

Python pour les statistiques

Fiche 1 : Rappels

M2 Modélisation statistique, 2023–2024

But du cours

Le but de ce cours est d'acquérir les compétences en Python pour pouvoir aborder tout problème statistique et de *data science*, au niveau M2 et en stage, en confiance. Voilà les compétences qui seront acquises dans ce cours :

- Des bases de programmation solides *avant* d'utiliser les outils élaborés existant.
- Manipuler des vecteurs de données numériques avec Numpy.
- Créer des graphiques lisibles avec Matplotlib.
- Lire et manipuler des données avec Pandas.
- Visualisation de données avec Seaborn.
- Utilisation basique du *machine learning* de Scikit-learn.

Ce document fournit un rappel condensé des bases de Python (le langage Python sans importation de module). Il ne vise pas à être un cours complet, et il est toujours bon de se référer à la [documentation officielle](#), notamment la page de la [bibliothèque standard](#), qui décrit tous les types natifs de Python (`str`, `list`, `tuple`, `dict`, etc.) et les fonctions et méthodes associées. En plus de ça et bien sûr de vos cours des années passées, des explications sur la syntaxe (notamment des boucles `for`, des structures de conditionnement `if`, `while`, etc.) sont trouvables sur la page du [tutoriel officiel Python](#).

NB : De manière générale, la meilleure compétence à acquérir en ce qui concerne l'informatique n'est pas technique : à mon sens c'est la capacité à aller **fouiller dans les documentations** (et sur votre moteur de recherche favori) pour trouver ce qu'on cherche.

Ainsi, malgré toutes les ressources existantes en ligne, voici encore quelques pages qui ont le but de fonctionner comme un pense-bête, en énonçant quelques points important du langage de programmation Python. Ces rappels sont suivis d'une sélection d'exercices.

Table des matières

1	Syntaxe	2
2	Anaconda et Spyder	3
3	Quelques objets Python vus en M1	4
3.1	List	4
3.2	Tuple	4
3.3	String	5
3.4	Dict	6
3.5	Set	6

4 Structures	7
4.1 Boucles et structures conditionnelles	7
4.2 Compréhension	8
4.3 Compréhension filtrée	8
4.4 Exceptions	8
5 Syntaxes utiles	9
5.1 Chaînes formatées	9
5.2 Lire et écrire dans un fichier	10
5.3 Zip : deux listes en une	11
5.4 Étoiles dans les arguments de fonctions	11
6 Exercices	12

1 Syntaxe

La syntaxe et la clarté sont primordiales en Python. L'usage [des conventions](#), en particulier dans l'utilisation des espaces, est fortement recommandé pour la lisibilité. Quelques exemples :

- **Le signe = est entouré d'espaces**, sauf quand il donne la valeur d'un argument optionnel dans l'appel d'une fonction (ou bien dans sa définition) :

```
x = ma_fonction('un argument', opt_arg=None)
```

- **Même chose pour les signes arithmétiques**, la majorité du temps (exceptions : à juger soi-même, recommandée pour distinguer les opérations prioritaires) :

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

- On met un **espace après une virgule**, mais pas d'espace avant. On ne met **pas d'espace immédiatement après des parenthèses ou crochets ouvrants**, ou immédiatement avant des parenthèses ou crochets fermants :

```
fibo = [1, 1, 2, 3, 5, 8, 13]
a, b = (1, 2, 3), 4
plt.plot(x, fibo)
```

- Idéalement, on **évite d'écrire des lignes trop longues** (plus de 79 caractères). On peut couper une ligne de code sur plusieurs lignes de plusieurs manières :

- En revenant à la ligne entre certains arguments d'une fonction :

```
x = ma_fonction(ma_variable_1, ma_variable_2,
                ma_variable_3, ma_variable_4)
```

- Lors d'une suite d'opérations :

```
x = (ma_variable_1
     + ma_variable_2
     + ma_variable_3
     + ma_variable_4)
```

À noter ici que si l'on n'avait pas mis de parenthèses, on aurait dû alors indiquer qu'une ligne continue sur la suivante en signalant la coupure par un backslash \.

- **Ne pas mettre plus d'espaces que nécessaire.** Par contre, il faut en mettre suffisamment quand nécessaire : **l'indentation** (de quatre espaces par défaut) **est nécessaire** pour définir un bloc de code, et elle doit être la même pour chacune des lignes de ce bloc. Ce que j'appelle des blocs de code sont, entre autres, les corps des fonctions, des boucles, des instructions conditionnelles. Tous ces blocs commencent après une ligne qui finit par un "deux points". Exemple :

```
def ma_somme(x, y):
    # Cette fonction renvoie la somme de x et de y, sauf si l'un vaut None.
    if x is None or y is None:
        raise Exception('somme de None impossible')
    z = x + y
    return z # Oui, on pourrait se passer de z et écrire `return x + y`.
```

Au passage, noter qu'il n'y a pas d'espaces avant un "deux points".

- Finalement, **espacer le code verticalement** en sautant deux lignes entre chaque définition de fonction. Ne pas forcément chercher à sauter plus de lignes que ça, sauf s'il paraît raisonnable de séparer deux parties logiques du code.

2 Anaconda et Spyder

On impose dans les cours du master l'utilisation d'un éditeur de code Python appelé **Spyder**. Il présente l'avantage d'être facilement installable via la distribution Python [Anaconda](#), de disposer d'outils d'analyse de code, d'un *debugger*, d'un *profiler*.

Écrire un script (ce que permet de faire Spyder correctement) est dans un sens plus exigeant que bidouiller avec un *notebook*, c'est pourquoi on demandera ici de maîtriser l'écriture (basique) de scripts.

