Algorithmique

Numérique & Aléatoire

Annexes

Pourquoi Python ? Le langage de programmation Python a été créé en 1989 par Guido van Rossum, aux Pays-Bas. Le nom Python vient d'un hommage à la série télévisée Monty Python's Flying Circus dont G. van Rossum est fan, rien à voir avec le serpent. La première version publique de ce langage a été publiée en 1991.

La dernière version majeure de Python est la version 3. La version 2 de Python est désormais obsolète, dans la mesure du possible évitez de l'utiliser.

La Python Software Foundation 1 est l'association qui organise le développement de Python et anime la communauté de développeurs et d'utilisateurs.

Ce langage de programmation présente de nombreuses caractéristiques intéressantes :

- il est multiplateforme. C'est-à-dire qu'il fonctionne sur de nombreux systèmes d'exploitation.
- Il est entièrement gratuit.
- C'est un langage de haut niveau. Il demande relativement peu de connaissance sur le fonctionnement d'un ordinateur pour être utilisé.
- C'est un langage interprété. Un script Python n'a pas besoin d'être compilé pour être exécuté, contrairement à des langages comme le C ou le C++.
- Il est orienté objet. C'est-à-dire qu'il est possible de concevoir en Python des entités qui miment celles du monde réel (une cellule, une protéine, un atome, etc.) avec un certain nombre de règles de fonctionnement et d'interactions. Nous ne développerons pas ce point, conformément au programme de BCPST.
- Il est relativement simple à prendre en main.
- Enfin, il est très utilisé en bioinformatique[1] et plus généralement en analyse de données.

Toutes ces caractéristiques font que Python est désormais enseigné dans de nombreuses formations, depuis l'enseignement secondaire jusqu'à l'enseignement supérieur.

Comment l'installer ? On recommande le duo suivant :

1. Installation de la distribution Anaconda (ou Miniconda pour une version allégée).
2. Installation de *Pyzo* en tant qu'IDE (environnement de développement) ou alors utiliser *Spyder* fourni directement avec Anaconda.

Toutes les instructions sont détaillées ici (y compris pour mettre à jour la distribution de temps en temps) :

`https://jonathanharter.legtux.org/bcpst1/installations/`

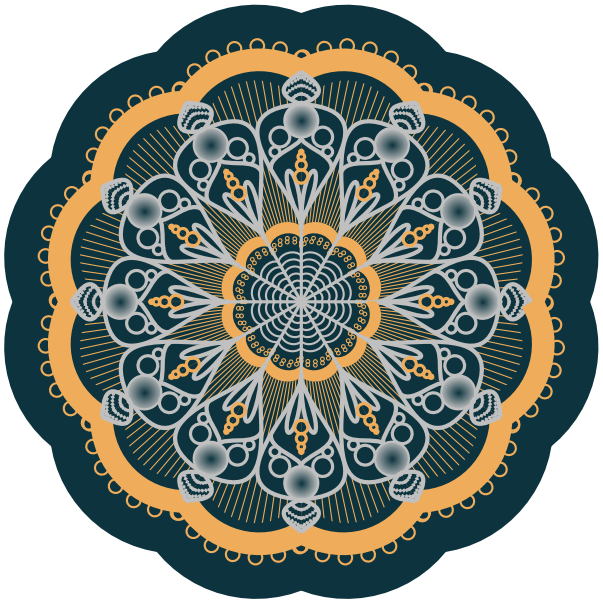
Objectifs. Conformément au programme, nous aborderons l'apprentissage de l'Informatique avec plusieurs objectifs :

- la connaissance d'un langage de programmation et d'un environnement de développement (`Python` et *Pyzo* ou *Spyder* ici),
- la compréhension et la maîtrise de différents algorithmes (recherche dans une liste, tri d'un tableau de nombres, parcours de graphes ...),
- l'utilisation autonome de méthodes numériques (calcul approché d'une intégrale, simulation d'une variable aléatoire, résolution d'un système linéaire ...) en lien avec le cours de Mathématiques.

O. Carcone, J. Harter

[1] La science dont le but est le traitement informatique de phénomènes biologiques

Version du 11 septembre 2022



I

Première partie

Algorithmique

Chapitre ALGO1.

Fondamentaux

Résumé & Plan

L'objectif de ce chapitre est de redécouvrir les principales structures en Python : les boucles, les tests, et de savoir les manipuler. Nous travaillerons pour le moment uniquement sur des objets de type `int` (des entiers), `float` (flottants, des nombres à virgule). Les autres types seront traités dans de prochains chapitres.

Si debugger, c'est supprimer des bugs, alors programmer ne peut être que les ajouter.

— Edsger Dijkstra

1. INTRODUCTION À PYTHON

1.1. Environnement de travail

Le réseau du lycée. Vous disposez d'un accès au réseau pédagogique du Lycée Montaigne qui vous fournit un espace disque accessible depuis n'importe quel ordinateur du lycée.

- Saisissez votre identifiant et votre mot de passe personnels pour vous connecter au réseau.

- Naviguez dans l'arborescence réseau et identifiez votre dossier personnel (dossier de travail). Vos fichiers doivent être enregistrés à cet emplacement, en créant au besoin des sous-dossiers (vous pouvez d'ors et déjà créer un dossier « Informatique », puis un sous-dossier TP1).
- Identifiez également le dossier partagé, qui est accessibles à tous les élèves de la classe ainsi qu'aux enseignants. Vous y trouverez pour certaines séances des fichiers déposés pour certains TPs.

L'IDE *Pyzo*. L'environnement de développement *Pyzo* est choisi pour sa simplicité de mise en oeuvre et sa licence open source. Il est constitué :

- d'un **éditeur** (fenêtre de gauche par défaut) qui permet de saisir le programme (les mots-clé du langage sont colorés, ce qui permet une relecture facile, et l'indentation est automatique),
- d'une **console** (fenêtre de droite par défaut), qui permet d'exécuter des instructions en ligne de commande (en tapant à la suite de `>>>` et de suivre le déroulement de son programme. Pour vous familiariser avec la console, taper par exemple les commandes `1+1`, `x = 3` puis `x+1`.
- D'un **explorateur de variables** (menu « Workspace »), qui permet de connaître les valeurs contenues dans les variables en cours d'utilisation. Cet fenêtre est généralement moins utile sauf pour les longs codes.
- La **structure du code** liste les différentes fonctions ainsi que leur dépendance.

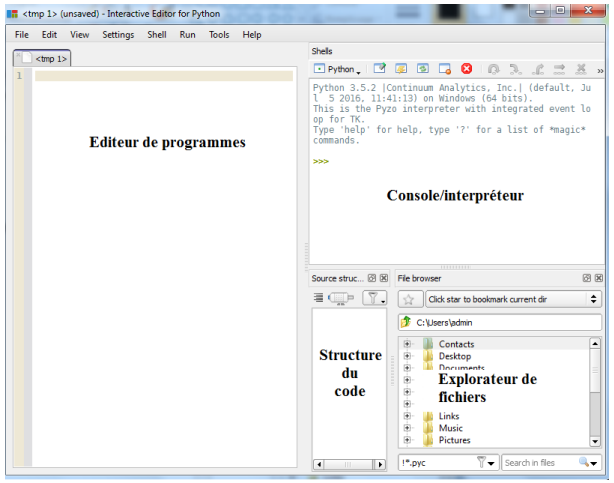
Afin de prolonger le travail fait en classe, il est recommandé que vous l'installiez sur votre ordinateur personnel à partir du site `http://www.pyzo.org`.

Attention Différence entre éditeur et console

Il faut enregistrer son travail sous la forme de fichiers `.py` de façon régulière, dans un répertoire créé à cet effet (par exemple `TP1`) et on exécutera le programme en cours à l'aide de la commande « démarrer comme un script » ou de la touche F5. Une différence importante est que l'ensemble des instructions du fichier sera exécuté en une seule fois, contrairement à la console.

Attention

Les noms de fichier ne doivent ni contenir d'espaces, ni d'accents.



En `Python`, toute ligne commençant par un `##` est un commentaire : son contenu sera ignoré lors de l'exécution du programme. Les commentaires servent donc à améliorer la compréhension du code, mais il ne faut pas non plus en abuser. Par exemple écrire `## augment x de 1` à côté de l'instruction `x += 1` est complètement inutile.

De plus, l'indentation (décalage du début de ligne pour aligner verticalement les instructions) est non seulement essentielle pour relire son programme (quelles sont les instructions exécutées après un `if` ou dans une boucle `for` ?) mais aussi obligatoire pour le bon fonctionnement du programme. L'indentation ne se fait pas n'importe comment : on utilise la touche de tabulation ⇥ du clavier.

Dans un premier temps, nous allons travailler uniquement dans la console.

Résumé Privilégier l'éditeur

Il y a trois raisons pour lesquelles il faut travailler dans l'éditeur :
- On peut écrire plusieurs instructions à la suite.
- On peut sauvegarder son travail et donc le retrouver au prochain démarrage de pyzo.
- On peut y écrire des commentaires afin de clarifier son code.

1.2. Console

Il suffit pour exécuter une instruction saisie dans la console d'appuyer sur la touche ⏎. Exécutez les instructions suivantes et commentez les différents résultats sur la feuille du TP. La console peut donc aussi servir de calculatrice, afin d'afficher des résultats «à la volée».

- `1.2`,
- `1,2`,
- `6`,
- `6.`,
- `2+3`,
- `2+3.0`,
- `2-3`,
- `3*2`,
- `3**2`,
- `10/3`,
- `10//3`,
- `10%3`,
- `1 == 2`,
- `1 == 1.0`,
- `1 == 1`,
- `"Bon"`,
- `"Bon"+"jour"`,
- `3*"Boum!"`.

1.3. Types

Panorama des différents types. En Python, les objets manipulés ont un type et le typage est automatique. C'est-à-dire que lorsque vous créer un objet en Python (par exemple *via* les commandes précédentes exécutées dans la console), Python lui affecte un type de manière automatique. Pour connaître le type d'une expression, on peut utiliser la commande type. Par exemple :

```
>>> type(12) # entier
<class 'int'>
>>> type(12.0) # flottant
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type("douze")
<class 'str'>
>>> type("12") # chaîne de caractère
<class 'str'>
>>> type(["d", "o", "u", "z", "e"]) # liste
<class 'list'>
>>> type({"1" : "2"}) # dictionnaire
<class 'dict'>
```

Fixons un peu de vocabulaire.

- Un objet informatique est dit *immuable*, si son état ne peut pas être modifié après sa création. À l'inverse il est dit *variable*.
- On appelle *méthode* sur un objet `O` (liste, tuple, entiers, *etc.*), toute instruction du type
  `O.nom_methode()`.
  Par exemple, l'instruction `.append(x)` rajoute `x` dans `L` est une méthode sur les listes. Nous la verrons dans le Chapter ALGO2.
- Il faut distinguer les méthodes des fonctions. Par exemple, `sum` est une fonction Python qui retourne la somme des éléments d'une liste (la notion de fonction sera étudiée dans la prochaine section).

Dans ce TP, nous n'utiliserons que les types ci-après :

- le type `int` : les entiers relatifs.
- Le type `float` : les nombres flottants.
- Le type `str` : les chaînes de caractères («string» en Anglais).
- Le type `bool` : les booléens `True`, `False`.

Au cours de l'année, nous rencontrerons en revanche l'ensemble des types ci-après.

Type	Exemple	Quelques opérations	Mutable ?
Entier `int`	`42`	`+ , - , *, **`	Oui
Flottant `float`	`0.3`	`+ , - , *, ** , /`	Oui
Booléen `bool`	`1 == 2`	`and, or, not`	Oui
Chaînes `str`	`"blablabla"`	`+ , len , in`	Non
Liste `list`	`[0,1,2]`	`+ , len , in`	Oui
Tableaux `numpy`	`np.array([0, 1, 2])`	`+ , len , in`	Oui

Tuple `tuple`	`(0,1,2)`	`+ , len , in`	Non
Dictionnaire `dict`	`{'clef': 'val', ...}`	`.Keys, .Values`	Oui

Types numériques. En Python, les nombres peuvent avoir deux types, `int` ou `float`, suivant qu'il s'agit d'un entier relatif ou d'un nombre flottant. On dispose de plus des opérations suivantes :

Commande	Effet
`+`	Additionne deux quantités. Si les deux sont des entiers, le résultat est un entier. Sinon un flottant.
`-`	Même chose, mais pour la soustraction.
`*`	Multiplie deux quantités. Si les deux sont des entiers, le résultat est un entier. Sinon un flottant.
`/`	Divise deux quantités. Le résultat est **toujours** un flottant. Par exemple, `2/2` retourne `1.0`.
`**`	Élève à la puissance deux quantités. Si les deux sont des entiers, le résultat est un entier. Sinon un flottant.

Dans le cas des entiers (type `int`), on possède deux commandes supplémentaires pour l'arithmétique.

Arithmétique

```
>>> 5//2 # quotient de la division euclidienne
2
>>> 5%2 # reste de la division euclidienne
1
```

Attention aux flottants!

En Python, les entiers ont une taille arbitraire, limitée seulement par les capacités de la machine, les calculs se font donc en **valeurs exactes**.

```
>>> 2**100
1267650600228229401496703205376
>>> (10**100 +1) - 10**100
1
```

Les flottants en revanche ont un nombre de décimales limité[1] et il peut y avoir des erreurs d'arrondi, les calculs se font donc en **valeurs approchées**.

```
>>> 2.0**100
1.2676506002282294e+30
>>> (10**100 +1.)-10**100 # curieux, non ?
0.0
```

Attention Les fonctions numériques usuelles (racine carrée, logarithme, etc...) ne sont pas disponibles de base. Il faudra pour les utiliser importer des bibliothèques de calcul numérique comme `math` ou `numpy`.

Type booléen & Tests logiques. Ce type est adapté aux tests logiques (nous en reparlerons longuement quand nous ferons les tests `if` plus tard dans ce TP). Les booléens sont `True` et `False`.

Attention

On n'écrit **pas** `"True"` et `"False"` qui sont des chaînes et non des booléens.

```
>>> type("True")
<class 'str'>
>>> type(True)
<class 'bool'>
```

Nous avons, comme dans le cours de Mathématiques, des opérations «ou» et «et» sur les booléens.

[1] Le concept «d'infini» est incompatible avec la notion même d'ordinateur. Par exemple, $\frac{1}{3}$ possède un nombre infini de 3 après la virgule, Python est incapable de tous les prendre en compte donc tronquera la série de 3

Opérateur « et »

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

Opérateur « or »

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

On dispose aussi d'un opérateur **not** pour obtenir la négation d'une proposition logique.

Négation

```
>>> not True
False
```

```
>>> not True
False
```

Comment obtient-on concrètement un booléen ? Par un test logique d'égalité. La syntaxe est la suivante `variable_1 == variable_2`, le résultat est alors un booléen.

Exemple 1 (Tests logiques) Prédire et observer les résultats des tests logiques ci-après.

```
>>> 1 == 2
False
>>> 1 == 1.0
True
>>> 1 == 1**2
True
>>> "abc" == "a bc"
False
>>> int(1.0) == 0+1
True
>>> int(1.0) == 0+1.0
True
>>> 1 == 1/1
True
>>> 2.0 == 1.0 + 1.0
True
```

Attention

Comme en témoigne l'exemple précédent, il faut donc rester **très méfiant** à propos des test `==`. Par exemple, le résultat de `1 == 1.0` n'est pas du tout intuitif, Python opère implicitement à une conversion en flottant de l'entier `1` avant de réaliser le test. En cas de doute, mieux vaut effectuer quelques tests rapides dans la console avant.

