisant par exemple varier un paramètre entre chaque itération. Pour simplifier l'écriture et la lisibilité du code, il est fortement recommandé d'utiliser les boucles plutôt que d'écrire 10 fois la même ligne. Nous étudierons ici les boucles `for`, qui sont les plus fréquemment utilisées, ainsi que les boucles `while`. L'utilisation des boucles `for` se base sur les mots-clé `for` et `in`, l'instruction se terminant par le symbole " :". Lors de l'exécution, les valeurs de l'objet itérable (une liste, un tuple, un range, un dictionnaire, ...) mentionné après le mot-clé `in` sont utilisées une à une. Les instructions de la boucle doivent être indentées, à l'image des instructions au sein d'une fonction. Voici un exemple de boucle utilisant une liste, puis une autre utilisant un range :

```
# boucle sur une liste
panier = ["pommes", "radis", "céréales", "tablettes de chocolat"]
for element in panier:
    print("allons acheter des", element)
```

```
# boucle sur un range
for n in range(5):
    print(n)
    print(n**2)
print(n, " boucles effectuées") # instruction hors boucle
```

Dans le second exemple, on effectue une boucle sur un range créé au moment de déclarer la boucle. n est le nom que l'on a choisi pour l'élément courant. Notez que la boucle contient deux instructions (visibles par leur indentation). Dès que l'on supprime l'indentation, cela signifie que l'on sort de la boucle. L'élément courant de la boucle existe toujours lorsque la boucle est terminée, il conserve la dernière valeur utilisée.

Il est parfois utile lorsque l'on itère sur les éléments d'une liste d'avoir également accès au numéro de l'itération en cours. Pour cela, il faut utiliser la fonction `enumerate()`. Entre les instruction `for` et `in` se trouveront alors deux noms de variable : le premier correspondant au numéro de l'itération et le second au nom de l'élément courant dans la liste. Voici un exemple :

```
fruits = ["pomme", "poire", "abricot"] # création d'une liste
for i,fruit in enumerate(fruits):
    print("le fruit numéro ", i, " est : ", fruit)
```

Le second type de boucle disponible en python utilise l'instruction `while` ('tant que ...'). Cette instruction est utlisée lorsque l'on ne connaît pas à priori le nombre d'itérations qui sera requis. La syntaxe est la suivante : "`while` condition vraie :" suivi des instructions à exécuter tant que la condition donnée en préambule reste vraie. Dès que la condition devient fausse, la boucle est interrompue et la suite du code est exécutée.

Par exemple, si l'on se rappelle que la fonction exponentielle peut s'écrire comme une somme infinie, on peut écrire :

$$\exp(1) = \sum_{n=0}^{\infty} \frac{1}{n!}$$

Pour avoir une valeur approchée de exp(1), on peut interrompre la somme au bout de N termes :

$$\exp(1) \approx \sum_{n=0}^{N} \frac{1}{n!}$$

Le code ci-dessous permet de calculer le nombre N de termes nécessaire pour converger vers le nombre exp(1) à 10^{-6} près :

```
import math
somme = 0
n = 0
while abs(math.exp(1)-somme) > 1e-6:
   somme = somme + 1/math.factorial(n)
```

```
    n = n+1
print(n)
```

Notez l'utilisation de la valeur absolue dans la condition de boucle. Cette valeur absolue permet d'être certain d'avoir une valeur positive à comparer avec la valeur limite (ici 1e-6). Même si elle n'est pas nécessaire ici, c'est une bonne habitude d'utiliser systématiquement la valeur absolue dans ce type de condition de boucle, de manière à éviter les boucles infinies.

J'aimerais pour finir sur les boucles aborder un sujet connexe, nommé "list comprehension". Il s'agit d'une notation rapide pour créer des listes ou des tuples à l'aide d'une boucle. Cette notation peut être difficile à appréhender au premier abord, mais une fois habitué on ne peut plus s'en passer ! La syntaxe consiste à écrire une expression entre crochets pour créer une liste (ou entre parenthèses pour créer un tuple). Cette expression contient la valeur d'un élément de la liste créée suivi de l'instruction de boucle vue précédemment ("for x in liste"). Voici quelques exemples

```
fruits = ["pomme", "poire", "abricot"] # création d'une liste
# crée une nouvelle liste contenant la première lettre de chaque fruit
premieres_lettres = [e[0] for e in fruits]
# crée une nouvelle liste contenant le carré des premiers entiers
carres = [x**2 for x in range(5)]
```

Exercice 6 :

a) La fonction exponentielle peut être définie par une somme infinie :

$$exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Écrivez une fonction permettant de calculer une valeur approchée de exp(x,N) en utilisant la somme sur les N premiers éléments.

b) En utilisant une boucle while, regardez combien d'itérations sont nécessaires pour calculer la valeur correcte à 10^{-9} près de exp(x) pour x=0, 1 et 2.