Pour **l'organisation** : je vous recommande de créer un dossier spécifique à votre cours actuel, puis dans ce dossier, un sous-dossier par exercice ou séance de TD / TP, contenant le ou les scripts qui répondront aux problèmes posés.

Les astuces à connaître avec Spyder :

- La touche F5 **exécute le script entier** dans la console. L'utiliser au moins une fois au début et à la fin du travail, car :
 - Par défaut cela synchronise le répertoire de travail de la console avec le répertoire du script, ce qui évite les surprises avec la manipulation des fichiers extérieurs (.txt, .csv, etc.)
 - À la fin, on doit vérifier que le script s'exécute en entier, sans bug. Il faudrait même le lancer dans une nouvelle console (évite les surprises dues à un environnement de travail non vide).
- La touche F9 **exécute la ligne courante (ou une sélection)** dans la console. Pratique pour tester une petite partie du code.
- Pour **indenter / désindenter plusieurs lignes à la fois**, les sélectionner puis appuyer sur Tab ou Shift+Tab pour indenter ou désindenter. N.B : il n'est pas nécessaire de sélectionner toute les lignes proprement, une partie suffit (permet de faire ça rapidement avec le clavier en utilisant la touche Shift et les flèches directionnelles).

- **Commenter / décommenter rapidement** peut être très pratique. Par défaut `Ctrl+1` accomplit cette tâche sur la ligne courante ou celles qui sont sélectionnées. Si ce raccourci n'est pas pratique – et c'est le cas avec un clavier français –, changez cela dans “Outils > Préférences > Raccourcis clavier” (par exemple je recommande `Ctrl+T` en remplacement).
- Quelque chose d'assez confidentiel, pour les amateur·ices de notebooks : séparer des bouts de codes entre des lignes de commentaires commençant par `#%%` a l'effet de créer des “cellules” de codes, visibles par les séparations visuelles horizontales dans l'éditeur. Avec le curseur à l'intérieur d'une cellule, `Ctrl+Entrée` **exécute cette cellule** dans la console.
- Utilisez les raccourcis clavier pour **basculer rapidement de l'éditeur à la console** et inversement (allez voir les raccourcis proposés et n'hésitez pas à les modifier si besoin).

3 Quelques objets Python vus en M1

Cette liste d'objets et les méthodes décrites ne sont pas exhaustives. Ne pas hésiter à consulter la [documentation des types standards](#) et le [Tutoriel Python officiel](#).

3.1 List

La `list` est un objet *mutable* omniprésent en Python. Elle peut contenir toute sorte d'objets (même des listes), et l'on récupère le n -ième élément d'une liste `l` avec `l[n]` – l'indexation Python commençant à 0. L'essentiel :

- Définir une liste : `l = [1, 'deux', 3.33, [4]]`.
- Ajouter un élément `x` à une liste : `l.append(x)`.
- Extraire la sous-liste, des éléments i à $j-1$: `l[i:j]`. Omettre l'un ou l'autre des indices signifie : extraire depuis le début / jusqu'à la fin.
- Extraire le n -ième élément à partir de la fin : `l[-n]`. *Attention au “piège”* : le dernier élément est désigné par l'indice `-1`. Penser comme si la liste était repliée sur elle même comme un tore : le dernier élément vient juste avant le premier élément, qui lui est indexé par 0.
- Concaténation de liste : `l = [1, 'deux'] + [3.33, [4]]` définit bien la même liste que ci-dessus.
- Utilisation comme une *pile* : exemple avec `pile = ['a', 'b', 'c']`
 - On ajoute un élément à la pile : `pile.append(42)`. La variable `pile` vaut maintenant `['a', 'b', 'c', 42]`.
 - On utilise la méthode `pop()` des objets `list` pour récupérer le dernier élément tout en le supprimant de la pile :

```
x, y = pile.pop(), pile.pop()
print(f'x={x}, y={y}') # x=42, y=c
print(pile) # ['a', 'b']
```

3.2 Tuple

Le `tuple` est un objet qui s'utilise globalement comme une liste, sauf qu'il est *immutable*. À noter que dans les exemples qui suivent, les parenthèses ne sont pas toujours obligatoires – ce sont les virgules qui définissent le tuple.

- Définir un tuple : `t = (1, 'deux', 3.33, [4])`.

- Indexation et slicing : `t[n]` renvoie le n -ième élément, `t[i:j]` renvoie un tuple constitué des éléments indexés de i à $j - 1$.
- Pour la définition de variables, on peut définir deux (ou plus) variables en même temps en utilisant des tuples. Exemple : `x, y = 1, 'deux'`, qui est équivalent à `x = 1`, suivi de `y = 'deux'`.
- Usage standard pour le renvoi de plusieurs valeurs, exemple pour l'algorithme de Bézout :

```
def bezout(a, b):
    """ Renvoie (p, u, v) tels que  $u*a + v*b = p = \text{pgcd}(a, b)$ . """
    q, r = a // b, a % b
    if r == 0:
        return (b, 1, 1-q)
    p, u, v = bezout(b, r)
    return (p, v, u-q*v)
```

3.3 String

Les chaînes de caractères sont *immuables*, indexables comme les listes ou les tuples. Elles s'écrivent délimitées soit par des apostrophes `'`, soit par guillemets `"` – il peut être pratique d'utiliser les guillemets pour définir la chaîne `"j'aime le chocolat"`, plutôt que d'utiliser les syntaxes moins lisibles `'j\'aime'` ou `'j' aime'`.

- Comme les listes, on les concatène avec l'opérateur `+`. Une autre possibilité est d'utiliser la méthode `join()`. Exemple :

```
x = 'Bonjour' + ' le' + ' monde!'
y = ' '.join(['Bonjour', 'le', 'monde!'])
```

Ici les deux variables créées sont égales. Attention aux espaces présents dans les différentes chaînes.