CONVERSION DE TYPES. Il est possible de convertir, par exemple, une expression de type `int` en `float`.

```
>>> a = 1
>>> type(a)
<class 'int'>
>>> float(a)
1.0
```

```
>>> a = float(a) # conversion en flottant
>>> type(a)
<class 'float'>
>>> a = int(a) # retour aux entiers
```

Il existe beaucoup d'autres possibilités de conversion, toutes sont intuitives! Par exemple :

```
>>> int(True)
1
>>> list(str(100))
['1', '0', '0']
```

1.4. Variables

Pour stocker un résultat ou une valeur, on crée des objets en Python que l'on affecte à un nom que l'on appelle *variable*. La syntaxe d'une déclaration de variable est la suivante :

Déclaration d'une variable

```
>>> x = 1 # création de la variable x puis affectation de l'objet
↪  int 1
```

De manière générale, on écrit : `nomdelavariable = expression`. On peut aussi affecter à la chaîne plusieurs variables.

```
>>> a, b, c = 1, 2, "coucou"
>>> a
1
>>> b
2
>>> c
'coucou'
```

Le symbole «=» doit être compris comme «reçoit la valeur», on noterait cela mathématiquement $x \longleftarrow \ldots$ (comme lorsque l'on remplace une ligne par une autre dans un système linéaire par exemple), et non comme une égalité classique.

ÉCHANGER LES VALEURS DE DEUX VARIABLES. On suppose crées deux objets `a`, `b` et on souhaiterait échanger leur valeur. Une idée naïve serait la suivante.

Exemple 2 (Échange naïf)

```
>>> a = 1
>>> b = 2.0
>>> b = a
>>> a = b
>>> a
1
>>> b
1
```

Analyser les valeurs de `a`, `b`. L'échange a-t-il été fait correctement? Pourquoi?

La bonne façon de voir les variables c'est comme des tasses que l'on remplit à l'aide de certains objets (entiers, flottants *etc.*). La question est donc : comment échanger le contenu de chaque tasse?

Exemple 3 (Échange naïf) Compléter la version ci-après pour que l'échange soit correct.

```
a = 1
b = 2.0
...
...
...
```

On peut même utiliser une commande toute faite en Python pour faire cela :

Résumé Échanger les valeurs de deux variables

```
a, b = b, a
```

Incrémenter des variables entières ou des flottants . Il existe des raccourcis classiques pour ajouter ou multiplier ce type de variable. Voyez sur l'exemple qui suit.

Privilégier

```
>>> n = 1
>>> n += 1
>>> n
2
>>> n *= 2
>>> n
4
```

Éviter

```
>>> n = 1
>>> n = n+1
>>> n
2
>>> n = 2*n

>>> n
4
```

Accès aux objets définis. On peut par ailleurs accéder à l'ensemble des variables déclarées *via* la commande `globals()`, elles sont également à retrouver dans la fenêtre « explorateur d'objets » de Pyzo.

1.5. Importation de modules

Des modules supplémentaires ont été crées pour le traitement spécifique de certaines tâches. Nous pouvons citer les principaux que nous utiliserons :

- `math` : qui comme son nom l'indique, est dédié aux Mathématiques (toutes les fonctions usuelles principales par exemple).
- `numpy` : qui est dédié à tout le calcul numérique (résolution de systèmes linéaires, opérations diverses sur les matrices *etc.*). Nous utiliserons largement ce module dans le Chapter NUM7.
- `matplotlib` : qui est dédié à l'affichage de graphiques divers.
- dans une moindre mesure vous utiliserez le module `scipy` pour les statistiques de deuxième année par exemple. Nous utiliserons largement ce module dans le Chapter NUM8.

Afin d'utiliser une bibliothèque, on commence par l'importer. Pour cela, on procède de la manière suivante.

```
import nom_module as prefixe_a_choisir
```

Pour les modules cités précédemment nous ferons toujours :

```
import math as ma
import numpy as np
import matplotlib.pyplot as plt # sous-module pyplot de matplotlib
```

Pour accéder à une fonction on utilisera

```
prefixe_a_choisir.nom_de_la_fonction
```

Par exemple :

```
>>> ma.cos(0)
1.0
```

Attention **Méthodes alternatives d'importation à éviter**

On peut aussi importer les modules sans préfixe, c'est-à-dire que lorsque nous ferons appel aux fonctions dudit module, on peut éviter de taper le nom du préfixe. Par exemple,

```
import math
cos(0)
```

Mais cela est dangereux : en effet, les modules `math` et `numpy` possèdent par exemple tous les deux des fonctions `cos`, `sin` *etc.*. Donc en important `math` puis `numpy`, on «écrase» les fonctions `math` par celles de `numpy` lors de la seconde affectation. Utiliser un préfixe permet donc d'avoir une garantie sur quelle fonction on utilise quand y fait appel.[2]

2. FONCTIONS & PROCÉDURES

Nous créerons la plupart du temps des «groupes» d'instructions Python que l'on appelle «fonctions» ou «procédures».

- On appellera *fonction* toute série d'instructions informatiques retournant un résultat (présence d'un `return` dans la suite).
- On appellera *procédure* toute série d'instructions informatiques ne retournant pas de résultat (par exemple, une modification des variables données en argument).

Attention

Un comportement analogue aux fonctions, que l'on trouve parfois dans littérature, consiste à utiliser des commandes du type `input` / `print`, mais nous nous l'interdirons.

Jusque maintenant, nous avons essentiellement travaillé dans la console. À présent, on part plutôt du côté de l'éditeur (fenêtre de gauche).

[2]Et il existe des différences entre le `cos` de `math` et le `cos` de `numpy` par exemple, nous y reviendrons.

2.1. Généralités

Dans le langage Python, une fonction est une suite d'instructions dépendant de paramètres. Rappelons la syntaxe pour la définir en langage Python.

Structure d'une fonction

```
def f(x1, x2, ..., xn):
    """
    Documentation (docstring)
    """
    # instructions créant y1,
    ↪  ..., yp
    return y1, y2, ... , yp
```

Structure d'une procédure

```
def f(x1, x2, ..., xn):
    """
    Documentation (docstring)
    """
    # instructions utilisant
    ↪  x1, ..., xn
```

Les paramètres `x1, x2, ..., xn` sont ici obligatoires, il existe aussi des paramètres dits optionnels (*i.e.* possédant une valeur par défaut), mais nous ne les utiliserons pas. La seule différence entre les deux structures de code est que l'une ne renvoie rien (procédure).

```
f(val1, val2, ..., valn) # exécution de fonction
help(f) # affichage de la documentation
res_1, ..., res_p = f(val1, val2, ..., valn) # stockage du
↪  résultat de la fonction dans des variables
```

Une fonction est dite *itérative* si elle ne fait pas appel à elle-même (généralement construite à partir de structures simples comme `if`, `while`, `for` *etc.*). Elle est dite *récursive* dans le cas contraire, nous ne travaillerons pour le moment pas avec des fonctions récursives, cela sera fait dans le Chapter ALGO4.

Exemple 4 Observons cet exemple qui aide à mieux comprendre la gestion des paramètres d'une fonction et de ce qu'elle retourne.

```
def f(x):
    return x+1

def g(x):
    x = 3
```

On peut alors envisager différents types d'exécution.

```
>>> x = 1.0
>>> f(x) # retourne un résultat
2.0
>>> x    # variable x NON modifiée
1.0
>>> g(x) # ne renvoie rien
>>> x     # mais x a été modifiée
1.0
```

Enfin, notons qu'il est bien sûr possible d'utiliser dans le corps d'une fonction une autre fonction définie au préalable. Par exemple l'exécution du programme suivant entré dans l'éditeur :

```
def f(x):
    return x+1

def g(x):
    return x**2

def h(x):
    return g(f(x))

>>> h(2)
9
```

Variables locales et globales. Toute grandeur définie à l'intérieur d'une fonction ne peut être utilisée à l'extérieur, sauf exception. Les variables déclarées à l'intérieur d'une fonction sont donc appelées *variables locales*.[3]

Cependant il est aussi possible de décréter une variable comme étant «globale» et que l'on pourra utiliser ensuite à l'extérieur.

Exemple 5 Dans ce premier exemple, la variable a est locale.

```
def g(x):
    a = x**2
    return a
```

Mais dans le second,

```
def h(x):
    global a
    a = x**2
    return a
```

cette fois-ci, la variable a est bien définie même en dehors de la fonction. On parle alors de *variable globale*. Une fois la fonction h exécutée, un appel à a dans la console donnera la valeur de a.

Les fonctions `print` et `input`. Deux fonctions importantes de Python — présente nativement sans importation de module — sont `print` et `input` et qui s'utilisent comme suit :

- `print(x)` : prend en argument une variable x ou un objet, et affiche l'objet. Cette fonction sert la plupart du temps à des fins de débogage, en faisant afficher des quantités interessantes au milieu d'un algorithme qui renverrait une erreur.
- `x = input("message d'information")` : prend en argument une chaîne qui est un message d'information, et stocke dans x la valeur entrée. Cette fonction sera très rarement utilisée, elle est seulement faite pour interagir avec un utilisateur (par exemple si vous programmez un jeu et que l'utilisateur doit faire un choix).

[3] Sur le même principe que les variables de sommes ou d'intégrales en Mathématiques.

Mais, en TP, la plupart du temps l'utilisateur ce sera vous-même donc on privilégiera l'usage de fonctions (voir plus bas).

On peut également afficher plusieurs objets à la suite qui doivent être séparés par des virgules.

```
>>> print("2+3=5")
2+3=5
>>> print(2+3, "=", 5)
5 = 5
>>> print("2+3", "=", 5)
2+3 = 5
```

Attention `print` ou pas `print` dans l'éditeur ?

En exécutant le code de l'éditeur, les résultats sont affichés dans la console, mais seulement ceux des instructions dont l'affichage est explicitement demandé par la commande `print`. Ainsi, si l'éditeur est

```
x = 1
```

vous ne verrez aucun affichage (mais la variable x sera créee). En revanche, un affichage aura lieu si l'éditeur contient :

```
x = 1
print(x)
```

En revanche, dans la console, tout est affiché par défaut.

L'INSTRUCTION `lambda` : DÉFINIR EN « MODE *INLINE* » DES FONCTIONS Pour définir des fonctions d'une variable réelle, une syntaxe relativement commode est la suivante

```
>>> g = lambda x: x**2
>>> g(2)
4
>>> def g_prim(x):
...     return x**2
...
>>> g_prim(2)
4
```

Les deux commandes définissent dans le cas présent la fonction $x \longmapsto x^2$.

Vous voyez ici que le mot **`lambda`** est reconnu par Python. Il est donc interdit d'utiliser **`lambda`** en nom de variable.[4]

2.2. Fonctions de modules

Python dispose de plusieurs bibliothèques de fonctions destinées à un usage spécifique. Il convient de les charger grâce à la commande **`import`** avant de pouvoir les utiliser. Détaillons celles qui nous seront les plus utiles pendant l'année. Profitez-en également pour revoir, à l'aide de votre cours de première année, la manière d'importer des modules en Python (voir cours de première année).

LE MODULE `math` Prenons l'exemple de la bibliothèque `math` : elle contient notamment les fonctions suivantes.

Voir toutes les fonctions disponibles dans un module

```
>>> import math
>>> dir(math)
```

[4] Privilégier alors `lamba` par exemple

```
['__doc__', '__file__', '__loader__', '__name__', '__package__',
 ↪ '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
 ↪ 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees',
 ↪ 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
 ↪ 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd',
 ↪ 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan',
 ↪ 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
 ↪ 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow',
 ↪ 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan',
 ↪ 'tanh', 'tau', 'trunc', 'ulp']
```

On retiendra que la fonction `factorial` peut désormais être utilisée. Les fonctions usuelles peuvent aussi être importées depuis la bibliothèque `numpy` : ces dernières sont plus adaptées aux calculs numériques matriciels, par exemple pour les tracés de courbe que nous reverrons plus tard.

Les modules `numpy`, `scipy`, `random` et `matplotlib` Les bibliothèques `numpy` et `scipy` fournissent des outils pour le calcul scientifique. Nous utiliserons surtout `random` (simulations), `numpy` (calculs numériques), `matplotlib` (graphiques), plus rarement `scipy` (en Statistiques essentiellement). Tout autre module n'est pas un attendu du programme.

2.3. Chronométrage

Il est important en Informatique de pouvoir quantifier l'efficacité d'un programme plutôt qu'un autre, cela peut se faire à l'aide du module `time`. Voyons un exemple d'utilisation.

Chronométrage naïf

```
>>> import time as ti
>>> t_1 = ti.time()
>>> x = 3
>>> y = x**4
>>> format(ti.time() - t_1, "10.2E") # temps mis pour l'exécution
 ↪ en notation scientifique
'  1.00E-04'
```

Remarque 1 (Complexité) Il est également possible de quantifier l'efficacité de manière théorique en comptant le nombre d'opérations, on parle alors de *complexité temporelle*. Cette notion est hors-programme en BCPST.

Pour être plus robuste, mieux vaut réaliser ce chronométrage un certain nombre de fois et retourner la moyenne.

Chronométrage robuste

```
>>> import time as ti
>>> nb_ex = 1000 # nb d exécutions
>>>
>>> temps_moy = 0
>>> for _ in range(nb_ex):
...     t_1 = ti.time()
...     x = 3
...     y = x**4
...     temps_moy += ti.time() - t_1
...
>>> format(temps_moy/nb_ex, "10.2E") #affichage du temps moyen en
 ↪ notation scientifique
'  2.81E-07'
```

3. TESTS LOGIQUES & BOUCLES

On présente dans cette section les principales structures en Python.

3.1. Tests logiques

Test if simple

```
if test:
    instructions
else:
    instructions
```

Test if avec plusieurs conditions

```
if test:
    instructions
elif test:
    instructions
elif condition:
    instructions
```

Remarque 2 S'il y a plus que deux cas nécessitant un traitement différencié, on peut utiliser l'instruction `elif` (contraction de else et if). L'instruction `else` finale sera traitée seulement si les cas précédents n'ont pas été rencontrés.

Exercice 1 | Fonctions mathématiques définies par morceaux Solution Coder en Python les deux fonctions mathématiques ci-après :

$$f_1 : x \mapsto \begin{cases} -x & \text{si } x \leqslant -1, \\ x^2 & \text{si } x > -1 \end{cases}, \quad f_2 : x \mapsto \begin{cases} -x & \text{si } x \leqslant -1, \\ x^2 & \text{si } -1 < x < 2 \\ x+2 & \text{si } x \geqslant 2. \end{cases}$$

3.2. Boucles

3.2.1. Inconditionnelle : boucle for

Ce sont les boucles dont on connait à l'avance le nombre d'étapes nécessaires pour arriver au résultat.

Boucle for

```
for k in string/list/etc:
    instructions
```

Lorsque le corps de boucle de dépend d'aucun indice, il s'agit alors d'une répétition d'actions, on peut alors écrire.

```
for _ in string/list/etc:
    instructions
    # répétition des instructions
```

La plupart du temps, nous utilisons l'itérateur `range` pour faire avancer l'indice d'une boucle `for`. Le mieux est de comprendre cela sur un exemple.

Exemple 6

```
>>> for i in range(3):
...     print(i)
...
0
1
2
>>> for i in range(1, 3):
...     print(i)
...
1
2
>>> for i in range(3, 1):
...     print(i)
...
```

Commentez ces deux exemples.

Dans ce TP nos boucles `for` parcourront des `range`, mais on peut aussi parcourir des listes, des chaînes de caractères et même des dictionnaires (*cf.* Chapters ALGO2 et ALGO3). Lorsque l'indice d'une boucle `for` n'est pas utilisé dans un bloc, on peut omettre de lui donner un nom, voici un exemple.