1.7 Instructions conditionnelles

Après les boucles, un second type d'instruction est très utilisé en programmation : les instructions conditionnelles. En python, on utilise le mot-clé `if` pour déterminer quelles instructions effectuer en fonction de la validité d'une condition. Comme pour les fonctions et les boucles,

l'indentation du code permet de délimiter les instructions faisant partie de la condition. Voici un exemple :

```
age = input("entrez votre âge : ")
age = int(age)
if age > 0 :    # vérification que 'age' est positif
    print("Vous êtes né(e) en ", 2021-age)
print("A bientôt !")
```

Dans cet exemple simple, on ne traite pas le cas où l'utilisateur entre un nombre négatif. pour traiter le cas où la condition n'est pas réalisée, on utilise `else`, qui donne lieu à un nouvel ensemble d'instructions (marqué comme d'habitude par une indentation :

```
not_ok = True
while not_ok :
    age = int(input("entrez votre âge : "))
    if age > 0 :    # vérification que 'age' est positif
        print("Vous êtes né(e) en ", 2021-age)
        not_ok = False
    else :
        print("entrez un entier positif !")
print("A bientôt !")
```

Pour finir, il peut être intéressant d'enchaîner différentes conditions, grâce à la commande `elsif` :

```
not_ok = True
while not_ok :
    age = int(input("entrez votre âge : "))
    if age >= 18 :
        print("Vous êtes une personne majeur(e) né(e) en ", 2020-age)
        not_ok = False
    elif age >0 :
        print("Vous êtes un(e) mineur(e) né(e) en ", 2020-age)
        not_ok = False
    else :
        print("entrez un entier positif !")
print("A bientôt !")
```

Le fonctionnement est le suivant : la première condition (`if...`) est testée. Si elle est fausse, le programme passe à l'instruction suivante (`elif...`), etc... Dès qu'une condition est valide, le programme sort de la séquence `if...elif...else` entière. Pour une séquence donnée, il y a un unique `if` et un unique `else`, par contre, il est possible d'enchaîner autant d'instructions `elif` que nécessaire.

Dans les exemples ci-dessus, nous avons créé des conditions qui sont vraies ou fausses à l'aide d'opérateurs de comparaison (>= pour supérieur ou égal et > pour supérieur strict). Il est également possible de comparer si deux variables sont égales avec l'opérateur ==. Attention à bien utiliser cet opérateur pour les comparaisons, qui est différent de l'opérateur = (assignation d'une variable). Il existe également un opérateur "est différent de" : !=. Le tableau ci-dessous récapitule les différents opérateurs de comparaison.

symbole	signification	symbole	signification
>	supérieur strict	>=	supérieur ou égal
<	inférieur strict	<=	inférieur ou égal
==	égal à	!=	différent de

TABLE 1 – opérateurs de comparaison

Tous ces opérateurs ont comme résultat un booléen (objet de type `bool`), qui a deux valeur possibles : `True` ou `False` (attention aux majuscules !). Il est donc possible d'assigner le résultat d'une comparaison à une variable comme dans le code suivant :

```
nombre = int(input("entrez un nombre entier"))
resultat = (nombre == 42)
```

Ici, resultat vaudra `True` si l'utilisateur entre le nombre 42 et `False` sinon. Cette assignation de variable peut remplacer de manière compacte une instruction `if... else`. Notez que les parenthèses autour de "`nombre == 42`" ne sont pas nécessaires, mais elles rendent le code plus lisible.

Exercice 7 :

a) Recopiez le code correspondant au dernier exemple du programme calculant l'année de naissance d'une personne. Exécutez-le et vérifiez ce qu'il se passe pour différentes entrées de l'utilisateur. Ajoutez une condition supplémentaire (par exemple "bébé" ou "vieux", ...).

b) Créez une fonction 'minmax', qui retourne le minimum et le maximum des nombres présents dans une liste passée en argument. Utilisez votre fonction et testez son bon fonctionnement. La fonction fonctionne-t-elle toujours correctement si vous l'utilisez avec un range au lieu d'une liste ?

1.8 Les exceptions (bases)

Il s'agit d'un chapitre que nous allons traiter de manière très rapide. L'objectif est d'une part de comprendre ce qu'est une exception et d'autre part de connaître les fonctions minimales per-

mettant de les gérer.

Les exceptions correspondent à la manière de gérer les erreurs en python. Par exemple, dans le code demandant l'âge d'une personne du chapitre précédent, il est possible que l'utilisateur fournisse une entrée qui ne peut pas être transformée en entier (par exemple "a"). La fonction `int()` renvoie dans ce cas une exception : Un message d'erreur est affiché et le programme est interrompu. Dans le cas où l'utilisateur tape 'a', le programme affiche ainsi :
ValueError : invalid literal for int() with base 10 : 'a'