- Pour décomposer une chaîne de caractères autour des espaces :

```
l = 'Bonjour le monde!'.split() # l = ['Bonjour', 'le', 'monde!']
```

- Remplacer une sous-chaîne par une autre :

```
s = "J'ai un ami, Bruno".replace('un', 'deux').replace('ami', 'amis')
# s = "J'ai deux amis, Brdeuxo"
```

- De nombreuses méthodes permettent de formater des chaînes de caractères. Expérimenter avec la fonction `print_titre()` suivante :

```
def print_titre(titre, deco='-', n=40):
    titre = titre.center(len(titre)+2)
    print(n * deco, titre.center(n, deco), n * deco, sep='\n')
```

- Pour une liste complète des méthodes utilisables, voir [la doc](#).
- Les chaînes de documentation (*docstrings*) sont des chaînes spéciales délimitées par trois guillemets `"""`. Elles peuvent être écrites sur plusieurs lignes, contenir des guillemets, des apostrophes. Il faut en fait les voir comme des commentaires servant à la documentation du code que l'on écrit. Une bonne pratique pour prendre l'habitude de documenter son code est d'écrire une petite docstring (une ligne, pour les exercices et projets du cours) pour chaque fonction écrite, qui décrit brièvement ce que fait la fonction. Exemple :

```
def addition(x, y):
    """ Effectue l'addition de x et y. """
    return x + y
```

Chaînes formatées. Pour voir comment insérer des expressions (variables, calculs, etc.) dans une chaîne de caractère, voir la section 5.1. On appelle ça le formatage de chaîne de caractères. Cela permet par exemple d’adapter automatiquement et en utilisant une syntaxe simple les titres de vos graphiques :

```
N_repet = 1000
# ... calculs et affichage d'un graphique ...
plt.title(f'Histogramme (échantillon de taille {N_repet})')
```

3.4 Dict

Les dictionnaires (de type `dict`) sont des objets *mutables* qui servent à associer des *valeurs* (objets immutables) à des *clés* (objets quelconques).

- Définition d’un dictionnaire : `d = {'clé': 'valeur', 1: 42, 0: ['une', 'liste']}`. Le dictionnaire précédent a trois valeurs, associées à trois clés.
- Les clés sont obtenues avec `d.keys()`. Ici, renvoie un itérable – une “quasi liste” – équivalent à `['clé', 1, 0]`.
- Les valeurs sont obtenues avec `d.values()`. Ici, renvoie un itérable équivalent à la liste suivante : `['valeur', 42, ['une', 'liste']]`.
- Les paires (clés, valeurs) sont obtenues avec `d.items()`. Ici, renvoie un itérable équivalent à la liste de tuples `[('clé', 'valeur'), (1, 42), (0, ['une', 'liste'])]`.
- On récupère la valeur indexée par la clé `key` avec `d[key]`. Attention, si `key` n’est pas une clé du dictionnaire, ceci soulève une exception `KeyError`.
- On peut la supprimer avec `del d[key]`.
- On peut la modifier en l’utilisant à gauche d’une définition : `d[key] = nouvelle_valeur`.
- On peut créer une nouvelle association (clé, valeur) de la même manière, par définition : `d[nouvelle_cle] = nouvelle_valeur`. On note donc que cette syntaxe ne soulèvera jamais d’exception `KeyError`.
- Utile dans des cas particuliers, le code `x = d.setdefault(key, valeur)` a les effets suivants :
 - Si `key` est déjà une clé du dictionnaire, alors c’est équivalent au code `x = d[key]`. La variable `valeur` est alors ignorée.
 - Sinon, alors c’est équivalent au bloc de code
`d[key] = valeur`
`x = d[key]`
Ici, l’exception `KeyError` n’est jamais soulevée.
- Le dictionnaire vide : `{}`.

3.5 Set

Un ensemble (type `set`) est un objet *mutable*, contenant des objets *immutables*. Comme un ensemble mathématique, ses éléments ne sont pas ordonnés, et il ne peut contenir deux copies d’éléments identiques.

- Définir un ensemble : `s = {1, 'deux', 3.33}`.
- Attention, `{}` ne définit pas un ensemble vide mais un dictionnaire. L’ensemble vide est `set()`.

- Ajouter un élément avec `s.add('nouvel_élément')`. Si l'on ajoute un élément déjà présent (par exemple avec `s.add(1)`, rien ne se passe).
- Supprimer un élément avec `s.remove(x)`. Si `x` n'est pas un élément de `s`, soulève une exception `KeyError`.

4 Structures

4.1 Boucles et structures conditionnelles

Les structures usuelles `if ... elif ... else`, `for` et `while` sont bien sûr présentes en Python.

- Comme les définitions de fonctions, ces structures sont définies par un bloc de code correctement indenté. Ces structures peuvent être imbriquées les unes dans les autres. Exemple/exercice : deviner ce que fait la fonction suivante.

```
def tables(n):
    if n > 12:
        print("On n'affiche pas les tables pour n >= 13.")
        return
    for i in range(1, n+1): # itère sur i de 1 à n
        print('Table de', i)
        j = 1
        while j <= n: # en pratique, itère aussi sur j de 1 à n
            j += 1
            print(f'- {i} * {j} = {i*j}')
```

- Attention pour un enchaînement de conditions exclusives : les mots-clés `if`, `elif` et `else` doivent être sur la même ligne d'indentation. Exemple :

```
def une_fonction_inutile(n):
    if n % 4 == 0:
        s = 'n est divisible par 4'
    elif n % 2 == 0:
        s = 'n est divisible par 2, mais pas par 4'
    else:
        s = 'n est impair'
    return s
```

- On souligne que les boucles `for` s'utilisent naturellement avec tout objet *itérable*. Si `ma_liste` est une liste qui vaut `[1, 3, 'deux', x]`, le code

```
for i in ma_liste:
    ... # [fait qqch avec i]
```

aura l'effet suivant : le bloc de code définissant la boucle sera répété avec la variable `i` prenant successivement les valeurs `1`, `3`, `'deux'` puis `x` – N.B. : si la variable `x` désigne un objet mutable, `i` sera une référence vers celui-ci, pas une copie.