Répétition d'instructions

```
>>> for _ in range(3):
...     print("Coucou !")
...
Coucou !
Coucou !
Coucou !
```

Résumé

- `range(a, b)` : parcourt tous les entiers entre `a` et `b-1` si $b > a$. Dans le cas contraire, ne fait rien.
- `range(a, b, h)` : parcourt tous les entiers entre `a` et `b-1` si $b > a$ en avançant avec un pas de `h`. Dans le cas contraire, ne fait rien.

Exercice 2 | Calculer un produit : la factorielle Solution Si $n \in \mathbf{N}$ est un entier, on appelle *factorielle de n* la quantité définie par :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ 1 \times 2 \times \cdots \times n. \end{cases}$$

Écrire une fonction d'en-tête `facto(n)`, où `n` est un entier, qui renvoie la valeur de $n!$. *On prévoira tous les cas possibles sur* `n`

Exercice 3 | Calculer une somme Solution

1. Écrire une fonction d'en-tête `harmo(n)`, `n` étant un entier strictement positif, et qui renvoie la valeur de

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \cdots + \frac{1}{n}.$$

2. Écrire un script qui affiche la valeur de $\sum_{k=1}^{10^p} \frac{1}{k} - \ln(10^p)$ pour $p \in [\![1, 8]\!]$. Qu'observe-t-on ?

Attention La présence d'un `return` arrête une boucle `for` !

Un point très important est à constater : dès que, au sein d'une fonction, on arrive sur une instruction `return` dans une boucle `for`, alors la boucle `for` est arrêtée. Voyez par exemple :

```
def f():
    n = 0
    for _ in range(3):
        n += 1
        return n

>>> f()
1
```

L'entier n n'a été augmenté qu'une seule fois de 1, le premier **return** a arrêté complètement la boucle **for**. De manière générale, le **return** arrête toute la fonction.

Exercice 4 | **Vérification de formules** Solution

1. Écrire une fonction d'en-tête `somme_puissance(p, n)` prenant en argument deux entiers, et renvoyant la valeur de $\sum_{k=0}^{n} k^p$.
2. Vérifier, à l'aide de Python, les assertions mathématiques ci-après :

$$\forall n \in [\![0, 10]\!], \quad \sum_{k=0}^{n} k = \frac{n(n+1)}{2}, \quad \sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}, \quad \sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4}.$$

Exercice 5 | **Dessins** Solution Écrire des fonctions prenant en argument le nombre de lignes n (ici `n = 5`) et permettant l'affichage des figures suivantes.

1.
```
    *
   * *
  * * *
 * * * *
```

2.
```
. . . .
* . . .
* * . .
* * * .
* * * *
```

3.
```
* * * *
. . . *
. . . *
. . . *
* * * *
```

Indication : On pourra observer le résultat de l'instruction `"*"*3+"."*2` *dans la console.*

Calculer les termes d'une suite. Commençons avec une suite récurrente à 1 pas, c'est-à-dire le terme suivant ne dépend que du terme précédent.

$$u_0 = 1, \quad \forall n \in \mathbf{N}, \quad u_{n+1} = 2u_n + 3.$$

On souhaite créer une fonction qui va retourner la valeur de u_n pour tout $n \in \mathbf{N}$. L'idée est la suivante :

Méthode **Calculer informatiquement le terme u_n d'une suite récurrente $u_{n+1} = f(u_n)$**

- On stocke la valeur initiale de la suite dans une variable u $\leftarrow u_0$.
- On itère la relation de récurrence (en faisant ici `u = 2*u + 3`) autant de fois que nécessaire à l'aide d'une boucle **for** de longueur n. Le plus simple est de faire coïncider l'indice des mathématiques avec celui de la boucle **for**.

Exemple 7 Pour notre suite précédente, cela donnerait :

```
def suite_U_exemple(n):
    u = 1
    for _ in range(1, n+1):
        u = 2*u + 3
    return u
```

On peut alors tester.

```
>>> suite_U_exemple(3)
29
>>> suite_U_exemple(0)
1
```

À vous de jouer.

Exercice 6 | Suite récurrente à un pas Solution On considère la suite « arithmético-géométrique »[5] définie par :

$$u_0 = 4, \quad \forall n \geq 0, \quad u_{n+1} = 2 - \frac{u_n}{2}.$$

Écrire un programme itératif, *i.e.* faisant appel à une boucle `for`, prenant en argument un entier n et qui calcule u_n.

Pour les récurrences à deux pas, cela se complique très légèrement. Considérons à nouveau un exemple :

$$u_0 = 1,\ u_1 = -1, \quad \forall n \in \mathbf{N}, \quad u_{n+2} = 2u_{n+1} - u_n.$$

Méthode Calculer informatiquement le terme u_n d'une suite récurrente $u_{n+2} = f(u_n, u_{n+1})$

- On stocke les premières valeurs de la suite dans deux variables `u` $\leftarrow u_0$, `v` $\leftarrow u_1$.
- On itère la relation de récurrence (en faisant ici `w = 2*v - u`) autant de fois que nécessaire à l'aide d'une boucle `for` de longueur `n`. Le plus simple est de faire coïncider l'indice des mathématiques avec celui de la boucle `for`. On modifie ensuite les variables précédentes `u, v = v, w` : le nouveau `u` est `v`, le nouveau `v` est `w`.

Attention

Il ne faut surtout pas faire quelque chose comme `v = 2*v - u`, puisqu'alors on perdrait l'ancienne valeur de `v` qu'il faudrait mettre dans `v`. La bonne méthode est bien de créer une troisième variable `w`.

Exemple 8 Pour notre suite précédente, cela donnerait :

```
def suite_U_exemple2(n):
    """
    retourne la valeur de u_n, pour n supérieur à 2 uniquement
    """
    u, v = 1, -1
    for _ in range(2, n+1):
        w = 2*v - u
        u, v = v, w
    return w
```

On peut alors tester.

```
>>> suite_U_exemple2(3)
-5
>>> suite_U_exemple2(4)
-7
```

Exercice 7 | Suite récurrente à deux pas – Suite de Fibonacci Solution On rappelle que la suite de Fibonacci (u_n) est définie par :

$$u_0 = 0, \quad u_1 = 1, \quad \forall n \in \mathbf{N}, \quad u_{n+2} = u_{n+1} + u_n.$$

1. Calculer à la main u_n pour $n \in [\![0, 5]\!]$.

[5] Nous étudierons ces suites dans le cours de Mathématiques

2. Écrire un programme itératif, *i.e.* faisant appel à une boucle `for`, prenant en argument un entier n et qui calcule u_n.
3. Contrôler la valeur obtenue pour u_5.

Exercice 8 | Fonctions mystères Solution Pour chaque fonction ci-après, écrire ce qu'elle renvoie en interprétant son résultat.

```
def mystere_1(N):
    S = 0
    for k in range(1, N+1):
        S += k
    return S
```

```
def mystere_2(N):
    S = 2
    for k in range(3):
        S = S**2
    return S
```

Compter. Python peut également servir à compter des choses au moyen d'une variable, appelée *compteur*, vouée à augmenter de 1 à chaque occurence de comptage. La structure type est donc la suivante.

Comptage

```
def comptage():
    N = 0 # compteur
    for _ in .....:
        if evenement_a_compter:
            N += 1
    return N
```

Exercice 9 | Nombre d'entiers divisibles par 4 Solution Écrire une fonction d'entête `compte_divisibles_par_quatre(n)`, `n` étant un entier, retournant le nombre de nombres divisibles par 4 entre 0 et n. *Indication : On rappelle que le symbole % permet de retourner le reste de la division euclidienne.*.

Optimiser : trouver le minimum/maximum d'une fonction sur une liste d'entiers. Cet exercice n'est à traiter qu'en fin de TP, s'il reste du temps. Les techniques mises en jeu seront largement revues dans le Chapter ALGO2 sur les listes.

Exercice 10 | Trouver un maximum / minimum Solution Dans cet exercice, on suppose créee une fonction `f` qui prend en argument un entier et renvoie un flottant. On pourra définir pour tout l'exercice :

```
def f(x):
    return x**2
```

puis en tester d'autres à la fin.

1. On commence par essayer de chercher le maximum.
   1.1) Compléter la fonction `maximum` ci-après, prenant en argument un entier $N \in \mathbf{N}$ et retournant la plus grande valeur parmi `f(0), ..., f(N)`.

```
def maxi_f(N):
    """
    N : int -> maximum de f(0), ..., f(N)
    """
    maxi = f(0)
    for k in range(......):
        if .... > maxi :
            maxi = .....
    return maxi
```

   1.2) Dans la fonction précédente, peut-on remplacer le symbole > par >=?
   1.3) Adapter la fonction précédente, en une fonction `maxi_ind_f`, et retournant en plus du maximum un indice $i \in [\![0, N]\!]$ où $f(i)$ est égal audit maximum.
2. Faire le même travail que dans la première question, mais pour trouver le minimum.

3.2.2. Conditionnelle : boucle `while`

L'arrêt de la boucle dépend ici d'une condition. On se sait pas *a priori* lorsqu'elle va se terminer. Il convient donc de faire attention au fait que cette boucle se termine bien avant de lancer le code python.

Boucle while

```
while test:
    instructions
```

Exercice 11 | Suite récurrente à 1 pas, le retour. Algorithme de seuil. Solution On reprend l'exercice 6. On admet (en attendant le cours de Mathématiques) que cette suite « converge vers $\frac{4}{3}$ », c'est-à-dire que u_n est aussi proche que l'on veut de $\frac{4}{3}$ pourvu que n soit assez grand. Autrement dit, u_n est une bonne approximation de $\frac{4}{3}$ lorsque n est grand.

Écrire une fonction d'en-tête `cherche_n(eps)` prenant en argument `eps` un réel strictement positif, et qui retourne le premier entier n de sorte que u_n soit assez proche de $\frac{4}{3}$ au sens suivant :

$$\left|u_n - \frac{4}{3}\right| < \varepsilon.$$

Exercice 12 | Suite de Heron. Algorithme de seuil. Solution Soit $a \in \mathbf{R}^{+\star}$ et (u_n) la suite récurrente définie par :

$$a_0 = a, \quad \forall n \in \mathbf{N}, \quad u_{n+1} = \frac{u_n^2 + a}{2u_n}.$$

On admet (en attendant le cours de Mathématiques) que cette suite « converge vers $\sqrt{a}$ », c'est-à-dire que u_n est aussi proche que l'on veut de $\sqrt{a}$ pourvu que n soit assez grand. Autrement dit, u_n est une bonne approximation de $\sqrt{a}$ lorsque n est grand.

1. Écrire un programme itératif, *i.e.* faisant appel à une boucle `for`, prenant en argument un entier n et qui calcule u_n.
2. Écrire une fonction d'en-tête `heron_approx(a, eps)` prenant en argument a et `eps` un réel strictement positif, et qui retourne le premier entier n de sorte que u_n soit assez proche de $\sqrt{a}$ au sens suivant :

   $$\left|u_n - \sqrt{a}\right| < \varepsilon.$$

   En combien d'étapes la suite de Heron donne-t-elle une valeur approchée de $\sqrt{167}$ à 10^{-4} près ? à 10^{-8} près ?

Exercice 13 | Suite & Conjecture de Syracuse Solution Soit $N > 0$. On définit alors la suite suivante

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{si } u_n \text{ est impair,} \end{cases} \qquad u_0 = N.$$

La conjecture de Syracuse est la suivante : « pour tout entier $N > 0$, il existe un indice n tel que $u_n = 1$ ».

1. Créer une fonction d'en-tête `Syracuse(N, n)` et qui retourne la valeur de u_n, pour n un entier positif.
2. On appelle *temps de vol* le plus petit indice n tel que $u_n = 1$. Créer une fonction d'en-tête `temps_vol(N)` et qui retourne le temps de vol de la suite. *On prévoit que si l'on a pas atteint* 1 *en* 10^3 *coups, on retourne* `False`.
3. On appelle *altitude maximale de vol* la plus grande valeur de u_n jusqu'à son temps de vol (c'est-à-dire d'atteinte de 1). Adapter la fonction précédente pour qu'elle retourne en plus l'altitude maximale de vol.

4. SOLUTIONS DES EXERCICES

Solution (exercice 1)

Énoncé

```
def f_1(x):
    if x <= -1:
        return -x
    else:
        return x**2

def f_2(x):
    if x <= -1:
        return -x
    elif -1 < x < 2:
        return x**2
    else:
        return x+2
```

Solution (exercice 2)

Énoncé

```
def facto(n):
    if n == 0:
        return 1
    else:
        P = 1
        for k in range(2, n+1):
            P *= k # multiplication de P par k
        return P
```

```
>>> facto(1)
1
>>> facto(4)
24
```

Solution (exercice 3)

Énoncé

```
def harmo(n):
    S = 0
    for k in range(1, n+1):
        S += 1/k # multiplication de P par k
    return S

>>> for p in range(1, 9):
...     print(harmo(10**p)-ma.log(10**p))
...
0.6263831609742079
0.5822073316515288
0.5777155815682065
0.5772656640681646
0.5772206648931064
0.5772161649007153
0.5772157148989514
0.5772156699001876
```

On constate que lorsque p grandit, la suite se rapproche d'une certaine valeur. *La valeur en question s'appelle la constante d'Euler, elle sera étudiée dans le cours de Mathématiques en 2ème année.*

Solution (exercice 4)

Énoncé

```
def somme_puissance(p, n):
    S = 0
    for k in range(0, n+1):
        S += k**p
    return S

def verifications_formules(n):
    verif_1 = (somme_puissance(1, n) == n*(n+1)/2)
    verif_2 = (somme_puissance(1, n) == n*(n+1)/2)
    verif_3 = (somme_puissance(1, n) == n*(n+1)/2)
    return verif_1 and verif_2 and verif_3

def verifications():
    for n in range(0, 11):
        if verifications_formules(n) == False:
            return False
    # si on arrive ici, c'est que toutes les formules sont
    ↪ vérifiées
    return True

>>> verifications()
True
```

Solution (exercice 5)

Énoncé

```
def dessin_1(n):
    for i in range(n):
        print('*'*i)

def dessin_2(n):
    for i in range(n):
        print('*'*i+'.'*(n-i-1))

def dessin_3(n):
    print('*'*(n-1))
    for i in range(1,n-1):
        print('.'*(n-2)+'*'*(1))
    print('*'*(n-1))
```

```
>>> dessin_1(5)

*
**
***
****
>>> dessin_2(5)
....
*...
**..
***.
****
>>> dessin_3(5)
****
...*
...*
...*
****
```

Solution (exercice 6)

Énoncé

```python
def suiteU_arithmgeo(n):
    u = 4
    for _ in range(1, n+1):
        u = 2 - u/2
    return u

>>> suiteU_arithmgeo(3)
1.0
```

Solution (exercice 7)

Énoncé À la main, on trouve

$$u_0 = 0, \quad u_1 = 1, \quad u_2 = 1, \quad u_3 = 2, \quad u_4 = 3, \quad u_5 = 5.$$

```python
def fib(n):
    """
    retourne la valeur de u_n
    """
    u, v = 0, 1
    # Cas particuliers
    if n == 0:
        return u
    elif n == 1:
        return v
    else:
        # cas n supérieur ou égal à 2
        for _ in range(2, n+1):
            w = u+v
            u, v = v, w
        return w

>>> fib(2)
1
>>> fib(3)
2
>>> fib(4)
3
>>> fib(5)
5
>>> fib(1)
1
>>> fib(0)
0
```

Solution (exercice 8)

Énoncé La première fonction calcule $\boxed{S_N = 1 + \cdots + N}$ donc $\frac{N(N+1)}{2}$ d'après le cours de Mathématiques. Dans la seconde on élève au carré la variable S à chaque étape, donc S prend successivement les valeurs $2, 4, 16, 16^2 = \boxed{256}$.