Il est possible "d'attraper" l'exception au moment où elle est déclenchée par la fonction `int()` et de modifier son comportement par défaut (envoi du message d'erreur et interruption du programme). Ceci se réalise à l'aide du mot-clé `try:`, pour spécifier le bloc dans lequel on souhaite récupérer les interruptions, puis de la commande `catch:` pour décrire le comportement attendu lorsqu'une exception apparaît. Notez que si le bloc `try` ne soulève pas d'exception, le bloc `catch` n'est pas exécuté et tout se déroule normalement. Voici ce que l'on pourrait faire pour éviter d'interrompre le programme lorsque l'utilisateur fait une entrée incorrecte :

```
not_ok = True
while not_ok :
    try:
        age = int(input("entrez votre âge : "))
    except:
        age = -1
    if age >= 18 :
        print("Vous êtes une personne majeur(e) né(e) en ", 2021-age)
        not_ok = False
    elif age >0 :
        print("Vous êtes un(e) mineur(e) né(e) en ", 2021-age)
        not_ok = False
    else :
        print("entrez un entier positif !")
print("A bientôt !")
```

Dans cet exemple, lorsqu'une exception est levée car l'utilisateur n'a pas entré un entier, on récupère cette erreur et on fixe la variable 'age' à -1. Tout se passe donc comme si l'utilisateur avait tapé -1 et le programme peut continuer normalement.

1.9 Lecture et écriture de fichiers

Pour finir cette première partie de cours sur les bases de python, voyons comment lire et écrire les données contenues dans un fichier. Dans cette partie, nous présentons la méthode générale permettant de lire ou d'écrire n'importe quel type de fichier. Nous verrons au chapitre suivant comment ouvrir et mettre en forme des fichiers contenant des données numériques en utilisant la

libraire "pandas".

La méthode conseillée pour ouvrir un fichier avec python consiste à utiliser un bloc `with`. Les instructions à l'intérieur de ce bloc peuvent accéder aux données présentes dans le fichier (ou en ajouter), mais dès que le bloc prend fin, le fichier est refermé (et il sera alors accessible par une autre application). La fonction permettant d'ouvrir un fichier est `open()`, qui prend comme argument le nom du fichier (ou le chemin complet) suivi d'un argument permettant de préciser si l'on souhaite ouvrir le fichier en lecture ou en écriture :

- 'r' pour "read" : ouverture en lecture.
- 'w' pour "write" : ouverture en écriture, écrase les données éventuellement présentes dans le fichier.
- 'a' pour "append" : ouverture en écriture, ajout à la fin du fichier.

La fonction `open()` retourne un objet permettant de manipuler le fichier - cet objet est nommé le 'handle' du fichier.

Une fois le fichier ouvert, on souhaite récupérer son contenu ou écrire quelque chose dedans. Ces actions sont menées à bien à l'aide de méthodes (fonctions) du 'handle' du fichier ouvert :

- `readlines()` permet de lire l'ensemble du fichier. Cette fonction retourne une liste de chaînes de caractères, chaque élément de la liste correspondant à une ligne du fichier.
- On peut aussi utiliser directement le handle du fichier au sein d'une boucle `for` pour accéder à chaque ligne du fichier et éventuellement traiter ces données au fur et à mesure qu'elles sont lues.
- `write()` permet d'écrire une chaîne de caractères dans le fichier.

Voici un exemple montrant l'écriture de données dans un fichier :

```
donnees = ["exemple de fichier écrit avec python", 1.5, 2, 4, 5.9, 10]
with open("exemple.txt", 'w') as f: # f est le "handle"
    for d in donnees:
        f.write(str(d))
        f.write("\n")
```

Notez que la fonction `write()` n'ajoute pas de retour à la ligne après avoir écrit la chaîne de caractères. C'est pour cela que le code proposé ajoute le caractère '\n' après chaque élément de la liste. Voici à présent un exemple montrant la lecture des données contenues dans un fichier :

```
with open("exemple.txt", 'r') as f:
    lignes = f.readlines()
print(lignes)
```

Une autre façon d'obtenir le même résultat, qui montre que l'on peut directement utiliser le handle du fichier (f) au sein d'une boucle `for` :

```
lignes = []
with open("exemple.txt", 'r') as f:
    for ligne in f:
        lignes.append(ligne)
print(lignes)
```

Si on ouvre un fichier en lecture qui n'existe pas ou un fichier en écriture à un endroit où l'on n'a pas la permission, une exception est levée et le programme s'interrompt. On associe donc souvent un bloc `try` pour récupérer les exceptions et continuer le programme lorsque c'est possible.

Pour finir cette section sur les fichier, il peut être intéressant de formater les données numériques que l'on écrit (fixer le nombre de chiffres après la virgule ou utiliser la notation scientifique par exemple). Il existe de nombreuses méthodes de formatage des nombres en python, nous présentons ici notre préférée (qui n'est valable que depuis python 3.6). Tout d'abord, voici une syntaxe pour créer une chaîne de caractères faisant intervenir des noms de variable : la chaîne de caractères est précédée de 'f', et les variables sont mises entre accolades. Les variables seront remplacées par leur valeur dans la chaînes créée par cette méthode :

```
a = 2.44
texte = f"la variable a vaut {a} !"
print(texte)
```

Pour mettre en forme la manière dont la valeur de la variable numérique est affichée, on peut ajouter à l'intérieur des accolades, après le nom de variable, un code de formatage tel que ' :.2f'. Le code de formatage contient le nombre de décimales et une lettre correspondant au type d'affichage ('f' pour l'affichage d'un réel avec virgule fixe ou 'e' pour l'affichage scientifique par exemple)[4]. Si l'on souhaite forcer 5 chiffres après la virgule pour l'affichage de 'a' dans l'exemple précédent, le code sera :

```
a = 2.44
texte = f"la variable a vaut {a:.5f} !"
print(texte)
```

Pour l'écriture des fichier contenant des colonnes de nombres, l'intérêt du formatage en fixant le nombre de chiffres après la virgule est d'obtenir des colonnes bien alignées.