- Les tuples, les ensembles et les dictionnaires sont aussi itérables. Pour les dictionnaires, la variable de la boucle (`i` dans l'exemple ci-dessus) vaudra les valeurs possibles des clés. Pour itérer sur les valeurs d'un dictionnaire `d`, utiliser `for v in d.values()`, et pour itérer sur les couples (clé, valeur), utiliser `for (k, v) in d.items()`.
- Si l'on veut utiliser directement une itération comme ci-dessus, mais en ayant accès à l'indice de l'élément, on utilisera : `for (i, y) in enumerate(ma_liste)`. Dans le corps de la boucle, la

variable `i` contiendra l'indice de l'élément (ici noté `y`) – c'est-à-dire le compteur pour le nombre d'itérations effectuées. Exemple :

```
for (i, y) in enumerate(ma_liste):
    print(i, '. ', y, sep='')
# Cette boucle affichera:
# 0. 1
# 1. 3
# 2. 'deux'
# 3. [la représentation de la variable x]
```

4.2 Compréhension

La compréhension est une syntaxe présente en Python et qui facilite la construction de certains objets avec une boucle implicite. Elle sert à créer des listes, des tuples, des dictionnaires, des ensembles comme si l'on ajoutait des éléments un à un à partir d'une boucle `for`. Exemple pour une liste : le code `l = [i**2 for i in range(10)]` a le même effet que :

```
l = []
for i in range(10):
    l.append(i**2)
```

La syntaxe est très similaire pour les ensembles. Pour les tuples, il faut utiliser la syntaxe

```
t = tuple(i**2 for i in range(10)).
```

Pour un dictionnaire, la syntaxe est la suivante : `d = {clef(i): valeur(i) for i in ...}`.

Exemple :

```
d1 = {i: i**2 for i in range(10)}
# Équivalent à :
d2 = {}
for i in range(10):
    d2[i] = i**2
```

4.3 Compréhension filtrée

Pour effectuer une compréhension qui ne concerne que certains éléments de la liste en question, on rajoute `if [condition]` à la fin de la compréhension. Exemple : en supposant que l'on dispose de la liste de mots `basse_cour = ['poule', 'canard', 'oie', 'cochon', 'vache', 'chèvre']`, le code `[x for x in basse_cour if 'o' in x]` produira la liste `['poule', 'oie', 'cochon']`.

4.4 Exceptions

Il est parfois utile de définir des exceptions (des erreurs porteuses de messages qui arrêtent l'exécution du programme, à moins d'être interceptées et traitées dans le programme) spécifiques au code que l'on écrit. Par exemple si l'on veut écrire la fonction `produit_scalaire(x, y)`, on pourra écrire :

```
# Définit une classe `MauvaiseDimension` qui hérite de `Exception`.
class MauvaiseDimension(Exception):
    pass

def produit_scalaire(x, y):
```



```

if len(x) < len(y) or len(y) < len(x):
    raise MauvaiseDimension('vecteurs de longueurs différentes')
return sum(xi * yi for (xi, yi) in zip(x, y))

```

Ce programme définit d’abord une exception nommée `MauvaiseDimension`. C’est tout simplement une classe dérivée de la classe `Exception`, et qui ne rajoute rien de spécial (d’où le `pass` qui permet de signaler simplement la fin d’un bloc de code vide). Pour comprendre l’utilisation de `zip`, voir la section 5.3.

Ensuite, dans la fonction `produit_scalaire(x, y)`, si les longueurs de `x` et `y` ne sont pas les mêmes, on utilise le mot-clé `raise` pour soulever une nouvelle instance de l’exception, munie d’un message (grâce à un constructeur hérité de la classe mère `Exception`).

Si l’on essaye alors d’exécuter `produit_scalaire([1,2], [3])`, le programme plantera, et affichera notre message d’erreur. Par contre, l’on peut utiliser la structure `try ... except` pour traiter l’exception :

```

try:
    x, y = [1, 2], [3]
    print('*** essai ***')
    print('Le produit scalaire de', x, 'et de', y, 'vaut', produit_scalaire(x,y))
except MauvaiseDimension:
    print('On traite l\'exception et le programme continue.')

```

Ce bout de code ne renvoie pas d’erreur et n’interrompt pas le programme. Il affichera :

```

*** essai ***
On traite l'exception et le programme continue.

```

et le programme continuera en effet son exécution à la ligne suivante. À noter, le programme exécute en effet les lignes du bloc `try` : qui ne soulèvent pas d’exception (ici, les deux premières lignes sont exécutées).

5 Syntaxes utiles

5.1 Chaînes formatées

La manière la plus “intuitive” et rapide de créer une chaîne formatée est d’utiliser la syntaxe `f'ma chaîne {mon_exp}`. Des exemples :

```

x, y = 1, 2
a = f"L'addition de {x} et {y} vaut {x+y}."
# a = "L'addition de 1 et 2 vaut 3."

z = [x, y, x+y]
nom = 'liste'
b = f'Voilà une {nom}: {z}.'
# b = 'Voilà une liste: [1, 2, 3].'

```

On note que tout ce qui peut s’afficher peut s’insérer ainsi dans une chaîne formatée.