Solution (exercice 9)

Énoncé

```python
def compte_divisibles_par_quatre(n):
    N = 0
    for i in range(n+1):
        if i % 4 == 0:
            N += 1
    return N

>>> compte_divisibles_par_quatre(10)
3
```

Solution (exercice 10)

Énoncé

```
def f(x):
    return x**2

def maxi_f(N):
    """
    N : int -> maximum de f(0), ..., f(N)
    """
    maxi = f(0)
    for k in range(N+1):
        if f(k) > maxi :
            maxi = f(k)
    return maxi
```

Dans la fonction précédente, il est possible de remplacer le symbole > par >=. En effet, cela ne changera pas la valeur du maximum, en revanche, cela changera la valeur de l'indice trouvé dans la question qui suit.

```
def maxi_ind_f(N):
    """
    N : int -> maximum de f(0), ..., f(N), et un indice en lequel
 ↪  il est atteint
    """
    maxi = f(0)
    ind = 0
    for k in range(N+1):
        if f(k) > maxi :
            maxi = f(k)
            ind = k
    return maxi, ind

def mini_ind_f(N):
    """
    N : int -> maximum de f(0), ..., f(N), et un indice en lequel
 ↪  il est atteint
    """
    mini = f(0)
    ind = 0
    for k in range(N+1):
        if f(k) < mini :
            mini = f(k)
            ind = k
    return mini, ind
```

Pour le minimum c'est exactement la même fonction, il suffit de changer le symbole > en <. Sur la fonction carré, croissante sur $\mathbf{R}^+$, on trouve les résultats ci-après.

```
>>> maxi_ind_f(10)
(100, 10)
>>> mini_ind_f(10)
(0, 0)
```

Solution (exercice 11)

Énoncé

```
def suiteU_arithmgeo_seuil(eps):
    u = 4
    n = 0
    while abs(u-4/3) >= eps:
        u = 2 - u/2
        n += 1
    return n
```

```
>>> suiteU_arithmgeo_seuil(10**(-3))
12
```

Solution (exercice 12)

Énoncé

```
def suiteU_Heron(n, a):
    u = a
    for _ in range(1, n+1):
        u = (u**2+a)/(2*u)
    return u

>>> suiteU_Heron(10**2, 2)
1.414213562373095
>>> ma.sqrt(2)
1.4142135623730951

def heron_approx(a, eps):
    u = a
    n = 0
    while abs(u-ma.sqrt(a)) >= eps:
        u = (u**2+a)/(2*u)
        n += 1
    return n

>>> heron_approx(167, 10**(-4))
7
>>> heron_approx(167, 10**(-8))
8
```

Solution (exercice 13)

Énoncé

```
def syracuse(N, n):
    """
    retourne la liste des n premiers termes de syracuse
    """
    u = N
    for _ in range(1, n+1):
        if u % 2 == 0:
            u = u/2
        else:
            u = 3*u+1
    return u

>>> N = 10
>>> syracuse(N, 2)
16.0
>>> syracuse(N, 10)
4.0
```

Ensuite, on peut s'occuper du temps de vol.

```
def temps_vol(N):
    """
    retourne le temps du vol au-dessus de 1 de Syracuse
    """
    temps_vol = 0
    n = 0
    while syracuse(N, temps_vol) != 1:
        temps_vol += 1
    return temps_vol

def altitude_temps_vol(N):
```

```
    """
    retourne le temps du vol au-dessus de 1 de Syracuse
    """
    temps_vol = 0
    altitude_max = syracuse(N, temps_vol)
    n = 0
    while syracuse(N, temps_vol) != 1:
        temps_vol += 1
        valeur = syracuse(N, temps_vol)
        if valeur > altitude_max:
            altitude_max = valeur
    return temps_vol, altitude_max
```

Cette fonction n'est pas optimale puisqu'il est inutile de recalculer tous les termes de la liste à chaque fois.

```
>>> altitude_temps_vol(10)
(6, 16.0)
```

.....

Chapitre ALGO2.

Listes, Tuples & Chaînes de caractères

Résumé & Plan

Les listes, tuples et les chaînes de caractères sont des structures de données basiques en programmation. Les manipulations de ces types d'objets sont très proches : on se repère dans ces structures de la même façon, ai moyen d'entiers. Et c'est pour cette raison que nous les traitons de manière simultanée. Les listes sont à utiliser dans beaucoup de contextes généraux, les chaînes sont à privilégier si l'on a besoin de faire du traitement de texte (suppression d'accents, mise en minuscule/majuscule *etc.*), les tuples sont des listes non modifiables, on les utilise assez peu en tant que tel.

Les tentatives de création de machines pensantes nous seront d'une grande aide pour découvrir comment nous pensons nous-mêmes.

— Alan Turing

1. LISTES ET TUPLES

1.1. Généralités

- Une *liste* est une suite finie d'éléments pouvant être de types différents. Ces éléments sont modifiables (ou mutables), contrairement aux tuples que nous verrons juste après. On peut de plus ajouter ou enlever des éléments à une liste, ce qui permet de représenter des structures de données évoluant au cours du temps (création d'une liste initiale, puis modifications successives). On se repère dans les éléments d'une liste par des entiers relatifs (positif lorsque l'on compte directement, négatif en comptage à rebours).
- Les *tuples* (« n-uplets » en français) correspondent aux listes à la différence qu'ils sont non modifiables : c'est presque la seule et unique différence avec les listes !

Définir une liste ou un tuple en Python

Dans la pratique, on définit un tuple à l'aide de parenthèses *a contrario* des listes qui utilisent des crochets.

```
>>> L = ["ATGC", 1, 3.0, [1, 2, 3]] # une liste
>>> T = ("ATGC", 1, 3.0, [1, 2, 3]) # un tuple
```

On constate bien qu'il est possible que chaque élément d'une liste soit de type différent, et même soit de type liste! Les listes de listes seront assez largement utilisées pour traduire en Python la notion mathématique de matrice.

Différence entre les deux : la mutabilité des éléments

```
>>> L = ["ATGC", 1, 3.0, [1, 2, 3]]
>>> L[2]
3.0
>>> L[2] = 15 # ça marche !
>>> L
['ATGC', 1, 15, [1, 2, 3]]
>>>
>>> t = ("ATGC", 1, 3.0, [1, 2, 3])
>>> t[2]
3.0
>>> t[2] = 15 # oups !
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Précisons quelques autres propriétés sur les listes.

Longueur

Une liste possède une *longueur*, qui correspond au nombre d'éléments d'une liste, elle est donnée par la fonction `len` en Python. Par exemple :

```
>>> L = ["ATGC", 1, 3.0, [1, 2, 3]]
>>> len(L)
4
```

L'objet `"ATGC"` est ce qu'on appelle une «chaîne de caractères». Peu importe ce que c'est pour le moment, nous l'étudierons dans la prochaine section.

Repérage & Changement de valeur

On accède à un élément d'une liste en utilisant son *indice* dans ladite liste, numéroté **à partir de zéro**. Par exemple :

```
>>> L[0]
'ATGC'
>>> L[1]
1
>>> L[2]
3.0
>>> L[3]
[1, 2, 3]
>>> len(L[3]) # ici, le dernier élément est une liste donc possède
 ↪  lui aussi une longueur
3
>>> len(L[len(L)-1]) # le dernier élément a pour indice len(L) -1
 ↪  : ne pas oublier le -1 car on compte à partir de 0.
3
>>> # on peut aussi compter à rebours :
>>> L[-1]
[1, 2, 3]
>>> L[-2]
3.0
>>> L[-3]
1
```

Lorsque l'on sort des indices admissibles d'une liste, on obtient une erreur.

```
>>> L[5] # oups !
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list index out of range
```

Enfin, il est possible de modifier les éléments d'une liste, par exemple en faisant :

```
>>> L[1] = 2
>>> L
['ATGC', 2, 3.0, [1, 2, 3]]
>>> L[1] += 1 # on ajoute 1 au deuxième élément
>>> L
['ATGC', 3, 3.0, [1, 2, 3]]
```

Suppression, Ajout d'éléments & Concaténation

On peut facilement supprimer un élément d'une liste en connaissant son indice (à l'aide de **del**), ou alors supprimer le premier (en partant de la gauche) élément possédant une certaine valeur à l'aide de remove. Par exemple,

```
>>> L = ["ATGC", 1, 3.0, [1, 2, 3]]
>>> del L[1] # suppression du deuxième élément
>>> L # la liste est modifiée directement
['ATGC', 3.0, [1, 2, 3]]
>>> L.append("ATGC") # analogue à L += ["ATGC"]
>>> L
['ATGC', 3.0, [1, 2, 3], 'ATGC']
>>> L.remove("ATGC")
>>> L # supprime le premier "ATGC" en partant de la gauche
[3.0, [1, 2, 3], 'ATGC']
```

Parcourir une liste

On dispose de deux options pour cela : soit au moyen d'un entier correspondant à l'indice de l'élément, soit directement en parcourant les éléments de la liste.

```
>>> for i in range(len(L)):
...     print(L[i])
...     # à utiliser quand vous avez besoin de travailler avec i ensuite
...
3.0
[1, 2, 3]
ATGC
```

```
>>> for x in L:
...     print(x)
...     # à utiliser quand seul l'élément vous importe
...
3.0
[1, 2, 3]
ATGC
```

Attention

Bien mettre **for** i **in** range(len(L)) car on rappelle que par défaut un range s'arrête strictement avant la borne de droite. Ainsi, i parcourra 0, ..., len(L)-1 ce qui est exactement la liste des indices de ses éléments.

Concaténation

On peut aussi fusionner deux listes au moyen de l'opérateur +, ou encore de la méthode extend ou encore de manière « artisanale » à l'aide d'une boucle **for**.

```
>>> L
[3.0, [1, 2, 3], 'ATGC']
>>> M = [1, 2, 3]
>>> N = L + M
>>> N
[3.0, [1, 2, 3], 'ATGC', 1, 2, 3]
```

```
>>> L
[3.0, [1, 2, 3], 'ATGC']
>>> M = [1, 2, 3]
>>> L.extend(M)
>>> L # modifie directement la liste initiale
[3.0, [1, 2, 3], 'ATGC', 1, 2, 3]
```

On peut aussi utiliser, comme annoncé, plusieurs append au moyen d'une boucle **for**.

```
>>> L
[3.0, [1, 2, 3], 'ATGC', 1, 2, 3]
>>> M = [1, 2, 3]
>>> for x in M:
```

```
...     L.append(x)
...
>>> L
[3.0, [1, 2, 3], 'ATGC', 1, 2, 3, 1, 2, 3]
```

MODES DE DÉFINITION D'UNE LISTE. Comme pour les ensembles, il est possible de définir une liste de plusieurs manières.

Construire une liste par énumération

Une liste peut être définie par énumération (c'est le mode utilisé précédemment).

```
>>> L = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On peut également partir d'une liste vide, et la remplir à l'aide d'une boucle `for`.

```
>>> L_bis = []
>>> for i in range(10):
...     L_bis.append(i**2)
...
>>> L_bis
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Construire une liste par compréhension

On peut également la définir en compréhension :

```
>>> M = [i**2 for i in range(10)]
>>> M
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On peut aussi filtrer les éléments selon une condition, indiquée dans la compréhension. Par exemple, si on ne souhaite que les carrés d'entiers pairs, on tapera

```
>>> N = [i**2 for i in range(10) if i%2 == 0]
>>> N
[0, 4, 16, 36, 64]
```

Et pour finir, comme pour les boucles `for`, si l'indice du `range` n'est important, on peut utiliser le symbole underscore. Par exemple, pour créer une liste de zéros de taille 10, on tape

```
>>> zeros = [0 for _ in range(10)]
>>> zeros
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

CAS PARTICULIER DES LISTES D'ENTIERS. Pour créer rapidement des listes d'entiers, on se sert de `range`.

Produire des listes d'entiers avec `range`

L'instruction `list(range(n))` produit une liste d'entiers consécutifs entre 0 et $n-1$:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

De manière générale, l'instruction `list(range(a, b))` produit une liste d'entiers consécutifs entre `a` et `b-1` si `b-1` est supérieur ou égal à `a` et une liste vide dans le cas contraire. Dans le premier cas, on peut également spécifier un pas d'avancement : `list(range(a, b, h))` retourne la liste des entiers de la forme `a + h*k` tant que `a + h*k` est inférieur strictement à `b`.

Attention

Il faut bien faire attention au fait que `range(...)` **n'est pas** une liste mais un *itérateur, i.e.* un objet de référence pour décrire une boucle `for`. Il faut donc d'abord le convertir en liste pour afficher ses valeurs.

Exercice 1 | Années à coupe du monde Solution Construire de plusieurs manières la liste des années à coupe du monde de foot entre 1998 et 2100.

Exercice 2 | Tables de multiplication Solution Dans tout l'exercice, on répondra aux questions en une seule commande avec les instructions `range()` et `list()`.

1. Affichez la table de multiplication par 9, *i.e.* la liste des entiers $0, 9, 18, \ldots, 90$.
2. Combien y a-t-il de nombres pairs dans l'intervalle `[2, 10000]`?

LISTES DE LITES. On a parfois besoin de créer/manipuler des listes dont les éléments sont eux-mêmes des listes, il faut donc s'habituer à manipuler les syntaxes faisant intervenir plusieurs crochets `[]` à la suite. Plutôt que de longs discours, voyons cela avec des exercices.

Exercice 3 | Liste de listes Solution Créez 4 listes `hiver, printemps, ete` et `automne` contenant les mois correspondants à ces saisons sous forme de chaînes de caractères. Créez ensuite une liste `saisons` contenant les listes `hiver`, `printemps`, `ete` et `automne`. Prévoyez ce que renvoient les instructions suivantes, puis vérifiez-le dans l'interpréteur :

1. `saisons[2]`
2. `saisons[1][0]`
3. `saisons[1:2]`
4. `saisons[:][1]`. Comment expliquez-vous ce dernier résultat?

Exercice 4 | Séparation pairs / impairs Solution Écrire une fonction d'en-tête `split_list(L)` qui prend en entrée une liste d'entiers et qui renvoie une liste de listes, dont la première est la liste constituée des entiers pairs et la seconde de la liste des entiers impairs de la liste d'origine. *Par exemple,* `split_list([1,3,4,2,5,2])` *doit renvoyer* `[[4, 2, 2], [1, 3, 5]]`

DÉCOUPAGES DE LISTES. On peut aussi, à partir d'une liste complète, en extraire des tranches. Voyons comment. Soit `L` une liste de longueur `n`, et `p, q` deux entiers tels que $0 \leqslant p, q \leqslant n$.

- `L[p:q]` est la liste constituée de tous les éléments de `L` dont l'indice est dans $[\![p, q-1]\!]$.
- `L[:q]` est la liste constituée des `q` premiers éléments de `L`, c'est-à-dire ceux entre les indices `0` et `q-1`.[1]
- `L[p:]` est la liste `L` privée de ses `p` premiers éléments.

Découpage

```
>>> L = [0,1,2,3,4,5,6,7,8,9]
>>> L[0:3]
[0, 1, 2]
>>> L[3:6]
[3, 4, 5]
>>> L[3:4]
[3]
>>> L[3:3]
[]
>>> L[3:2]
[]
>>> L[3:]
[3, 4, 5, 6, 7, 8, 9]
>>> L[2:]
[2, 3, 4, 5, 6, 7, 8, 9]
>>> L[:] # on extrait tout, pas très utile à part pour copier des
 ↪ listes (voir ci-dessous)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Dans toute la suite, nous ne travaillerons plus qu'avec des listes.