Exercice 8 :

a) Créer un fichier de données numériques formatées, précédées d'une en-tête (par exemple pour décrire les données). Vous pouvez utiliser des données numériques quelconques que vous générerez, elles seront placées sur 2 colonnes.

4. D'autres options de formatage existent, voir https://docs.python.org/3/library/string.html.

b) Ajoutez au fichier précédemment créé dix lignes correspondant aux 10 premiers entiers pour la première colonne et à leur carré pour la seconde colonne.

c) Récupérer les données du fichier, les transformer en `float` puis les stocker dans une liste

2 Python scientifique et analyse de données

2.1 Tableaux numériques avec numpy

Le module le plus important en python scientifique est nommé `numpy`. Ce module offre un nouveau type d'objet correspondant à des tableaux de nombres à n dimensions, ainsi que de très nombreuses fonctionnalités permettant de travailler avec ces tableaux. Pour utiliser le module `numpy`, il faut tout d'abord l'importer :

```
import numpy as np
```

la fin de l'appel (`as np`) n'est pas obligatoire, mais est très souvent utilisée. Les tableaux numériques proposés par numpy sont nommés `ndarray`. Pour créer un tel objet, plusieurs fonctions peuvent être utilisées :

```
a = np.asarray(liste_de_nombres)
b = np.arange(nb_elements)
c = np.zeros(dimensions)
d = np.linspace(debut, fin, nb)
```

— La fonction `asarray()` initialise le tableau avec une liste de nombres, qui peut être une liste de listes dans le cas de tableaux à plusieurs dimensions.

— La fonction `arange()` est l'équivalent de la fonction `range()` et crée un tableau à une dimension contenant une suite de nombres. Les arguments sont identiques à ceux de `range()` (voir section 1.5).

— La fonction `zeros()` crée un `ndarray` dont tous les éléments valent 0. Les dimensions du tableau sont précisées sous forme d'un tuple : par exemple `y=np.zeros((10,5))` crée un `ndarray` de 10x5 éléments (10 lignes et 5 colonnes).

— La fonction `linspace()` crée un tableau à une dimension avec nb éléments, équi-répartis entre debut et fin. Cette fonction peut être plus pratique à utiliser que `arange()` car on définit le nombre d'éléments directement, plutôt que l'écart entre deux éléments.

Les tableaux numpy imposent un type commun à tous leurs éléments (int, float, complex,...). Ce type peut être précisé lors de la création du `ndarray` par l'utilisation de l'argument nommé `dtype`. Par exemple :

```
b = np.arange(10, dtype='float')
```

On peut accéder aux éléments d'un `ndarray` à l'aide de l'opérateur [], comme pour les listes. Cependant, les `ndarray` pouvant avoir plusieurs dimensions, il faut préciser l'indice dans chaque direction. par exemple, pour un tableau en deux dimensions, on pourrait avoir le code suivant :

```
y = np.zeros((10,5))     # tableau 2d 10x5 éléments
a = y[3,3]               # lecture valeur élément (3,3)
y[2,4] = 1.5             # modification valeur élément (2,4)
```

Attention, comme pour les listes, la numérotation commence à 0 : `y[0,0]` sera donc l'élément à l'intersection de la première ligne et la première colonne et `y[2,4]` sera situé en 3ème ligne, 5ème colonne. Comme pour les listes, il est possible de spécifier "facilement" un sous-ensemble du tableau (technique nommée "slicing"). La syntaxe utilise le symbole ' :' et est similaire à celle des listes. Cependant, le fait qu'il y a potentiellement plusieurs dimensions complique l'écriture... A titre d'exemple, voici l'utilisation du slicing sur un tableau bidimensionnel. Le résultat retourné par l'opération est toujours un objet de type `ndaray`.

```
y = np.zeros((10,5))     # tableau 2d 10x5 éléments
z1 = y[2:4,0]            # lignes 2 et 3, colonne 0
z2 = y[2:8:2,2:4]        # lignes 2, 4 et 6, colonnes 2 et 3
z3 =  y[:,2:4]           # toutes les lignes, colonnes 2 et 3
z3[:,:] = 2              # modification de toutes les valeurs de z3
z4 = y[:4,3:]            # lignes min=0 à 3, colonnes 3 à max=4
```