Il y a d'autres syntaxes qui permettent de formater des chaînes, et *beaucoup* de possibilités pour des formatages spécifiques, voir [la doc](#). Par exemple on peut créer la chaîne suivante :

```
titre = 'Un beau titre'
titre = f'{" "+titre+" ":~^50}'
# titre = '~~~~~ Un beau titre ~~~~~'
```

5.2 Lire et écrire dans un fichier

On utilise la fonction `open('nom_du_fichier', mode, encoding='utf-8')` pour manipuler les fichiers du système d'exploitation. Cette commande renvoie un objet qui peut être utilisé en lecture ou en écriture, en fonction de l'argument `mode`. Les 4 modes à connaître sont les suivants :

- `'r'` (par défaut) : lecture du fichier (on ne peut pas le modifier).
- `'w'` : écriture du fichier (s'il existe un fichier avec le nom donné, il sera écrasé).
- `'a'` : mode *append*, c'est-à-dire écriture à la fin du fichier (le contenu existant ne sera pas écrasé).
- `'r+'` : lecture du fichier, avec écriture possible (on peut donc modifier une partie du contenu).

De manière générale, pour un usage simple on utilisera surtout les modes `'r'` pour lire un fichier et `'w'` pour écrire dans un nouveau fichier.

En pratique. Pour ouvrir le fichier, on recommande d'utiliser le mot-clé `with`, qui permet d'ouvrir le fichier dans un bloc de code (et seulement dans celui-ci). Par exemple, pour lire le contenu d'un fichier `fichier.txt` :

```
with open('fichier.txt', encoding='utf-8') as f:
    print('Voilà le contenu du fichier:')
    print(f.read())
# À la fin du bloc indenté, la variable f n'est plus valide, le fichier est fermé.
```

La méthode `f.read()` renvoie tout le contenu du fichier dans une chaîne de caractère. Pour lire le fichier ligne par ligne, on peut utiliser la méthode `f.readline()`. Le premier appel renvoie la première ligne, le second renvoie la ligne suivante, etc, jusqu'à renvoyer une chaîne de caractère vide ". Notons qu'un caractère "saut de ligne" `'\n'` est présent à la fin de chacune des lignes.

Enfin, l'objet `f` peut être utilisé comme un itérateur (donc dans une boucle `for`), pour itérer sur les lignes du fichier. Par exemple voici un code qui affiche le nombre de mots de chaque ligne du fichier `fichier.txt`.

```
with open('fichier.txt', encoding='utf-8') as f:
    for i, line in enumerate(f):
        n = len(line.split())
        print(f'Ligne {i+1}: {n} mots.')
```

Enfin, l'écriture dans un fichier se fait avec la méthode `f.write(contenu)`, où `contenu` est une chaîne de caractère. Exemple :

```
with open('fichier.txt', 'w', encoding='utf-8') as f:
    f.write('Ce fichier contient 2 lignes.\nCeci est la deuxième.')
```

On pourra se référer à [la doc](#) pour compléter cet aperçu bref.

5.3 Zip : deux listes en une

La fonction `zip` s'utilise dans la situation où l'on veut parcourir plusieurs listes à la fois. Ce que j'entends par là est la chose suivante : faire une boucle `for` qui récupère, à la i -ème itération de la boucle, le i -ème élément de chacune des listes. Un exemple et son équivalent "moins pythonesque" :

```
liste1, liste2 = [1, 2, 3], ['a', 'b', 'c']
for x, y in zip(liste1, liste2):
    print(f'x={x}, y={y}')

# Équivalent à:
for i in range(min(len(liste1), len(liste2))):
    print(f'x={liste1[i]}, y={liste2[i]}')
```

Ces deux codes affichent :

```
x=1, y=a
x=2, y=b
x=3, y=c
```

On peut bien sûr fournir à `zip` autant de listes que voulu, l'itération renverra un tuple de même longueur que le nombre de listes, qui itère élément après élément pour chacune des listes. Notons que l'objet renvoyé est un itérateur et non pas une liste. Si nécessaire, pour récupérer la liste constituée des couples $(a_i, b_i)_{i \geq 1}$ formés des éléments de deux listes `a` et `b`, on utilise `list(zip(a, b))`.

5.4 Étoiles dans les arguments de fonctions

Imaginons que vous voulez appeler une fonction `f` qui prend trois arguments, et que vous récupérez ces trois arguments d'une fonction `g` qui renvoie un tuple, une liste, ou autre itérable. Une erreur serait d'appeler directement `f(g())`, car dans cette situation le tuple est compris comme un seul argument. Il en manque donc deux.

Une solution possible est de déstructurer le tuple :

```
x, y, z = g()
f(x, y, z)
```

Cela fonctionne mais ce n'est pas l'idéal, il faut donner un nom à chacun des arguments, etc. Une autre solution est d'utiliser la syntaxe `f(*iterable)`, ce qui a pour effet de déstructurer l'itérable directement dans la liste d'arguments de la fonction : ainsi on peut appeler directement `f(*g())`.

Exemple :

```
l = [1, 2, 3]
print(l)      # [1, 2, 3]
print(*l)     # 1 2 3                                <- comme `print(1, 2, 3)`
```

Inversement, vous pouvez définir ainsi des fonctions qui prennent un nombre quelconque d'arguments, en utilisant la syntaxe de (dé)structuration lors de la définition d'une fonction. Ainsi, la signature `def f(a, b, *c)` : indique que la fonction `f` prend au moins deux arguments, et les arguments indiqués à partir du troisième seront stockés dans un tuple nommé `c`.