1.2. Copies de listes

Les copies de listes sont sources de pièges, et cela nécessite bien une section dédiée. Regardons un premier exemple.

[1] Tout comme pour les `range`, la borne de droite est exclue.

```
>>> L = [1,2,3]
>>> M = L
>>> M
[1, 2, 3]
>>> L[1] = -15
>>> L
[1, -15, 3]
>>> M
[1, -15, 3]
```

On voit que `M` a également été modifiée. Ceci est dû au fait suivant : les éléments de `M` sont liés à ceux de `M`, l'instruction `M = L` ne réalise donc pas, par défaut, une copie en dure des éléments de `L`.

Ceci est dû au fait, qu'en Python, une liste est un *tableau d'adresses*, donc en faisant `M = L`, on crée une liste d'adresses qui sont les mêmes que celles de `L`. Ainsi, une modification de `L` entrainera une modification de `M`.

Remarque 1 (Intérêt de l'adressage) La taille (comptée en bits) d'une adresse est beaucoup plus petite que celle des données. Il est alors beaucoup plus rentable, par exemple dans les problèmes de tri, de travailler sur les adresses que sur les objets. Pour le comprendre, on peut faire l'analogie avec le problème concret suivant :
Un laboratoire de chimie contient 100 bombonnes numérotées de 1 à 100, et les techniciens disposent d'un carnet où est noté le contenu des différentes bombonnes. Néanmoins le carnet n'est pas du tout à jour ! Pour régler cela il y a deux façons de procéder :

- *Soit on transfère le contenu des différentes bombonnes pour que cela corresponde au carnet : c'est l'équivalent de travailler sur les données.*
- *Soit on corrige le carnet : c'est l'équivalent de travailler sur les adresses.*

Il y a une solution moins coûteuse que l'autre...

Comment effectuer une « vraie » copie ? Pour recopier une liste `X` **contenant des entiers** dans une autre liste `Y`, on écrira plutôt

```
X = Y[:] #un découpage crée une copie
#ou Y=list(X)
```

Et là ça fonctionne :

```
>>> X = [1,2,3]
>>> Y = X[:]
>>> Y
[1, 2, 3]
>>> X[1] = -15
>>> X
[1, -15, 3]
>>> Y
[1, 2, 3]
```

Remarque 2 (Pour des entiers, tout va bien) Le phénomène précédent ne se produit en revanche **pas** pour des entiers. En effet :

```
>>> a = 1
>>> b = a
>>> a = 2
>>> b
1
```

En revanche, la méthode précédente ne fonctionne toujours pas pour les listes de listes (donc les listes qui contiennent au moins une liste dans une des coordonnées). Voici une méthode fonctionnant toujours, à l'aide du module `copy`.

```
>>> import copy
>>> X = [[1, 2], [3, 4]]
>>> Y = copy.deepcopy(X)
>>> X[1][1] = 99
>>> X
[[1, 2], [3, 99]]
```

```
>>> Y
[[1, 2], [3, 4]]
```

1.3. Résumé non exhaustif des opérations sur les listes

Étant donnée une liste `L`, on dispose des opérations suivantes :

Commande	Effet
`len(L)`	longueur
`L[0]`	premier élément
`L[-1]`	dernier élément
`L[i:j]`	liste extraite des éléments d'indices entre i (inclus) et j (exclus)
`L[i:]`	liste extraite à partir de l'indice i (inclus)
`L.append(v)`	ajoute l'élément v à la fin de la liste
`L.remove(v)`	supprimer le premier élément v apparaissant dans la liste,
	retourne une erreur s'il n'est pas présent
`L.extend(s)`	ajoute la liste s à la fin de la liste
`L.instert(i,v)`	insert l'objet v à l'indice i
`del L[i]`	supprime l'élément d'indice i
`L.pop()`	supprime le dernier élément et retourne l'élément supprimé

On peut aussi citer d'autres fonctions et méthodes, qui serviront parfois dans certains exercices (si on vous l'y autorise) :

- `L.reverse()`, retourne la liste et modifie `L`,
- `L.sort()`, trie la liste dans l'ordre croissant et modifie `L`,
- `M = sorted(L)`, trie la liste par ordre croissant en créeant une copie de `L`,

1.4. Algorithmes classiques sur les listes

On s'interdira dans cette partie d'utiliser une fonction déjà existante de Python.

Attention

Cette sous-section est presque la plus importante de l'année en Informatique. Toutes ces briques de base algorithmiques seront très largement réutilisées tout au long de votre CPGE.

Exercice 5 | Somme des éléments d'une liste Solution

1. Écrire une fonction itérative d'en-tête `somme(L)` qui renvoie la somme des termes de la liste.
2. Même question mais avec une version récursive.

Exercice 6 | Maximum/Minimum d'une liste Solution

1. Écrire une fonction `maximum_occur(L)` qui recherche le maximum de la liste `L` et renvoie la liste des occurrences où le maximum est atteint.
2. Même question avec le minimum.

Exercice 7 | Appartenance d'un élément dans une liste – Méthode par balayage Solution

1. Écrire une fonction `appartient(x, L)` qui détermine si la liste L contient l'élément `x`. Notez que cette fonction existe déjà : `x in L` renvoie le même résultat.
2. Écrire une fonction `indice(x, L)` qui renvoie le premier indice où l'élément x apparait dans la liste L, et `None` si L ne contient pas `x`. Notez que cette fonction existe déjà : `L.index(x)` renvoie le même résultat.
3. **(Suppression des doublons)** Écrire une fonction d'en-tête `suppr_doublon(L)` qui retourne une autre liste contenant les éléments de `L` mais apparaissant une unique fois. *Par exemple,* `suppr_doublon([1,2,2,3,5,4,5])` *retournera* `[1, 2, 3, 5, 4]`.

Exercice 8 | Renverser une liste Solution Écrire une fonction d'en-tête `renverse(L)`, et qui étant donnée une liste `L`, retourne la même liste mais avec des éléments écrits dans l'autre sens. *Notez que cette opération existe déjà sous forme de méthode, qui modifie* `L` *directement :*

```
>>> L = [1, 2, 3]
>>> L.reverse()
>>> L
[3, 2, 1]
```

Exercice 9 | **Décaler une liste** Solution Écrire une fonction d'en-tête `decale(L)`, et qui étant donnée une liste `L`, retourne la même liste mais avec les éléments décalés une fois vers la droite.

Exercice 10 | **Insérer dans une liste** Écrire une fonction `inserer` qui prend en entrée une liste, un élément `x` et un indice `i`, et qui insère l'élément `x` dans la liste à la position `i`. *Notez que cette fonction existe déjà, on peut utiliser la méthode* `insert` *sur les listes.* Attention : on demande une fonction qui **modifie la liste `L` donnée en entrée**.

Exercice 11 | **Appliquer une fonction sur une liste** Solution L'exercice suivant peut être fait très facilement avec la bibliothèque `nummpy`, mais nous allons nous en passer.

1. Créer une fonction d'en-tête `Applique_Fonction(L, f)` qui étant données une liste `L` et une fonction `f`, retourne la liste où la fonction `f` est appliquée à chaque élément.
2. **(Application)** Créer une fonction `Calc_Somme_Carre` prenant en argument un entier `n` et retournant $\sum_{i=1}^{n} i^2$. *On pourra se servir de la fonction* `sum` *de Python*

POUR ALLER PLUS LOIN. On propose ici quelques exercices supplémentaires une fois tout le reste du TP terminé.

Exercice 12 | **Valeurs extrêmes** Solution Écrire une fonction qui renvoie `True` si le premier et le dernier élément d'une liste sont identiques, et `False` sinon.

Exercice 13 | **Max/Min en un coup** Solution Proposer une fonction d'en-tête `min_max(L)` qui retourne le minimum et le maximum de `L`, à l'aide d'une seule boucle `for`.

Exercice 14 | **Deux maximums d'une liste** Solution Écrire deux fonctions d'en-tête `deux_maximum(L)`, qui à une liste `L` retourne :

- si elle ne possède qu'un élément, retourne cet élément,
- si elle en possède au moins deux, retourne les deux plus grands.

La première sera basée sur la fonction `maximum_occur`, la seconde sera indépendante.

2. CHAÎNES DE CARACTÈRES

2.1. Généralités

Les chaînes de caractères (type `string` sont des listes de caractères (lettres de l'alphabet, chiffres, symboles). On les note entre guillemets[2] ou apostrophes :

```
>>> ch1 = "bcpst1" # <- utiliser plutôt cette version, moins
 ↪  sensible aux accents
>>> ch2 = 'bcpst1'
```

Nous présenterons très peu de généralités sur les chaînes, car elles se manipulent essentiellement comme des listes.

- On accède à chacun des caractères comme pour une liste :

```
>>> ch1[0]
'b'
```

- et on dispose des fonctions de concaténation (`ch2 + ch2`).

```
>>> ch3 = ch1+ch2 # création d'une nouvelle chaîne par
 ↪  concaténation
>>> ch3
'bcpst1bcpst1'
```

[2] Mieux vaut privilégier ceux-ci afin d'éviter les complications dues aux accents

- Une chaîne possède une longueur, et attention : les espaces sont comptés comme un caractère.

```
>>> s = "les BCPST1 aiment l'informatique"
>>> len(s)
32
```

Attention On ne peut pas modifier une chaîne existante!

Il est fondamentale de retenir qu'on ne peut changer une chaîne existante. Comme pour les `tuple`, c'est un type non mutable. Voyez l'exemple ci-après.

```
>>> ch1 = "bcpst1"
>>> ch1[5] = "2"
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> ch1_modif = ch1[0:5] + "2" # Il faut créer une nouvelle
 ↪  chaîne
```

Ainsi, pour manipuler des chaînes, on passe souvent par des listes que l'on retransforme en chaîne.[3]

Passage de chaînes aux listes

```
>>> s = "lesBCPST1aimentl'informatique"
>>> L = list(s)
>>> L # là on peut modifier L
['l', 'e', 's', 'B', 'C', 'P', 'S', 'T', '1', 'a', 'i', 'm', 'e',
 ↪  'n', 't', 'l', "'", 'i', 'n', 'f', 'o', 'r', 'm', 'a', 't',
 ↪  'i', 'q', 'u', 'e']
>>> "".join(L) # permet de revenir à la chaîne initiale
"lesBCPST1aimentl'informatique"
```

Notez que l'on peut, comme pour les listes :

- effectuer des découpages de chaînes (avec la même syntaxe),
- parcourir une chaîne avec une boucle `for`.
- En revanche, on ne **peut pas** créer de chaînes par compréhension.

Opérations spécifiques aux chaînes.

Séparer les éléments d'une phrase en fonction des espaces

```
>>> s = "les BCPST1 aiment l'informatique"
>>> L = s.split()
>>> L
['les', 'BCPST1', 'aiment', "l'informatique"]
```

2.2. Résumé non exhaustif des opérations sur les chaînes

Etant donnée une chaîne `s`, on dispose des opérations suivantes (très proches des précédentes), avec en plus des méthodes/fonctions sur la mise en forme de caractères propres aux chaînes (passage en majuscule, en minuscule, séparation des caractères pour les mettre dans une liste, *etc.*).

Commande	Effet
`len(s)`	longueur
`s[0]`	premier élément
`s[-1]`	dernier élément
`s[i:j]`	chaîne extraite des éléments d'indices entre i (inclus) et j (exclus)
`s[i:]`	chaîne extraite à partir de l'indice i (inclus)
`s.join`	concatène à la chaîne de départ la nouvelle
`L = s.split()`	affecte à `L` la chaîne `s` dont les mots ont été séparés (et toute la chaîne si elle n'a pas de trou)

On dispose aussi de méthode permettant de traiter du texte :

[3] Mais peut-être que le type `string` n'était alors pas le bon type à utiliser dans ce contexte ...

```
>>> ch = "Ils vécurent HEUREUX et eurent beaucoup d'ENFANTS"
>>> ch.lower() # passage en minuscule
"ils vécurent heureux et eurent beaucoup d'enfants"
>>> ch.upper() # passage en majuscule. Ces deux méthodes ne
 ↪  modifient pas ch.
"ILS VÉCURENT HEUREUX ET EURENT BEAUCOUP D'ENFANTS"
```

Il en existe encore beaucoup d'autres (suppression des espaces, des accents par exemple) mais inutiles de toutes les connaitre. S'il y en besoin, l'exercice vous les donnera.

2.3. Algorithmes classiques sur les chaînes

Comme pour les listes, il y a des algorithmes très classiques sur les chaînes à bien connaitre. Les chaînes forment un moyen très commode pour traiter des problèmes de génétique, une séquence d'ADN pouvant être codée très simplement à l'aide d'une chaîne ne comportant que les lettres `A, T, G, C`.

Exercice 15 | Rechercher un caractère dans une chaîne – Méthode par balayage
Écrire une fonction `cherche_carac(x, ch)` qui détermine si la chaîne `ch` contient l'élément `x`.

Exercice 16 | Recherche de mots dans une chaîne Solution

1. Écrire une fonction `Recherche_Mot(m, t)` qui recherche si le mot `m` est présent dans le texte `t`, et qui renvoie la position de la première occurence du mot s'il est présent, et `None` sinon.
2. **(Application)** On dit qu'une chaîne de caractère code une séquence `t` d'ADN si elle est composée uniquement des lettres `'A'`, `'T'`, `'G'`, `'C'` et qu'un *codon stop* est un triplet du type `TAA`, `TAG` ou `TGC`.
   Écrire une fonction d'en-tête `codonstop(t)` qui renvoie `True` si la séquence `t` contient un codon stop et `False` sinon.

Pour aller plus loin. Vous trouverez ici quelques exercices supplémentaires à destination des plus rapides et des 5/2, si vous n'êtes pas dans ce cas de figure, passez à la suite.

Exercice 17 | Nombre de caractères communs entre deux chaînes Solution Écrire une fonction d'en-tête `compte_commun(c1, c2)` prenant en argument deux chaînes `c1, c2` et qui retourne le nombre de caractère en commun.

Exercice 18 | Décalages dans une chaîne, prémisses du codage de Vigenère Solution

1. Écrire une fonction `codage(c)` qui étant donnée une chaîne `c` ne comportant que des lettres non accentuées, retourne leur indice dans l'alphabet entre 0 et 25.
2. Écrire une fonction `decodage(L)` qui étant donnée une liste `L` ne comportant que des entiers entre 0 et 25, retourne la chaîne associée.
3. Écrire une fonction `decalage(c, n)` qui étant donnée une chaîne `c` ne comportant que des lettres non accentuées, retourne une autre chaîne dont les lettres ont été décalées de `n` modulo 26.

3. SOLUTIONS DES EXERCICES

Solution (exercice 1)

Énoncé Je rappelle qu'une coupe du monde a lieu tous les 4 ans.

1 ère solution : par compréhension.

```
>>> L = [1998 + 4*k for k in range(100) if 1998 + 4*k <= 2100]
>>> L
[1998, 2002, 2006, 2010, 2014, 2018, 2022, 2026, 2030, 2034, 2038,
 ↪  2042, 2046, 2050, 2054, 2058, 2062, 2066, 2070, 2074, 2078,
 ↪  2082, 2086, 2090, 2094, 2098]
```

2ème solution : par complétion à l'aide d'un `while`

```
>>> M = [1998]
>>> k = 0
>>> while 1998+4*k <= 2100:
...     M.append(1998+4*k)
...     k += 1
...
>>> M
[1998, 1998, 2002, 2006, 2010, 2014, 2018, 2022, 2026, 2030, 2034,
 ↪  2038, 2042, 2046, 2050, 2054, 2058, 2062, 2066, 2070, 2074,
 ↪  2078, 2082, 2086, 2090, 2094, 2098]
```

Solution (exercice 2)

Énoncé Il s'agit de faire afficher les entiers $9k$ avec $k \in [\![1\,,\,9]\!]$.

```
>>> list(range(0, 91, 9))
[0, 9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

Cela peut également se faire par compréhension.

Pour la seconde question, la commande suivante répond au problème.