Comme précédemment, la numérotation commence à 0, donc dans l'exemple de z1, ligne 2 et 3 correspond aux 3ème et 4ème lignes. Le slicing permet de spécifier la ligne ou colonne de début (inclue), de fin (exclue) et éventuellement le pas. Si l'on souhaite modifier l'ensemble des lignes ou des colonnes, il faut indiquer ' :' à l'emplacement correspondant (voir exemples z3). Plus généralement, si un nombre est omis, sa valeur par défaut est utilisée (le premier élément pour start, le dernier pour stop, et la valeur 1 pour le pas), voir l'exemple z4. Pour finir, notons que tous les sous-tableaux créés sont ce que l'on nomme des 'vues' du tableau principal (par exemple z1 est une vue de y). Cela signifie concrètement qu'il s'agit du même objet. Par exemple, la modification des éléments de z3 dans le code précédent modifie de fait le tableau y duquel z3 est une vue.

L'intérêt des tableau `numpy` par rapport aux listes réside entre-autre dans le fait qu'il est possible de faire de l'arithmétique sur des tableaux. Il est notamment possible de :

— Faire des opérations (addition, multiplication, puissance, ...) sur des tableaux. Ces opérations sont réalisées élément par élément (pas de multiplication matricielle !) et nécessitent donc des tableaux de mêmes dimensions. Si les tableaux n'ont pas la même dimension, `numpy` essaie quand même de réaliser l'opération en agrandissant le tableau trop petit dans une ou plusieurs directions. Ceci est nommé 'broadcasting' et sera explicité plus loin.

— Appliquer des fonctions mathématiques (sin, exp, ...) aux tableaux. Là encore, la fonction est appliquée élément par élément à tous les éléments du tableau.

Voici quelques exemples de possibilités offertes par `numpy` :

```
import numpy as np
a = np.ones((4,6))  # tableau 4 lignes 6 colonnes rempli de 1
b = np.zeros((4,6)) # tableau 4 lignes 6 colonnes rempli de 0
c1 = a+b            # tableau 4x6
c2 = a*4+2          # tableau 4x6
c3 = np.sin(a)      # tableau 4x6
```

On voit ici avec c2 un premier exemple de broacasting : on demande la multiplication de a (dimension 4x6) avec 4 (dimension 1x1). Le nombre 4 est alors étendu en un tableau de 4x6 contenant uniquement des 4, et la multiplication peut être faite. Il en va de même avec l'addition qui suit. Voici d'autres exempls de broadcasting :

```
import numpy as np
a = np.full((4,6),7)# tableau 4 lignes 6 colonnes rempli de 7
b = np.arange(6)    # tableau 1 ligne 6 colonnes
x1 = a*b            # tableau 4x6
c = np.asarray([1],[2],[3],[4])  # tableau 4 lignes 1 colonne
x2 = a*c            # tableau 4x6
x3 = b*c            # tableau 4x6
```

Les opérations permettant de calculer x1 et x2 dans l'exemple précédent fonctionnent grâce au broadcasting. Dans le premier cas, le tableau b est étendu à 4 lignes (identiques) pour pouvoir être multiplié par a. Dans le second cas, c est étendu à 6 colonnes (identiques). Notez que si l'on avait choisi une taille différente de 6 éléments pour b et 4 pour c, l'opération aurait échoué : le broadcasting ne peut fonctionner que si les dimensions des deux tableaux peuvent être rendues compatibles. Pour finir, l'exemple de x3 montre que le broadcasting fonctionne même si les deux tableaux ont besoin d'être étendus (on ajoute des lignes à b et des colonnes à c).

L'utilisation de fonctions mathématiques telles que `sin` ou `exp` sur des tableaux est pratique car elle évite de devoir faire une boucle sur tous les éléments du tableau et d'appliquer la fonction sur chaque élément. Par ailleurs, ces fonctions ont été optimisées et leur temps d'exécution est nettement inférieur à celui obtenu en passant par une boucle `for`. Il est donc fortement conseillé d'utiliser ces fonctions, quitte à créer un tableau `numpy` spécialement pour cela.

Outre la possibilité de créer des sous-tableaux et l'arithmétique sur les tableaux, de nombreuses fonctionnalités existent pour manipuler les `ndarrays`. Ces fonctionnalités sont présentes soit sous formes de fonctions générales au module `numpy`, soit comme méthodes associées à l'objet `ndarray`. Dans de nombreux cas, les deux approches sont disponibles, même si certains détails peuvent changer d'une implémentation à l'autre. Voici un exemple autour de la fonctionnalité consistant à "aplatir" un tableau à n dimension en une seule dimension :

```
import numpy as np
a = np.ones((5,6))      # tableau 5 lignes 6 colonnes rempli de 1
b = np.ravel(a)         # fonction générale de numpy
b = a.flatten()         # méthode de l'objet ndarray
```

Dans cet exemple, les fonctions `ravel` et `flatten` ont le même effet d'aplatir le tableau, mais la première est une fonction générale alors que la seconde est une méthode. Si l'on regarde en détail la documentation, on voit de plus de légères différences entre les fonctions, par exemple `ravel` retourne une vue de a alors que `flatten()` en produit une copie. Voici quelques exemples de fonctions utiles :

— `a.T` où a est un tableau permet de transposer ce tableau (inverser lignes / colonnes)

— `np.hstack()` et `np.vstack()` prennent en argument un tuple contenant les tableaux à assembler. `hstack` assemble les tableaux horizontalement et `vstack` verticalement.