Comme exemple, nous pouvons réécrire une version de la fonction `str.join()` (rappel : le bout de code `'-'.join('a', 'b', 'c')` renvoie `'a-b-c'`).

```
def my_join(joint, *liste):
    # Initialisation nécessaire pour ne pas commencer par `joint`.
    s = '' if len(liste) == 0 else liste[0]
    for x in liste[1:]:
        s += joint + x
    return s

print(my_join('-', 'a', 'b', 'c')) # a-b-c
```

Double-étoiles pour les dictionnaire. Enfin, une syntaxe similaire (avec deux étoiles) permet de passer à une fonction des arguments “mots-clés”, que l’on utilise habituellement comme arguments optionnels. Démonstration :

```
# Une fonction prenant deux arguments optionnels `a` et `b`.
def f(a=1, b=2):
    return (a, a+b)

# Un appel qui utilise un dictionnaire.
d = {'b': 3, 'a': 5}
print(f(**d)) # (5, 8)
```

Inversement, une fonction peut regrouper dans un dictionnaire (parfois nommé `kwargs`) les arguments non-nommés dans sa définition de la manière suivante :

```
def f(a=0, **kwargs):
    # Dans cette fonction l'argument a est ignoré.
    print(kwargs)

f(a=1, b=2, c=3) # {'b': 2, 'c': 3}
```

De nombreuses fonctions (de la librairie Matplotlib en particulier) utilise cette syntaxe pour “passer” des arguments optionnels d’une fonction à une autre – c’est pourquoi ce `**kwargs` apparaît souvent dans la documentation.

6 Exercices

Exercice 1 : manipulations de listes.

- Écrire une fonction `max_min(liste)` qui prend en argument une liste de valeurs numériques `liste` et renvoie, sans utiliser les fonctions `max` et `min` de Python, un dictionnaire de la forme `{'max': max(liste), 'min': min(liste), 'imax': i, 'imin': j}`, où `i` et `j` sont les indices du maximum et du minimum de la liste.
- Écrire une fonction `sum_list(l1, l2)` qui prend deux listes en argument et renvoie la liste `l3` obtenue en faisant la somme de `l1` et `l2` de manière vectorielle. Si l’une des listes est plus longue que l’autre, on considérera que des zéros complètent la liste la plus courte afin d’atteindre la même longueur.
- Copier le bout de code suivant dans votre script :

```
def inverse(l):
    return l[::-1]
```

- i) Écrire une fonction `inverse_naif(1)` qui a le même fonctionnement, mais sans utiliser cette syntaxe `l[::-1]`. On pourra par exemple utiliser la compréhension de liste.
- ii) Comparer les temps d'exécution de ces deux fonctions, sur une liste suffisamment longue (10000 éléments). Pour ce faire, utiliser le *profiler* fourni avec Spyder :
 - Il n'est peut-être pas activé \leadsto cliquer sur "View > Panes" puis cocher *Profiler*.
 - Aller dans l'onglet *Profiler* (au niveau de l'explorateur de variables), puis sélectionner votre script en cliquant sur l'icône de dossier, à côté du bouton "Profile".
 - Les fonctions appelées par votre script devraient s'afficher dans la fenêtre, avec le temps total passé dans chacune de celle-ci, le nombre de fois où elles sont appelées, etc.

Exercice 2 :

- a) Écrire une fonction `fact(n)` qui calcule la factorielle $n!$.
 - b) Écrire une version récursive de la fonction précédente (c'est-à-dire une fonction qui s'appelle elle-même, comme dans la définition d'une récurrence).
- Si vous avez d'abord écrit une version récursive de la factorielle, écrivez-en une itérative.*

Exercice 3 : On rappelle la définition de la suite de Fibonacci par récurrence :

$$f_0 = 0, \quad f_1 = 1, \quad \text{et } f_{n+1} = f_n + f_{n-1}, \quad \forall n \geq 1.$$

- a) Écrire une fonction `fibonacci(n)` qui calcule le n -ième terme de la suite de Fibonacci.
- b) Écrire une version récursive de la fonction `fibonacci`.
- c) La fonction précédente est-elle efficace ? Comparez ses temps d'exécution avec la version itérative.
- d) Si la réponse à la question précédente est non, envisager une méthode récursive *efficace* pour le calcul de la suite de Fibonacci.

Exercice 4 : fonction à mémoire.

- a) Recopier le bout de code suivant :

```
def mapile(x, pile=[]):
    pile.append(x)
    return pile
```

Que renverra `mapile(0)` ? Vérifier en exécutant cette ligne deux fois de suite.

- b) La variable `pile` agit comme une variable globale, elle n'est pas effacée lors d'un second appel. Analyser le code suivant et comprendre pourquoi cette version récursive apparemment naïve de Fibonacci ne plante pas quand on l'appelle pour de grandes valeurs de n .

```
def fibo_mem(n, pile=[0,1]):
    if len(pile) > n:
        return pile[n]
    else:
        pile.append(fibo_mem(n-1) + fibo_mem(n-2))
        return pile[-1]
```

Le but de cet exercice est de bien comprendre ce qui se passe dans le code précédent.

Exercice 5 : Un *bon parenthésage* est un “mot” constitué uniquement de parenthèses ouvrantes et fermantes de sorte que (a) à la lecture du mot, l’on ne ferme pas plus de parenthèses qu’il n’y en a d’ouvertes, et (b) à la fin du mot, toutes les parenthèses sont fermées. Exemples de bons parenthésages :

(), ()(), (()), ((()()))(), ou encore ((()())((()())()()))

Mauvais parenthésages :

)(, ((), (()).

- a) Écrire une fonction `combine(liste1, liste2)` qui, étant donnée deux listes de parenthésages (chaînes de caractères) renvoie une liste de tous les parenthésages possibles de la forme $(x)y$, où x est un parenthésage de la liste 1 et y un parenthésage de la liste 2.

Pour tester cette fonction : `combine(['', '()'], ['()', '(()'])` doit renvoyer la liste suivante : `['()()', '()()', '(()()', '(()()())']`.