```
>>> L = list(range(2,100001,2))
>>> len(L)
50000
```

Solution (exercice 3)

Énoncé

```
>>> hiver = ['Janvier', 'Fevrier', 'Mars']
>>> printemps = ['Avril', 'Mai', 'Juin']
>>> ete = ['Juillet', 'Aout', 'Septembre']
>>> automne = ['Octobre', 'Novembre', 'Decembre']
>>> saisons = [hiver, printemps, ete, automne]
>>> saisons[2]
['Juillet', 'Aout', 'Septembre']
>>> saisons[1][0]
'Avril'
>>> saisons[1:2]
[['Avril', 'Mai', 'Juin']]
>>> saisons[:]
[['Janvier', 'Fevrier', 'Mars'], ['Avril', 'Mai', 'Juin'],
 ↪  ['Juillet', 'Aout', 'Septembre'], ['Octobre', 'Novembre',
 ↪  'Decembre']]
>>> saisons[:][1]
['Avril', 'Mai', 'Juin']
```

Solution (exercice 4)

Énoncé

```
def split_list(L):
    L_imp = []
    L_pai = []
    for x in L:
        if x % 2 == 0:
            L_pai.append(x)
        else:
            L_imp.append(x)
    return [L_imp, L_pai]
```

Solution (exercice 5)

Énoncé

```
def somme(L):
    """
    Retourne la somme des éléments d'une liste
    """
    S = 0
    for x in L:
        S += x
    return S
```

La récursivité est basé sur le principe suivant : on extrait par exemple l'élément de tête, et on le somme avec la même fonction appliquée à la liste tronquée entre 1 et `len(L)`. On arrête le processus une fois que la liste tronquée deviens une liste vide.

```
def somme_rec(L):
    if len(L) == 0:
        return 0
    else:
        return L[0] + somme_rec(L[1:])
```

Solution (exercice 6)

Énoncé

```
def maximum_occur(L):
    """
    Retourne le maximum de L, et renvoie la liste des occurences où
    il apparaît
    """
    maxi = L[0]
    ind_maxi = []
    for k in range(1, len(L)):
        if L[k] > maxi:
            maxi = L[k]
    for k in range(len(L)):
        if L[k] == maxi:
            ind_maxi.append(k)
    return maxi, ind_maxi

def maximum_occur_bis(L):
    """
    Retourne le maximum de L, et renvoie la liste des occurences où
    il apparaît. Un seul parcours de la liste ici.
    """
    maxi = L[0]
    ind_maxi = [0]
    for k in range(1, len(L)):
```

```
        if L[k] == maxi:
            ind_maxi.append(k)
        if L[k] > maxi:
            maxi = L[k]
            ind_maxi = [k]
    return maxi, ind_maxi

def minimum_occur(L):
    """
    Cherche le minimum de L, et renvoie la liste des occurences
    """
    mini = L[0]
    ind_mini = []
    for k in range(1, len(L)):
        if L[k] < mini:
            mini = L[k]
    for k in range(len(L)):
        if L[k] == mini:
            ind_mini.append(k)
    return mini, ind_mini
```

Solution (exercice 7)

Énoncé

```
def appartient(x, L):
    """
    retourne True si x est dans L
    """
    for y in L:
        if y == x:
            return True
    return False
```

```
def indice(x,L):
    k = 0
    for i in range(len(L)):
        if L[i] == x:
            return i

def suppr_doublon(L):
    L_prim = []
    for x in L:
        if x not in L_prim:
            L_prim.append(x)
    return L_prim

>>> L = [1,2,2,3,5,4,5]
>>> suppr_doublon(L)
[1, 2, 3, 5, 4]
```

Solution (exercice 8)

Énoncé

```
def renverse(L):
    """
    retourne la liste des éléments de L écrits dans l'autre sens
    """
    L_renv = []
    for i in range(len(L)):
        L_renv.append(L[-i-1])
    return L_renv
```

Autres solutions possibles : utiliser un `range` à rebours, ou encore une liste par compréhension.

Solution (exercice 9)

Énoncé

Nous proposons plusieurs possibilités.

```
def somme(L):
    """
    Retourne la somme des éléments d'une liste
    """
    S = 0
    for x in L:
        S += x
    return S
```

Solution (exercice 11)

Énoncé

1.

```
def Applique_Fonction(L, f):
    """
    retourne une liste où f a été appliquée à chaque élément de
    ↪ L
    """
    return [f(x) for x in L]
```

2.

```
def Calc_Somme_Carre(n):
    """
    retourne la somme des n premiers carrés
    """
    L = list(range(0, n+1))
    return somme(L, lambda x:x**2)
```

Plutôt que d'utiliser l'instruction `lambda`, on aurait pu définir une fonction `f` à l'aide d'un `return` classique.

Solution (exercice 12)

Énoncé

```
def est_ident(L):
    return L[0] == L[-1]
```

Solution (exercice 13)

Énoncé

```
def min_max(L):
    """
    Cherche le minimum et le maximum de L
    """
    maxi = L[0]
    mini = L[0]
    for k in range(1, len(L)):
        if L[k] > maxi:
            maxi = L[k]
        elif L[k] < mini:
            mini = L[k]
    return mini, maxi
```

Cette fonction permet notamment de retourner facilement l'étendue d'une série statistique.

Solution (exercice 14)

Énoncé

Première solution, en se basant sur la fonction `maximum_occur`.

```
def deux_maxi(L):
    """
    retourne les deux plus grands éléments, où seulement le plus
    grand si L est de taille 1
    """
    if len(L) == 1:
        return L[0]
    else:
        deux_max = []
        for _ in range(2):
            M, ind_M = maximum_occur(L)
            deux_max.append(M)
            del L[ind_M]
        return deux_max
```

Première solution, version indépendante.

```
def deux_maxi_bis(L):
    """
    retourne les deux plus grands éléments, où seulement le plus
    grand si L est de taille 1
    """
    if len(L) == 1:
        return L[0]
    else:
        if L[1] > L[0]:
            maxi_1 = L[0]
            maxi_2 = L[1]
        else:
            maxi_1 = L[1]
            maxi_2 = L[0]
        # maxi 2 sera le plus grand des deux
        for x in L[2:]:
            if x > maxi_2 :
                maxi_1, maxi_2 = maxi_2, x
            elif maxi_1 < x < maxi_2:
                maxi_1 = x
        return maxi_1, maxi_2
```

Solution (exercice 16)

Énoncé

1.

```
def Recherche_Mot(m,t):
    l = len(m)
    if len(m)>len(t):
        return None
    else:
        for i in range(len(t)-l+1):
            if m == t[i:i+l]:
                return True
```

2.

```
def codonstop(t):
    return Recherche_Mot("TAA",t) or Recherche_Mot("TAG",t) or
    ↪ Recherche_Mot("TGC",t)
```

Solution (exercice 17)

Énoncé

```
def compte_commun(c1, c2):
    S = 0
    for c in c1:
        if c in c2:
            S += 1
    return S
```

Solution (exercice 18)

Énoncé

On peut commencer par créer une liste correspondant à l'alphabet, puis on retournera l'indice de chaque lettre dans cette liste.

```
alphabet =
 ↪ ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q',
       'r','s','t','u','v','w','x','y','z']
def codage(c):
    code = []
    for x in c:
        i = 0
        while x != alphabet[i]:
            i += 1
        code.append(i)
    return code

>>> codage("bonjour")
[1, 14, 13, 9, 14, 20, 17]

def decodage(L):
    M = [alphabet[i] for i in L]
    return "".join(M)

>>> decodage(codage("bonjour"))
'bonjour'

def decalage(c, n):
    L = codage(c)
    L_decal = [(i + n)%26 for i in L]
    return decodage(L_decal)
```

Chapitre ALGO3.

Dictionnaires

Résumé & Plan

Dictionnaires

Simple exercice mais le reste est à faire :

Dans une liste ou une chaîne, on se repère à l'aide d'entiers relatifs. Parfois, il peut s'avérer plus pratique d'utiliser autre chose pour se repérer. Par exemple, prenons trois candidats `CAhyerre`, `JHarter`, `NLescure` qui se présentent tous les trois à l'élection du meilleur professeur. Si l'on souhaite mémoriser le nombre de votes reçus par chaque candidat, on pourrait se dire qu'on les stocke dans une liste, un problème apparaît de suite : comment numéroter les candidats? Au lieu de taper `L[0]`, par exemple si `0` correspond à `CAhyerre`, on aurait plutôt envie d'écrire `L['CAhyerre']`. C'est précisément la structure de dictionnaire qui répond à ce problème. On pourrait aussi utiliser des listes de couples (appelées *liste d'associations* dans un futur exercice), mais ce point de vue a de nombreux inconvénients, le premier étant celui du souhait de changement de valeur qui impliquerait de savoir où se situe la clef associée dans la liste... pas pratique.

Les dictionnaires sont des collections **non ordonnées** d'objets. On remplace les valeurs entières d'indexage, par des clés (donc des objets plus généraux). Reprenons l'exemple *supra*.

```
>>> votes = {"CAhyerre" : 13, "JHarter":12, "NLescure":15}
>>> votes
{'CAhyerre': 13, 'JHarter': 12, 'NLescure': 15}
>>> # si l'on souhaite connaitre le nombre de votes pour CAhyerre,
 ↪  on tape alors
>>> votes["CAhyerre"]
13
```

- les chaînes `"CAhyerre"`, `...` de l'exemple précédent sont appelées les *clefs* du dictionnaire,
- les nombres de votes sont les *valeurs*.

Un dictionnaire n'est ni plus ni moins d'une application d'un ensemble `C` (les clefs) vers un ensemble `V` (les valeurs).

Par défaut `votes(clef_qui_nexiste_pas) = truc` va créer une autre correspondance dans le dictionnaire, si la clef est inexistante. Si elle existe, la valeur associée sera changée en `truc`.

> **Remarque 1** Toutes les clés de dictionnaire utilisées jusqu'à présent étaient des chaînes de caractères. Rien n'empêche d'utiliser d'autres types d'objets comme des entiers (voire même des tuples), cela peut parfois s'avérer très utile aussi de mélanger les deux.

Pour faire afficher toutes les couples clefs/valeurs d'un dictionnaire, on tape

```
>>> votes = {"CAhyerre" : 13, "JHarter":12, "NLescure":15}
>>> for key in votes:
...     # par défaut, ce sont les clefs qui sont parcourues
...     print(key, votes[key], end = ' ')
...
CAhyerre 13 JHarter 12 NLescure 15
```

Attention **Instruction `in`**

L'instruction `key in votes` demande à Python de parcourir toutes les clefs. Attention, c'est une différence avec les listes : `x in L` si `L` est une liste, demande à Python de parcourir les éléments de la liste — et pas les entiers qui l'indexent ! —. On peut aussi tester l'existence d'une clef en tapant `"poids" in votes`.

Exercice 1 | Liste d'associations Solution Écrire une fonction d'en-tête `couples(dic)` prenant en argument un dictionnaire `dic`, et retournant une liste de tuples correspondant aux couples clef/valeur.

On peut aussi récupérer les clefs et les valeurs à l'aide des méthodes idoine, ou encore parcourir.

```
>>> votes = {"CAhyerre" : 13, "JHarter":12, "NLescure":15}
>>> votes.keys()
dict_keys(['CAhyerre', 'JHarter', 'NLescure'])
>>> votes.values()
dict_values([13, 12, 15])
>>> # Que l'on peut convertir en liste
>>> list(votes.values())
[13, 12, 15]
```

Itérer sur un dictionnaire

```
for x in d: # Itérer sur les clefs
    ...
for y in d.values(): # Itérer sur les valeurs
    ...
for x,y in d.items(): # Itérer sur les couples clefs/valeurs
    ...
for i,j in enumerate(d): # Itérer sur les couples numéros de
 ↪ clefs/clefs
    ...
```

Exercice 2 | Tableau d'effectifs d'une liste Solution Créer une fonction d'en-tête `tab_eff(L)` qui étant donnée une liste `L` renvoie un dictionnaire qui contient comme clef les éléments de `L`, et pour valeurs la nombre d'occurences de chacun de ses éléments.

Exercice 3 | Construction d'un index Soit `s` une chaîne de caractères contenant des mots séparés par un espace (typiquement une phrase), créer une fonction `index(s)` qui renvoie un dictionnaires de clefs les mots de `s`, et de valeurs la liste des occurences. *Indication : On pourra utiliser la méthode* `split` *sur les chaînes comme présenté ci-desous.*

```
>>> s = 'Les BCPST1 vous êtes toujours là ?'
>>> liste_mots = s.split()
>>> liste_mots
['Les', 'BCPST1', 'vous', 'êtes', 'toujours', 'là', '?']
```

Attention

Une liste ne peut servir de clef de dictionnaire : de manière général seuls les objets non mutables — comme les tuple par exemple, qui existent d'ailleurs uniquement pour cette raison — peuvent servir de clef d'un dictionnaire.

Exercice 4 | Système de vote Solution On suppose qu'une urne peut être générée par les commandes suivantes.

Génération de l'urne

```
>>> import random as rd
>>> candidats = ["CAhyerre", "JHarter", "NLescure"]
>>> urne = []
>>> for _ in range(100):
```

```
...     urne.append(rd.choice(candidats))
...
>>> urne[0:20] # les 20 premiers bulletins
['JHarter', 'NLescure', 'CAhyerre', 'NLescure', 'CAhyerre',
 ↪  'CAhyerre', 'JHarter', 'CAhyerre', 'JHarter', 'CAhyerre',
 ↪  'CAhyerre', 'JHarter', 'JHarter', 'JHarter', 'CAhyerre',
 ↪  'JHarter', 'CAhyerre', 'NLescure', 'JHarter', 'NLescure']
```

Écrire une fonction `calcul_gagn(urne)` qui retourne le gagnant du vote. Tester sur une urne générée aléatoirement. *Indication : On pourra utiliser la fonction* `tab_eff` *précédente.*

1. SOLUTIONS DES EXERCICES

Solution (exercice 1)

Énoncé

```
def couples(dic):
    L = []
    for key in dic:
        L.append((key, dic[key]))
    return L

>>> couples({"CAhyerre" : 13, "JHarter":12, "NLescure":15})
[('CAhyerre', 13), ('JHarter', 12), ('NLescure', 15)]
```

Solution (exercice 2)

Énoncé

```
def tab_eff(L):
    """
    crée un dictionnaire de clefs les éléments de L et valeurs les
 ↪  nombres d'occurences
    """
    nb_appar = {}
    for x in L:
        if x not in nb_appar.keys():
            nb_appar[x] = 1
        else:
            nb_appar[x] += 1
    return nb_appar

>>> L = [1, 3, 2, 2, 2]
>>> tab_eff(L)
{1: 1, 3: 1, 2: 3}
```

Solution (exercice 4)

Énoncé

```
def tab_eff(L):
    """
    crée un dictionnaire de clefs les éléments de L et valeurs les
 ↪  nombres d'occurences
    """
    nb_appar = {}
    for x in L:
        if x not in nb_appar.keys():
            nb_appar[x] = 1
        else:
            nb_appar[x] += 1
    return nb_appar

def calcul_gagn(urne):
    """
    retourne le candidat gagnant
    """
    effectifs = tab_eff(urne)
    maxi = 0
    gagn = ''
    for candidat in effectifs:
        if effectifs[candidat] > maxi:
            maxi = effectifs[candidat]
            gagn = candidat
    return gagn
```