— `a.reshape()` où a est un tableau permet de changer la forme du tableau (nombre de dimensions et nombre d'éléments par dimensions). L'argument attendu par cette fonction est un tuple contenant la nouvelle forme - par exemple (3,5) pour 3 lignes et 5 colonnes. Le nombre total d'éléments du tableau doit rester inchangé.

— `a.sum()` où a est un tableau retourne la somme de tous les éléments du tableau. Il existe également une fonction `average()` pour la valeur moyenne et `std()` pour la déviation standard.

Voici quelques exemples d'utilisation de ces fonctions :

```
import numpy as np
a = np.arange(12).reshape(3,4)
b = np.arange(20).reshape(5,4)
c = np.vstack((a,b))
d = np.hstack((a.T,b.T))
print(c.sum())
```

Exercice 9 :

a) Créer un tableau numpy avec 100 éléments répartis entre 0 et 2π, puis un autre tableau contenant le sinus de chacun des éléments du premier tableau.

b) Enregistrez ces deux tableau sous forme de deux colonnes dans un fichier

c) Créez un tableau à deux dimensions de taille (100x2) contenant les données des deux tableaux précédents. Créez un sous-tableau contenant uniquement les 20 premières lignes.

d) Utilisez la fonction `reshape()` sur le tableau créé précédemment et observez le résultat.

e) Calculez la somme des 100 premiers entiers sans utiliser de boucle.

2.2 Visualisation des données avec matplotlib

L'un des modules permettant la visualisation de données scientifiques le plus utilisé est matplotlib. Nous allons étudier quelques fonctions de ce module, mais sachez qu'il est très complet et permet une grande personnalisation des graphiques produits. Malheureusement, il est également relativement complexe et certaines fonctions sont mal documentées. Il faut souvent s'appuyer sur

des exemples trouvés sur le web pour obtenir le rendu souhaité. L'appel du module est généralement réalisé par la ligne suivante :

```
from matplotlib import pyplot as plt
```

Par la suite, pour produire un graphique, il faut faire appel à la fonction `plot()` de `matplotlib.pyplot`, puis à la fonction `show()` du même module :

— `plot()` génère le graphique en mémoire. On peut lancer plusieurs fois d'affilée cette fonction pour afficher plusieurs courbes dans le même graphique.

— `show()` permet l'affichage du graphique. On a tendance à l'oublier, mais si on ne fait pas appel à cette fonction, rien ne s'affiche !

Les arguments de la fonction `plot()` sont x,y et le type de courbe. x et y correspondent respectivement aux valeurs des abscisses et des ordonnées de la courbe à tracer. Il peut s'agir de tableaux numpy à une dimension ou de listes (si possible, utilisez des tableaux numpy). Le dernier argument est une chaîne de caractères décrivant le type de courbe attendu, par exemple '-' pour une ligne continue et '*' pour un affichage point par point. Voici un exemple récapitulant tout cela :

```
import numpy as np
from matplotlib import pyplot as plt
x = np.linspace(0,5,20)
y = np.sin(x) + 1
plt.plot(x,y,'*')
plt.show()
```

La fonction `plot()` accepte des arguments supplémentaires, dont les principaux sont résumés dans le tableau ci-dessous. Notons que les arguments de couleur et de marqueurs peuvent être rassemblés en un seul - par exemple : 'ro' affichera la courbe en rouge (r) avec des ronds (o).

nom	type	utilité	exemples de valeurs
color	str ou tuple	couleur de la courbe	'r' (rouge), 'black', (0.5,0,1) (RGB)
marker	str	type de marqueur	'*' (étoile), '.' (point), 'o' (rond)
linewidth	int	largeur ligne ou symbole	0 (pas de ligne), 2
label	str	texte de la légende	'Expérience', `r'coeff $\alpha (m.s^{-1})$'`

TABLE 2 – Arguments optionnels de la fonction `plot()`

Pour tracer plusieurs courbes, il suffit d'utiliser la fonction `plot()` plusieurs fois avant d'appeler `show()`. Dans ce cas, il est intéressant de distinguer les courbes grâce à une légende. Pour cela, il faut d'une part qu'un argument `label` soit défini lors de l'appel à la fonction `plot()` pour chaque courbe. D'autre part, il faut utiliser la fonction `legend()` du module `matplotlib.pyplot`, qui permet l'affichage de la légende sur le graphique. Cette fonction doit être appelée avant `show()` si l'on veut qu'elle soit prise en compte. Voici un exemple :

```
import numpy as np
from matplotlib import pyplot as plt
x = np.linspace(0,5,20)
y1 = np.sin(x) + 1
y2 = 1.5*np.cos(x)
plt.plot(x,y1,'r*', label='sinus')
plt.plot(x,y2, label='cosinus', marker = 'o', color = 'black')
plt.legend()
plt.show()
```

Voyons à présent les principales fonctions permettant de modifier l'apparence des axes du graphique.