Le but de cet exercice est d’obtenir, pour un entier pair fixé n , la liste de tous les bons parenthésages de longueur n .

- b) En utilisant soit une “fonction à mémoire”, soit une variable dictionnaire globale, écrire une fonction `parenth(n)` qui renvoie la liste voulue.

On utilisera pour cela la fonction `combine`, et le fait que l’on peut décomposer *de manière unique* un parenthésage p en $p = (q_1)q_2$, où q_1 et q_2 sont de bons parenthésages (éventuellement vides). Il faudra réfléchir aux longueurs possibles des parenthésages q_1 et q_2 , si p est de longueur n .

- c) Vérifier, pour tous les entiers pairs plus petits que 20, que le nombre de bons parenthésages de longueur n vaut bien

$$\frac{2}{n+2} \binom{n}{n/2}.$$

Exercice 6 : génération de nombres premiers.

- a) Écrire une fonction `isprime(n)` qui renvoie un booléen indiquant si l’entier n est premier.

On cherchera à diviser n par tous les entiers plus petits que \sqrt{n}

- b) Écrire une fonction `bigint(k)` qui génère, en utilisant le module `random`, un entier aléatoire compris entre 2^k et $2^{k+1} - 1$.

- c) Écrire une fonction `generate(k, isprime)` qui génère un nombre premier aléatoire compris entre 2^k et $2^{k+1} - 1$, en utilisant la question précédente.

L’argument `isprime` désigne bien une fonction qui sera passée en argument. Cette fonction pourra être, dans un premier temps, la fonction `isprime` codée plus haut.

- d) La **méthode de Fermat** consiste à tester (au sens statistique) si un entier n est premier. Elle consiste à tester si l’on a bien $a^{n-1} \equiv 1$ modulo n , pour un nombre N suffisamment grand de a pris uniformément aléatoire entre 1 et $n - 1$. Si les N équivalences sont vérifiées, on estime alors que n est premier.

Coder le test de Fermat. On prendra $N = 128$.

Pour calculer des puissances de grands nombres modulo un entier, il est judicieux d’utiliser la fonction `pow`.

- e) Utiliser les questions précédentes pour générer un nombre (pseudo-)premier sur 1024 bits.

Exercice 7 : algèbre linéaire. On considère une liste de liste comme une matrice, par exemple la matrice de rotation 2×2 d'angle θ sera représentée par

```
M = [[cos(theta), -sin(theta)],  
      [sin(theta), cos(theta)]]
```

et une liste simple sera interprétée comme un vecteur ligne. On va d'abord chercher à vérifier si nos objets sont interprétables comme des matrices ou vecteurs.

- Définir une exception `PasUneMatrice`.
- Écrire une fonction `dimensions(matrice)` qui renverra les dimensions de la matrice sous forme d'un tuple (`nb_lignes`, `nb_colonnes`). Si l'argument n'est pas interprétable comme un vecteur ou une matrice (par exemple si elle a des lignes de longueurs variables), la fonction soulèvera l'exception `PasUneMatrice`.
- Écrire une fonction `transpose(matrice)` qui transpose une matrice.
- Écrire une fonction `produit(mat1, mat2)` qui effectue le produit matriciel.

On n'a pas cherché à optimiser les choses, et les codes produits seront remplacés (plus tard, on a le temps) avantageusement par ceux de la librairie Numpy, qui gère l'algèbre linéaire en Python de manière infiniment plus efficace avec ces `ndarray` !

Exercice 8 : Considérons les données :

```
x = [7.63, 4.95, 0.38, 6.78, 7.25, 4.47, 2.73, 5.21, 2.75, 7.21]  
y = [16.27, 9.25, -2.0, 13.98, 18.14, 8.2, 0.75, 12.05, 5.27, 18.37]
```

- Écrire des fonctions `mean(x)`, `var(x)`, `covar(x,y)` qui calculent des moyennes, variances et covariances empiriques.
- Calculer les estimateurs des moindres carrés ordinaires $\hat{\beta}_0$ et $\hat{\beta}_1$ pour le modèle linéaire

$$y = \beta_0 + \beta_1 x + \varepsilon.$$

- Vérifier que la prédiction pour y en $x_1 = 6.5$ est de 14.52.

Exercice 9 :

- Aller sur le site du [projet Gutenberg](#), télécharger votre texte favori au format "Plain Text (UTF-8)" et le placer dans le même dossier que le script.
- Écrire une fonction `get_texte(fichier, n)` qui récupère les n premières lignes du fichier donné en argument, et renvoie leur concaténation en une longue chaîne de caractère, "nettoyée" par la fonction suivante :

```
import re  
def nettoie(texte):  
    # Convertir en minuscules.  
    texte = texte.lower()  
    # Séparer les apostrophes des mots suivants.  
    texte = re.sub(r'\s*([\'])\s*', r'\1 ', texte)
```

```
# Remplacer les sauts de lignes par des espaces.
texte = re.sub(r'\n', ' ', texte)
# Supprimer la ponctuation et renvoyer.
return re.sub(r'[^a-zA-Z \-\'èéââûûçîôüëïöä]', '', texte)
```

Attention à l'encodage du fichier : si vous utilisez la fonction `open()` pour ouvrir le fichier avec Python, il faudra sans doute utiliser l'argument optionnel `encoding='utf8'`.

- Écrire une fonction `occurrences_mots(texte)` qui, étant donné une chaîne de caractère `texte`, renvoie un dictionnaire dont les clés sont les mots du texte, et dont les valeurs donnent le nombre d'occurrences de chaque mot.
- Se fixer n le plus grand possible qui reste tel que l'exécution de la ligne `occurrences_mots(get_texte('votre_fichier.txt', n))` ne prenne pas trop de temps.
- Afficher les 100 mots les plus utilisés dans l'extrait choisi du texte téléchargé, et leur nombre d'occurrence.