```
import random as rd
candidats = ["CAhyerre", "JHarter", "NLescure"]
urne = []
for _ in range(100):
    urne.append(rd.choice(candidats))
```

Pour cette partie, le gagnant est : NLescure.

Chapitre ALGO4.

Récursivité

Résumé & Plan

Récursivité

1. SOLUTIONS DES EXERCICES

Chapitre ALGO5.

Tris

Résumé & Plan

L'objectif de ce chapitre est de parcourir les principaux principes de tris de listes au programme de BCPST. Nous analyserons enfin très brièvement leur complexité temporelle, de manière empirique, à l'aide du module `time` de Python.

La question de savoir si les machines peuvent penser... est à peu près aussi pertinente que celle de savoir si les sous-marins peuvent nager.

— Edsger Dijkstra

1. GÉNÉRALITÉS

L'étude des tris est une partie incontournable de l'algorithmique. En effet, dans de nombreux contextes nous aurions besoin de trier une liste :

- trouver les 10 plus grands éléments d'une liste (par exemple, une grande liste comportant des relevés de températures sur 1 siècle, quelles sont les 10 années les plus chaudes ?). Pour cette problématique, il est bien plus pratique de trier plutôt que de faire cette recherche de maximums à la main (trouver les deux plus grands éléments d'une liste n'est pas si facile).
- Calculer le minimum et le maximum d'une liste, on extrait alors les éléments de tête et de queue de la version triée,

- trouver la médiane d'une série statistique, …

Trier a bien entendu un coût mais qui est vite compensé lorsque le nombre de recherches devient conséquent. Il est donc nécessaire de pouvoir trier efficacement. Nous comparerons leur efficacité avec le module `time`, donc de manière empirique, présenté dans le précédent chapitre. Mais pour commencer, mettons en place un petit test afin de savoir si une liste est triée ou non.

Exercice 1 | Une liste est-elle triée ? Solution

1. Écrire une fonction `tri_test_croissant(L)`, qui prend en paramètre une liste `L` et qui renvoie `True` si le tableau est trié par ordre croissant, et `False` sinon.
2. Même question mais dans l'ordre décroissant.

Définition 1 | Vocabulaire des tris

1. On appelle *tri* toute procédure algorithmique permettant de renvoyer une version triée (ou de modifier) d'une liste ou chaîne de caractères d'entiers.
2. Lorsque la procédure précédente modifie directement la liste ou chaîne d'entrée, on dit que le tri s'effectue *en place*.
3. Un tri est dit *stable* si les éléments égaux ont dans la liste triée le même ordre que dans la liste initiale.

Exemple 1 Par exemple un tri qui, appliqué à la liste `[3,2,1,2]`, renvoie la liste `[1,2,2,3]` n'est pas stable.

Pour analyser la stabilité, encore faut-il pouvoir évaluer précisément ce qu'il advient de chaque élément.

Remarque 1 Python sait déjà trier des listes (fonction `sorted` et méthode `.sort()`), mais elles sont à éviter[a] aux concours car vous devez avant tout connaître les tris du programme.

[a] sauf si on vous les autorise dans un sujet

2. TRIS ITÉRATIFS

Dans un premier temps, nous commençons par les tris non récursifs.

2.1. Tri stupide *(bogosort)*

LE PRINCIPE : ON MÉLANGE JUSQU'À L'OBTENTION D'UNE LISTE TRIÉE. Le tri stupide consiste à regarder si une liste `L` donnée est triée. Dans le cas contraire, on mélangera aléatoirement ses éléments jusqu'à obtenir une liste triée.

Exercice 2 | Programmation du tri stupide Solution Écrire une fonction d'en-tête `tri_stupide(L)` mettant en oeuvre cet algorithme. *On pourra se servir de la fonction* `shuffle` *du module* `random` *qui mélange les éléments d'une liste, le tout en place en place. Voici ci-dessous une démonstration.*

```
>>> import random as rd
>>> L = [1, 2, 3]
>>> rd.shuffle(L)
>>> L
[1, 2, 3]
```

On se doute bien que ce tri sera très inefficace, nous le constaterons plus tard. Passons à présent aux tris itératifs « plus intelligents ».

2.2. Tri à bulles

LE PRINCIPE : PARCOURS SUCCESSIFS DE LA LISTE ET ÉCHANGES DES PAIRES MAL ORDONNÉES. Le tri à *bulles* ou tri *par propagation* est un algorithme qui consiste à comparer répétitivement les paires d'éléments consécutifs d'une liste, et à les permuter lorsqu'ils sont mal triés. Il doit son nom au fait qu'il déplace rapidement les plus grands éléments en fin de tableau, comme des bulles d'air qui remonteraient rapidement à la surface d'un liquide.

Le tri à bulles est souvent enseigné en tant qu'exemple algorithmique, car son principe est simple, mais c'est le plus lent des algorithmes de tri communément enseignés, et il n'est donc guère utilisé en pratique. Un point clef est l'introduction d'un booléen qui va vérifier si après chaque parcours de liste on a eu besoin de « remonter » un élément : l'algorithme s'arrête alors lorsque plus aucun élément n'a eu besoin d'être remonté. Le code, déjà vu en première année, est directement donné, puis nous constatons simplement son fonctionnement sur un exemple.

Tri à bulles

```
def tri_bulles(L):
    """
    Modifie la liste L pour la trier, selon le tri à bulles
    Tri en place
    """
    echange_fait = True
    while echange_fait == True:
        echange_fait = False
        for j in range(0, len(L)-1):
            if L[j] > L[j+1]:
                echange_fait = True
                L[j], L[j+1] = L[j+1], L[j]
```

Exemple 2 On considère la liste `L = [1, -4, 7, 4, 2]`. Trions à la main, selon le tri bulles, la liste `L`, en précisant bien étape par étape l'état d'`echange_fait`.

Liste initiale	echange_fa	Commentaire
`L = [1, -4, 7, 4, 2]`	`True`	Début du parcours de `L`
`L = [-4, 1, 4, 2, 7]`	`True`	`[1, -4]`, `[7, 4]`, `[7, 2]` mal triées
`L = [-4, 1, 2, 4, 7]`	`True`	`[4, 2]` mal triées
`L = [-4, 1, 2, 4, 7]`	`False`	aucune paire mal triée

L'algorithme se termine, `echange_fait = False`. On constate que le tri a lieu en place puisque les permutations sont faites directement dans la liste initiale.

2.3. Tri par insertion

Le principe : le tri d'un jeu de carte. On insère un élément dans une liste d'éléments déjà triés (par exemple, par ordre croissant). Imaginez un joueur de cartes qui dispose de cartes numérotées. Il a des cartes triées de la plus petite à la plus grande dans sa main gauche, et une carte dans la main droite. La question est alors : où placer cette carte dans la main gauche de façon à ce qu'elle reste triée ? Il faut la placer après les cartes plus petites, et avant les cartes plus grandes.

Pour trier entièrement un ensemble de cartes dans le désordre, il suffit alors de placer toutes ses cartes dans la main droite (la main gauche est donc vide), et d'insérer les cartes une à une dans la main gauche, en suivant la procédure précédente.

Exemple 3 (Tri par insertion sur un exemple) On considère la liste `L = [1, -4, 7, 4, 2]`. Trions à la main, selon le tri par insertion, la liste `L`, en précisant bien étape par étape l'évolution de la version triée de `L` notée `L_trie`, et la liste `L`.

Liste triée	Liste initiale	Commentaire
`L_trie = []`	`L = [1, -4, 7, 4, 2]`	
`L_trie = [1]`	`L = [(1), -4, 7, 4, 2]`	On ajoute 1 dans `L_trie`
`L_trie = [-4, 1]`	`L = [1, (-4), 7, 4, 2]`	On insère -4
`L_trie = [-4, 1, 7]`	`L = [1, -4, (7), 4, 2]`	On insère 7
`L_trie = [-4, 1, 4, 7]`	`L = [1, -4, 7, (4), 2]`	On insère 4
`L_trie = [-4, 1, 4, 7]`	`L = [1, -4, 7, 4, (2)]`	On insère 2
`L_trie = [-4, 1, 2, 4, 7`	`L = [1, -4, 7, 4, 2]`	

L'algorithme se termine, on a fini de parcourir `L`.

Rappel – Insérer un élément dans une liste

Pour réaliser l'étape d'insertion, on utilisera la méthode `insert` sur les listes. Observons l'exemple ci-après.

```
>>> L = [1, 2, 3, 4]
```

```
>>> L.insert(1, -1)
>>> L
[1, -1, 2, 3, 4]
```

Nous avons donc inséré l'élément -1 à **gauche** de la position 1 (donc à droite de 2). Cette méthode peut être programmée très facilement au besoin, c'est ce que nous avions fait dans un précédent TP :

```
def insertion(L, i, x):
    """
    procédure
    modifie L en place, en insérant x en position i lorsque cela
    ↪ est possible
    retourne False si i n'est pas une position convenable
    """
    if 0 <= i < len(L):
        L.append(x)
        for j in range(len(L)-1, i, -1):
            L[j], L[j-1] = L[j-1], L[j]
            # on fait redescendre x à la bonne position par
            ↪ permutations successives
    else:
        return False
```

Exercice 3 | **Programmation du tri par insertion** Solution En complétant le script qui suit, créer une fonction `tri_insertion` qui trie une liste `L` selon le principe du tri par insertion.

```
def tri_insertion(L):
    """
    Retourne une version de L triée, selon le tri à insertion
    """
    L_tri = [L[0]]
    for k in range(1, len(L)):
        i = 0
        while (i < ______) and (L_tri[i] < ______):
            i += 1
        # insertion a la bonne place dans Lla version triée
        ______
    return L_tri
```

2.4. Tri par sélection (du minimum)

LE PRINCIPE : RECHERCHER LE `min` ET LE PLACER AU BON ENDROIT. Le tri par insertion insert chaque élément l'un après l'autre dans une autre liste tout en la gardant triée. Pour le tri par sélection, va chercher le plus petit élément de `L` et le placer au début en permutant avec le premier élément. On recommence suite avec la liste `L[1:len(L)]`.[1]. Nous donnons donc une version en place de ce tri.

Exemple 4 **(Tri pas sélection (version en place))** On considère la liste `L = [1, -4, 7, 4, 2]`. Trions à la main, selon le tri par sélection, la liste `L`, en précisant bien étape par étape le minimum trouvé, et dans quelle sous-liste on l'a cherché.

Liste (Minimum)	Liste après échange	Commentaire
`L = [1, (-4), 7, 4, 2]`	`L = [-4, 1, 7, 4, 2]`	le min était -4
`L = [-4, (1), 7, 4, 2]`	`L = [-4, 1, 7, 4, 2]`	le min était 1
`L = [-4, 1, 7, 4, (2)]`	`L = [-4, 1, 2, 4, 7]`	le min était 2
`L = [-4, 1, 2, (4), 7]`	`L = [-4, 1, 2, 4, 7]`	le min était 4

L'algorithme se termine, en `len(L)` étapes

Exercice 4 | **Programmation du tri par sélection en place** Solution Créer une fonction `tri_selection` qui trie une liste `L` selon le principe du tri par sélection du minimum. *Indication : Vous pourrez vous servir d'un ou plusieurs scripts du premier TP sur les listes.*

[1] On pourrait aussi chercher le maximum, puis le placer à la fin

Exercice 5 | Programmation du tri par sélection non en place Solution Imaginer une version non en place du tri sélection, en utilisant des instructions `del`, append.

3. TRI RÉCURSIF : LE TRI RAPIDE *(QUICKSORT)*

LE PRINCIPE : PARTAGER LA LISTE EN DEUX SOUS-LISTES D'ÉLÉMENTS INFÉRIEURS/SUPÉRIEURS OU ÉGAUX À UN ÉLÉMENT CHOISI AU HASARD, PUIS RECOMMENCER AVEC LES DEUX SOUS-LISTES. Le tri rapide est un algorithme récursif basé sur le principe « diviser pour régner ». Étant donnée une liste `L`, on commence par choisir le premier élément comme *pivot*, et on sépare la liste entre deux sous-listes : la première contient des éléments inférieurs ou égaux au pivot, et la seconde contient des éléments supérieurs (strictement) au pivot. Puis on applique le tri rapide aux deux sous-listes obtenues.

Exercice 6 | Programmation du tri rapide Solution

1. Écrire alors une fonction récursive `tri_rapide_rec(L)` qui trie une liste `L` avec le tri rapide, et de manière récursive. On complètera, par exemple, le code ci-dessous.

Tri Rapide, version récursive

```
def tri_rapide_rec(L):
    if L == []:
        # Condition d'arrêt
        return []
    pivot = L[0]
    inf_pivot = []
    sup_pivot = []
    for x in L[1:]:
        if ____________________:
            ____________________
        else:
            ____________________
    return ________________________ # on recommence
```

2. Proposer une autre version où `inf_pivot` et `sup_pivot` sont complétées par compréhension.

Il est possible aussi de programmer le tri rapide de manière itérative, mais le principe même de ce tri invite à utiliser de la récursivité.

4. APPLICATIONS, COMPLÉMENTS & EFFICACITÉ DES TRIS

4.1. Calcul d'une médiane

Exercice 7 | Solution Écrire une fonction d'en-tête `mediane(L)` qui étant donnée une série statistique correspondante à `L` retourne la médiane de cette série. *Indication : Si vous avez oublié ce qu'est une médiane, documentez-vous sur internet.*

4.2. Recherche dichotomique dans une liste triée

Le mot *dichotomie* signifie « Division, opposition (entre deux éléments, deux idées) ». Nous rencontrerons ce mot plusieurs fois au cours de l'année, une technique similaire sera déployée en Mathématiques pour approcher des solutions d'équations.

LE PRINCIPE. On s'intéresse ici à la recherche d'un élément noté `x` dans un tableau ou une liste dans laquelle les éléments de même type ont été préalablement triés par ordre croissant. Cette situation, bien qu'*a priori* particulière, se rencontre fréquemment en Informatique.

Dans un précédent TP, pour un tableau ou une liste non trié `L`, on a vu que l'on pouvait employer une méthode « par balayage » en utilisant un parcours simple de `L`. Cet algorithme dit « naïf » a pour propriétés :

- il s'applique à tout tableau ou liste `L`, sans nécessiter un ordre particulier entre les éléments de `L`,
- il met en jeu n tests dans le pire des cas, ce qui ne le place pas parmi les algorithmes très rapides.

Nous allons nous intéresser ici uniquement à des tableaux ou listes triés par ordre croissant, et appliquer un principe de recherche dichotomique qui conduit à une résultat avec beaucoup moins de comparaisons.

La recherche par dichotomie de l'élément `x` dans `L`, l'algorithme dichotomique consiste à :

1. initialiser deux indices `i_g = 0` (comme *indice gauche*), et `i_d = len(L)-1` (comme *indice droite*).
2. Calculer le alors `i_milieu = int((i_g+i_d)/2)`. L'élément `L[i_milieu]` est alors un élément *au milieu* de la liste `L`.
   - Si `L[i_milieu] = x`, l'algorithme est terminé.
   - Si `L[i_milieu] < x`, on recommence le processus avec `L[i_milieu+1:]`.
   - Si `L[i_milieu] > x`, on recommence le processus avec `L[:i_milieu]`.

Pourquoi cet algorithme est beaucoup plus rapide ? Car les tailles des sous-listes où l'on recommence la recherche diminue très vite (divisée par deux à chaque étape). En revanche, le tri utilisé coûte en temps.

Exercice 8 | Appartenance d'un élément dans une liste. Méthode par tri & dichotomie. Solution

1. Étant donnée une liste `L`, écrire une fonction `recherchedicho(x, L)` qui :
   - commence par trier `L` selon le tri rapide (par exemple),
   - fouille dans la liste triée pour trouver `x` selon un principe dichotomique.

   Indication : *On pourra compléter le script à trous ci-après.*