— Pour le texte, on utilise les fonctions `xlabel()` et `ylabel()`, toujours situées dans le module `matplotlib.pyplot`. ces fonctions prennent en argument la chaîne de caractères correspondant au titre de l'axe. Notez que l'on peut utiliser la syntaxe latex, ce qui est intéressant pour afficher des lettres grecques ou des caractères en indice ou exposant.

— On peut fixer les échelles grâce aux fonctions `xlim()` et `ylim()`. Ces fonctions attendent comme argument deux nombres : la valeur minimale et maximale sur l'axe considéré. Là encore, il s'agit de fonctions du module `matplotlib.pyplot`.

— Pour contrôler le placement des 'ticks' (traits sur l'axe avec la valeur associée), il existe les fonctions `xticks()` et `yticks()`, qui prennent en argument un tableau numpy contenant les positions des ticks.

Il est également possible d'utiliser des échelles d'axe logarithmes, d'utiliser des noms à la places de valeurs sur les axes, ect... mais nous n'entrerons pas ici dans ces détails. Ci-dessous un code mettant en oeuvre les fonctions sur les axes.

```
# imports, définitions x,y
plt.plot(x,y1,'r*')
plt.xlabel('vitesse ($m.s^{-1}$)')
plt.ylabel('coefficient de frottement $\eta (m.s^{-2})$')
plt.xticks(np.linspace(0,5,5))
plt.ylim(0.5, 1.9)
plt.show()
```

Pour terminer, voyons comment gérer le graphique en général.

— Il est possible de fixer la taille et la résolution du graphique à l'aide de la fonction `subplots()`. Cette fonction prend deux paramètres optionnels : figsize est un tuple qui fixe la taille de la figure (en inch...) et dpi est un entier qui fixe la résolution (dot per inch).

— Il est possible de sauvegarder le graphique sous forme d'image. Pour cela, on utilise la fonction `savefig()`, qui prend comme argument le nom du fichier à créer (l'extension permet d'indiquer le format d'image souhaité, par défaut : PNG). Une option utile est de

fixer l'argument `bbox_inches='tight'`, pour éviter d'éventuels problèmes d'ajustement des bords de l'image.

Attention, la fonction `savefig()`, comme les autres fonctions de matplotlib.plt, doit être appelée avant l'appel de la fonction `show()` car cette dernière 'vide' le graphique. D'autres fonctionnalités concernant le graphique en général sont la possibilité d'ajouter des annotations ainsi que les graphiques imbriqués ou multiples. Une fois de plus nous n'entrerons pas dans ces détails, il faudra aller se renseigner sur internet si vous en avez un jour besoin !

Le code ci-dessous reprend l'ensemble des éléments vus précédemment :

```
# imports, définitions x,y
plt.subplots(figsize=(3,3), dpi=300)
plt.plot(x,y1,'r*')
plt.xlabel('vitesse ($m.s^{-1}$)')
plt.xticks(np.linspace(0,5,5))
plt.savefig('test.png', bbox_inches='tight')
plt.show()
```

Cela conclut ce chapitre de prise en main de matplotlib, voici quelques exercices pour pratiquer.

Exercice 10 :

a) Afficher le graphique y=sin(x), pour x variant de 0 à 3π.

b) Ajouter y=atan(x). Regarder ce qu'il se passe si on utilise la fonction `show()` avant le second appel de `plot()`.

c) Modifier le titre des axes, ajuster la couleur et l'épaisseur des lignes.

d) Créez une fonction qui prend un argument un nom de fichier et affiche sous forme de graphique les données contenues dans le fichier. Testez avec le fichier généré lors de l'exercice du chapitre précédent.

e) Modifiez la fonction précédente pour que la fonction enregistre le graphique correspondant aux données du fichier sous forme d'image au lieu de l'afficher.

2.3 Lecture et écriture de fichiers avec pandas

La librairie `pandas` permet de manipuler des tableaux de données. Elle est à l'interface entre les bases de données et les tableaux numériques de `numpy`. L'objet au coeur de ce module est nommé `DataFrame`. Il s'agit d'un tableau à deux dimensions contenant des données (qui peuvent être numériques ou autres : chaînes de caractères, dates, ...) ainsi que des noms pour les lignes

et les colonnes. Cette librairie est très polyvalente et contient des fonctions permettant le traitement avancé des données. Dans le cadre de ce cours, nous n'utiliserons qu'une petite partie de ce module : les fonctions permettant la lecture et l'écriture de fichier. Ces fonctions sont en effet très pratiques et permettent d'utiliser des formats de fichier autrement inaccessibles, notamment Excel. Voici dans un premier temps une présentation rapide de l'utilisation des `DataFrames` :

```
from pandas import DataFrame as df
import numpy as np
t = np.linspace(0, 10, 100)
V = np.sin(t) + 0.1
I = 2*np.cos(t) - 0.3
data = df({"Voltage" : V, "Intensity" : I}, index=t)
print(data.head())
```