Exercice 10 : On s'intéresse ici aux permutations de $[[0, n - 1]]$. On les modélise d'abord comme des tuples $\sigma = (\sigma(0), \sigma(2), \dots, \sigma(n - 1))$.

- Écrire une fonction `composition(sigma, tau)` qui renvoie $\sigma \circ \tau$.
- Écrire une fonction `inverse(sigma)` qui calcule σ^{-1} .
- Écrire une fonction `decomp_cycles(sigma)` qui effectue la décomposition en cycles de σ . Par exemple, avec `sigma=(1, 6, 2, 4, 5, 3, 0)`, on obtient la décomposition en cycles `[(0, 1, 6), (3, 4, 5)]` (2 étant un point fixe de la permutation, on l'omet de la décomposition).
- Écrire une fonction `reconstruit(cycles)` qui donne la forme "classique" de la permutation à partir de sa décomposition en cycles.
- On peut générer une permutation uniforme de $[[0, n - 1]]$ grâce au code suivant :

```
import numpy as np
def perm(n):
    return tuple(np.random.permutation(n))
```

Sur la base de simulations, établir un intervalle de confiance pour la probabilité qu'une grande permutation uniforme ne contienne pas de point fixe. Comparer avec la valeur asymptotique (quand $n \rightarrow \infty$) de cette probabilité : $p = e^{-1}$.

Exercice 11 : méthode de Newton. Si f est une fonction dérivable, strictement croissante et telle qu'elle s'annule en un certain $x \in \mathbb{R}$, on utilise la méthode de Newton–Raphson pour trouver x . On pose x_0 quelconque et l'on définit par récurrence

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Sous de bonnes hypothèses, on a alors $x_n \rightarrow x$.

- Écrire une fonction `newton(f, df, eps=1e-10, N=1000)` qui prend en argument deux fonctions `f` et `df`, et qui applique l'algorithme de Newton–Raphson. Plus précisément, la fonction :
 - utilisera l'argument `f` comme la fonction f , et `df` comme la fonction f' .
 - partira d'un point x_0 aléatoire entre 0 et 1, obtenu en utilisant `random.random()`, après avoir importé le module `random`.

- s'arrêtera soit au bout d'un nombre N itérations, en soulevant une exception définie par vous-même, soit si l'on calcule que $|x_n - x_{n+1}|$ est inférieur au seuil eps , auquel cas la fonction renverra x_{n+1} .

Exemple d'utilisation :

```
import math
def f(x):
    return math.exp(x) - 3
def df(x):
    return math.exp(x)
print('log(3) =', newton(f, df)) # affiche 'log(3) = 1.09861229'

# Ou encore, en utilisant des lambda-fonctions:
print('log(3) =', newton((lambda x: math.exp(x)-3), (lambda x: math.exp(x))))
```

- Calculer numériquement le point $x \in \mathbb{R}_+$ satisfaisant $\arctan(x) = 1 - \log(x)$.
- Calculer numériquement le minimum de la fonction $x \mapsto \cosh(x - 3) + x^2$.

Exercice 12 : Étant donné une réalisation de trajectoire d'une chaîne de Markov (x_0, x_1, \dots, x_n) , l'on peut estimer les probabilités de transition $p_{i \rightarrow j}$ de la chaîne par la proportion de transition de type $i \rightarrow j$ observées à partir de l'état i :

$$\hat{p}_{i \rightarrow j} = \frac{N_{i \rightarrow j}}{N_i}, \quad \text{avec } N_{i \rightarrow j} = \sum_{k=0}^{n-1} \mathbb{1}_{x_k=i, x_{k+1}=j} \quad \text{et } N_i = \sum_{k=0}^{n-1} \mathbb{1}_{x_k=i}.$$

- Écrire une fonction `transitions(traj)` qui prend en argument une trajectoire finie de chaîne de Markov sous forme de liste `traj`, puis renvoie un dictionnaire de forme $\{i : \{j : N_{ij}, \dots\}, \dots\}$. Par exemple pour la trajectoire `traj = ['A', 'B', 'A', 'C', 'B', 'A', 'D']`, la fonction renverra le dictionnaire

```
{'A': {'B': 1, 'C': 1, 'D': 1},
 'B': {'A': 2},
 'C': {'B': 1},
 'D': {}}
```

- Écrire une fonction `normalise(dico)` qui, partant d'un dictionnaire `dico` dont toutes les valeurs sont des nombres positifs, renvoie un dictionnaire "de Markov", c'est-à-dire dont la somme des valeurs somme à 1.
- Utiliser la question précédente pour écrire une fonction `norm_transitions(trans)` qui prend en argument un dictionnaire `trans` comme renvoyé par la fonction `transitions(...)`, et renvoie un dictionnaire de dictionnaires dont toutes les valeurs sont normalisées.

En reprenant l'exemple ci-dessus :

```
{'A': {'B': 0.33, 'C': 0.33, 'D': 0.33},
 'B': {'A': 1.0},
 'C': {'B': 1.0},
 'D': {'D': 1.0}}
```

- Écrire une fonction `chaîne(trans, x0, n)` qui prend en entrée un "dictionnaire de Markov", un état initial x_0 , une longueur n , et renvoie une trajectoire de la chaîne de Markov approximée.

- e) Comme dans l'exercice 9, télécharger son texte préféré sur le site du [projet Gutenberg](#), et après nettoyage du texte, le récupérer sous forme de liste de mots (pour python, des chaînes séparées par des espaces).

Traiter la liste obtenue comme une réalisation d'une chaîne de Markov, et générer des phrases aléatoires grâce aux fonctions écrites ci-dessus.