Dans ce code, on voit la création d'un objet `DataFrame` en utilisant la fonction `DataFrame()` (renommée `df` dans l'exemple), qui prend comme argument un dictionnaire (clé = nom de la colonne, valeur = tableau numpy contenant les valeurs de la colonne). Le paramètre optionnel 'index' permet de spécifier le nom des lignes. Après sa création, on peut avoir une idée du contenu du DataFrame à l'aide de la méthode `head()`, qui affiche le nom des colonnes ainsi que les 5 premiers éléments.

Une fois mis sous forme de DataFrame, il est possible de sauvegarder facilement les données. Nous verrons ici l'utilisation de deux types de format de fichiers : les fichier csv (coma separated variables) d'une part, dans lequel les colonnes sont séparées par une virgule (ou autre...) et d'autre part le format Excel. Voici les fonctions à utiliser dans ces deux cas :

— La méthode `to_csv()` permet l'écriture d'un fichier au format csv.

— La méthode `to_excel()` permet l'écriture d'un fichier au format xls ou xlsx de Microsoft Excel.

Ces fonctions attendent comme argument le nom du fichier à créer ainsi qu'un grand nombre d'arguments optionnels permettant de spécifier la mise en forme du fichier. Voir la documentation pour plus de détails ! Voici un exemple d'écriture de fichier au format csv :

```
import pandas as pd
# création dataframe : voir code précédent
data = df({"Voltage" : V, "Intensity" : I}, index=t)
data.to_csv("mesure.txt")
```

Pour la lecture de fichier, les fonctions de `pandas` permettent de générer un `DataFrame` à partir d'un fichier de données, qui peut contenir des titres de colonne. Ces fonctions sont particulièrement utiles pour lire les données issues d'instruments de mesure. Ici encore, plusieurs formats sont supportés, dont les principaux sont :

— Le format 'csv' . Ce type de fichier est lu grâce à la fonction `read_csv()` disponible dans le module `pandas`. Cette fonction attend en argument le nom du fichier à lire. Un argument nommé 'sep' (valeur par défaut ',') permet de définir le caractère de séparation des colonnes.

— Le format Microsoft Excel (.xls ou .xlsx) est lisible grâce à la fonction `read_excel()`. L'arguement 'sheet-name' (valeur par défaut 0), permet de préciser le numéro ou le nom de la feuille à utiliser.

Ces deux fonctions de lecture attendent comme argument le nom du fichier à lire. Cependant, plusieurs arguments facultatifs peuvent également être précisés et sont communs aux deux fonctions. En voici quelques exemples :

— L'argument 'header' indique le numéro de ligne contenant le nom des colonnes. Mettre à `None` s'il n'y a pas de noms de colonnes dans le fichier.

— L'argument 'skiprows' indique le nombre de lignes du début du fichier que l'on souhaite ignorer.

— L'argument 'index_col' indique le numéro de colonne à utiliser comme index (= noms des lignes)

Voici un exemple de lecture des données d'un fichier excel :

```
import pandas as pd
donnees = pd.read_excel('data.xlsx')
print(donnees.head())
```

Une fois les données sous forme de DataFrame, il est possible de travailler directement dessus grâce à différentes fonctions de la librairie `pandas`. Nous n'aborderons pas cela ici et nous extrayons simplement un tableau `numpy` à partir du DataFrame grâce à la méthode `to_numpy()`, qui retourne un `ndarray`. Il est alors possible de manipuler ce tableau grâce aux fonctions et opérateurs vus précédemment. Par exemple :

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt

d = pd.read_csv("mesure.txt")
a = d.to_numpy()
plt.plot(a[:,0], a[:,1], label=d.columns.values[1])
plt.plot(a[:,0], a[:,2], label=d.columns.values[2])
plt.legend()
plt.show()
```

Depuis la dernière version de `pandas`, une méthode `plot()` a même été implémentée sur les objets `DataFrame` pour vous faciliter encore plus la tâche ! L'exemple ci-dessous reprend la fonctionnalité de l'exemple précédent en mettant cela à profit. Le code nécessaire pour lire un fichier et l'afficher sous forme de graphique devient alors extrêmement court.

```
import pandas as pd
import numpy as np
```

```
from matplotlib import pyplot as plt

d = pd.read_csv("mesure.txt")
d.plot()
plt.show()
```

Exercice 11 :

a) Créer un DataFrame à partir d'un dictionnaire (nom, tableau numpy). Regarder l'effet de l'argument 'index'.

b) Enregistrer cette dataframe au format excel ou csv

c) Modifier votre fichier en ajoutant des lignes d'en-tête. Lire le fichier en utilisant l'argument 'skiprows'.

d) Affichez avec matplotlib les données issues d'un fichier excel présent sur votre ordinateur