```
def recherche_dicho(x, L):
    """
    recherche un élément par méthode dichotomique
    """
    L_tri = tri_rapide_rec(L)
    i_g = 0
    i_d = ...
    while ..........:
        i_milieu = ...
        if L[i_milieu] == x:
            return ...
        elif L[i_milieu] < x:
            i_g = i_milieu+1
        else:
            i_d = ...
    return False # x non trouvé dans L
```

2. Que se passe-t-il avec `i_g, i_d` lorsque `x` n'est pas présent dans `L` ?
3. Étant donnée une liste `L`, écrire une fonction `recherchedicho_rec(x, L)` qui fait la même chose que précédemment, mais de manière récursive.

4.3. Autres tris

Nous allons voir deux autres tris possibles. Le premier est le tri par dénombrement, qui fonctionne pour les listes d'entiers (ou plus généralement des listes dont on connaît clairement les éléments qui peuvent y apparaître) et est basé sur la création d'un « tableau d'effectifs ». Nous l'utiliserons ultérieurement dans un prochain TP de Statistiques.

Exercice 9 | Tri par dénombrement (counting sort) Solution Soit `N` un entier naturel non nul. On cherche à trier une liste `L` d'entiers naturels strictement inférieurs à `N`.

1. Écrire une fonction `comptage`, d'arguments `L` et `N`, renvoyant une liste `P` dont le k-ième élément désigne le nombre d'occurences de l'entier k dans la liste `L`.
2. Utiliser la liste `P` pour en déduire une fonction tri, d'arguments `L` et `N`, renvoyant la liste `L` triée dans l'ordre croissant.
3. Tester la fonction tri sur une liste de 20 entiers inférieurs ou égaux à 5, tirés aléatoirement.

Exercice 10 | Tri par paquets *bucket sort* Solution On suppose que tous les éléments de la liste `L` à trier sont dans l'intervalle $I = [a, b[$, avec $a < b$ deux réels.

Le tri par paquets consiste à découper l'intervalle I en `len (L)` sous-intervalles de même longueur puis à répartir les données en paquets correspondant à chacun de ces sous-intervalles. On triera alors chacun de ces paquets à l'aide d'un des algorithmes de tri précédemment implémentés.

On peut montrer que ce tri est toujours plus efficace que le tri auxiliaire utilisé. En effet, dans le pire des cas tous les éléments sont dans un seul sous-intervalle, et donc cela revient à n'utiliser que le tri auxiliaire. On obtient donc un tri toujours au moins aussi efficace que l'autre.

Notons `h = (b-a)/len(L)`, `n = len(L)` et $I_0, \dots, I_{n-1}$ les sous-intervalles.

1. Écrire une fonction d'en-tête `calcul_interv(x, L)` qui étant donné un élément `x` de `L`, retourne l'entier `k` entre `0` et `n` dans lequel se trouve `x`
2. En déduire une fonction d'en-tête `tri_paquets(L)` implémentant l'algorithme de l'énoncé.

4.4. Comparaison des temps d'exécution

Nous allons dans cette partie mesurer empiriquement les temps d'exécution de chacun des tris.

Générer une liste de 10 entiers, et les mélanger

```
>>> import random as rd
>>> n = 10
>>> L = list(range(n))
>>> rd.shuffle(L) #Mélange des éléments de L en place
>>> L
[1, 2, 3, 9, 7, 0, 8, 5, 6, 4]
```

Exercice 11 | Comparaison des temps d'exécution Solution

1. Créer une fonction d'en-tête `genere_liste(n)` qui prend en entrée un entier n, et génère une liste d'entiers de $[\![1, n]\!]$ de manière aléatoire.
2. réer une fonction d'en-tête `eval_temps_tri(n, nom_du_tri)` qui prend en entrée un entier n, et un nom de fonction de tri, et retourne la moyenne des temps d'exécution pour ce tri, sur `10**2` tris de listes générés selon la première question.
3. Faire afficher les courbes des temps d'exécution moyens, sur `10**2` essais, pour chaque tri en fonction de la taille de la liste $n \in [\![1, 1000]\!]$ en utilisant le script ci-après. Commentez. *Inutile de chercher à comprendre cette fonction pour l'instant, nous ne savons pas encore utiliser* `matplotlib`. *Observez simplement le résultat.*

```
fonctions = [tri_insertion, tri_selection, tri_bulles,
 ↪  tri_rapide_rec] # vous pouvez ajouter ici d'autres tris
 ↪  traités dans les compléments

def trace_temps():
    X = list(range(1, 100))
    Y = []
    for nom_du_tri in fonctions:
        for n in X:
            Y.append(eval_temps_tri(n, nom_du_tri))
        plt.plot(X, Y, label = nom_du_tri)
        Y = []
    plt.legend()
```

5. SOLUTIONS DES EXERCICES

Solution (exercice 1)

Énoncé

```
def tri_test_croissant(L):
    """
    retourne True si L est triée par ordre décroissant, et
    False sinon
    """
    for i in range(len(L)-1):
        if L[i+1] < L[i]:
            return False
    return True

def tri_test_croissant(L):
    """
    retourne True si L est triée par ordre décroissant, et
    False sinon
    """
    for i in range(len(L)-1):
        if L[i+1] < L[i]:
            return False
    return True
```

Solution (exercice 2)

Énoncé

```
import random as rd

def tri_test_croissant(L):
    """
    retourne vrai/faux selon que L est triee par ordre croissant
    """
    for i in range(len(L)-1):
        if L[i+1] < L[i]:
            return False
    return True

def tri_stupide(L):
    """
    mélange les éléments de L aléatoirement jusqu'à obtenir la
↪   version triée
    """
    while not tri_test_croissant(L):
        rd.shuffle(L)
    return L
L = [3, 1, 7]
```

Par exemple, pour la liste précédente on obtient [1, 3, 7].

Solution (exercice 3)

Énoncé

Tri par insertion

```
def tri_insertion(L):
    """
    Retourne une version de L triée, selon le tri à insertion
    Tri non en place
    """
    L_tri = [L[0]]
    for k in range(1, len(L)):
        i = 0
```

```
        while (i < len(L_tri)) and (L_tri[i] < L[k]):
            i += 1
        # insertion a la bonne place dans Lla version triée
        L_tri.insert(i, L[k])
    return L_tri
```

Solution (exercice 4)

Énoncé

```
def minimum(L):
    """
    Cherche le minimum de L, et renvoie le premier indice
    Sert pour le tri par sélection du minimum
    """
    mini = L[0]
    ind_mini = 0
    for k in range(1, len(L)):
        if L[k] < mini:
            mini = L[k]
            ind_mini = k
    return mini, ind_mini

def tri_selection(L):
    """
    Trie la liste L en place selon le tri par sélection
    """
    for i in range(len(L)):
        # Recherche du minimum de la liste démarrée à i
        mini, ind_mini = minimum(L[i:])
        # On le place au début
        L[i], L[i + ind_mini] = L[i + ind_mini], L[i]
```

Solution (exercice 5)

Énoncé

```
def minimum(L):
    """
    Cherche le minimum de L, et renvoie le premier indice
    Sert pour le tri par sélection du minimum
    """
    mini = L[0]
    ind_mini = 0
    for k in range(1, len(L)):
        if L[k] < mini:
            mini = L[k]
            ind_mini = k
    return mini, ind_mini

def tri_selection_nn_en_place(L):
    """
    Trie la liste L en créant une copie selon le tri par sélection
    (du min)
    """
    L_trie = []
    for _ in range(len(L)):
        mini, ind_mini = minimum(L)
        L_trie.append(mini)
        del L[ind_mini]
    return L_trie
```

Solution (exercice 6)

Énoncé Le second programme est à associer à la seconde question.

```
def tri_rapide_rec(L):
    """
    Retourne une liste triée par ordre croissant des éléments de
 ↪  L,
    selon le tri rapide. Version récursive.
    Tri non en place
    """
    if L == []:
        return []
    pivot = L[0]
    inf_pivot = []
    sup_pivot = []
    for x in L[1:]:
        if x < pivot:
            inf_pivot.append(x)
        else:
            sup_pivot.append(x)
    return tri_rapide_rec(inf_pivot) + [pivot] +
     ↪  tri_rapide_rec(sup_pivot)

def tri_rapide_rec_aux(L):
    """
    Retourne une liste triée par ordre croissant des éléments de
 ↪  L,
    selon le tri rapide. Version récursive.
    Les listes inf/sup sont construites par compréhension ici
    Tri non en place
    """
    if L == []:
        return []
    pivot = L[0]
    inf_pivot = [L[i] for i in range(1, len(L)) if L[i] < pivot]
    sup_pivot = [L[i] for i in range(1, len(L)) if L[i] >= pivot]
    return tri_rapide_rec(inf_pivot) + [pivot] +
     ↪  tri_rapide_rec(sup_pivot)
```

Solution (exercice 7)

Énoncé

Médiane

```
from tri_rapide_rec import *
def mediane(L):
    """
    Cherche la médiane d'une liste, après tri rapide des
 ↪  observations
    """
    L = tri_rapide_rec(L)
    n = len(L)
    if n % 2 == 1:
        # Nombre impair d'observations, on prend le milieu
        return L[n//2]
    else:
        # Nombre pair d'observations, on prend la moyenne des deux
         ↪  termes du milieu
        return (L[n//2-1] + L[n//2])/2
```

Solution (exercice 8)

Énoncé

1.

```
def recherche_dicho(x, L):
```

```
        """
        recherche un élément par méthode dichotomique
        """
        L_tri = tri_rapide_rec(L)
        i_g = 0
        i_d = len(L)-1
        while i_g <= i_d:
                i_milieu = int((i_g + i_d)/2)
                if L[i_milieu] == x:
                        return True
                elif L[i_milieu] < x:
                        i_g = i_milieu+1
                else:
                        i_d = i_milieu-1
        return False # x non trouvé dans L
```

2.

```
def recherche_dicho_rec(x, L):
    """
    recherche récursivement un élément par méthode dichotomique
    """
    L_tri = tri_rapide_rec(L)
    ind_pivot = int(len(liste_triee)-1/2) # élément au milieu
    if liste_triee[ind_pivot] == x:
        return ind_pivot
    elif liste_triee[ind_pivot] > x:
        return recherche_dichotomique_rec(x,
        ↪ liste_triee[:ind_pivot])
    else:
        return ind_pivot + recherche_dichotomique_rec(x,
        ↪ liste_triee[ind_pivot:])
```

Solution (exercice 9)

Énoncé

1.

```
def comptage(L, N):
    """
    renvoie le tableau des effectifs des entiers entre 1 et N
    ↪ dans L
    """
    eff = [0 for _ in range(N+1)]
    for i in L:
        eff[i] += 1
    return eff
```

2. On construit la version triée de L en assemblant autant de copies nécessaires de chaque entier.

```
def tri_denombrement(L, N):
    """
    renvoie le tableau des effectifs des entiers entre 1 et N
    ↪ dans L
    """
    eff = comptage(L, N)
    L_trie = []
    for i in range(N):
        for _ in range(eff[i]):
            L_trie.append(i)
    return L_trie

L = [0, 3, 1, 4, 3, 2, 5, 1, 0]
```

Testons avec la liste précédente : [0, 0, 1, 1, 2, 3, 3, 4].

3. Testons la fonction tri sur une liste de 20 entiers inférieurs ou égaux à 5, tirés aléatoirement.

```
import random as rd
L = [rd.randint(0, 5) for _ in range(20)]
```

la fonction donne [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 4, 4, 4].

Solution (exercice 10)

Énoncé

Utilisons par exemple le tri rapide en tri auxiliaire.

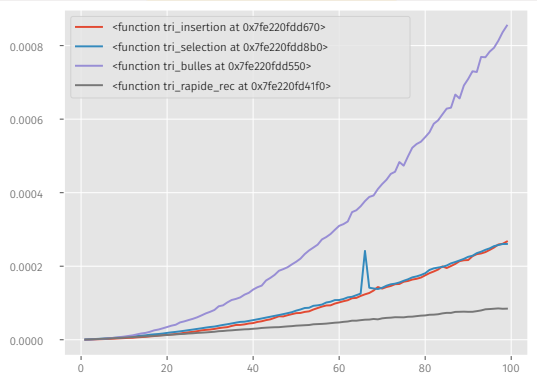
```
def calcul_interv(x, L, a, b):
    """
    retourne l'entier k de l'intervalle Ik associé
    """
    k = 0
    n = len(L)
    h = (b-a)/n
    while a+h*(k+1) < x:
        k += 1
    return k

def tri_paquets(L, a, b):
    """
    retourne L triée dans l'ordre croissant
    """
    n = len(L)
    L_blocs = [[] for _ in range(n)] # liste contenant les
    ↪ éléments regroupes en blocs
    for x in L:
        k = calcul_interv(x, L, a, b)
        L_blocs[k].append(x)
    L_tri = []
    for bloc in L_blocs:
        bloc_tri = tri_rapide_rec(bloc)
        for x in bloc_tri:
            L_tri.append(x)
    return L_tri
L = [3, 1, 7, 2]

>>> tri_paquets(L, 1, 7.1)
[1, 2, 3, 7]
```

Solution (exercice 11)

Énoncé

```
fonctions = [tri_insertion, tri_selection, tri_bulles,
↪ tri_rapide_rec]

import matplotlib.pyplot as plt
import random as rd

def genere_liste(n):
    """
    retourne une liste de 4 listes identiques d'entiers entre 1 et
↪ n
    """
    L = list(range(n))
    rd.shuffle(L)
    return L

import time as ti

def eval_temps_tri(n, nom_du_tri):
    """
    retourne le temps d'exécution moyen pour le tri nom_du_tri
```

```
    """
    somme = 0
    for _ in range(10**2):
        L = genere_liste(n)
        t_1 = ti.time()
        nom_du_tri(L)
        somme += ti.time()-t_1
    return somme/10**2

def trace_temps():
    X = list(range(1, 100))
    Y = []
    for nom_du_tri in fonctions:
        for n in X:
            Y.append(eval_temps_tri(n, nom_du_tri))
        plt.plot(X, Y, label = nom_du_tri)
        Y = []
    plt.legend()

trace_temps()
```



On constate que c'est le tri rapide, qui semble être le meilleur en terme de temps d'exécution.

Chapitre ALGO6.

Graphes

II

Deuxième partie

Numérique & Aléatoire

Chapitre NUM7.

Tableaux numpy & Images

1. SOLUTIONS DES EXERCICES

Chapitre NUM8.

Module matplotlib

Chapitre NUM9.

Méthodes numériques

Chapitre NUM10.

Probabilités & Statistiques

III

Troisième partie

Annexes

Chapitre ANN11.

Synthèse pour l'algorithmique

Résumé & Plan

Un condensé de commandes importantes et de scripts classiques à connaitre parfaitement pour les concours.

1. GÉNÉRALITÉS

1.1. Opérations élémentaires

Arithmétique

```
>>> 5**2 # puissance
25
>>> 5//2 # quotient division
2
>>> 5%2 # reste division
1
```

Gestion de variables

`x += 2` : ajoute 2 à la variable x
`x *= 2` : multiplie x par 2
`x, y = y, x` : échange x et y Les échanges fonctionnent également dans une liste, par exemple :

```
>>> L = [1, 2, 3]
>>> L[0], L[2] = L[2], L[0]
>>> L
[3, 2, 1]
```

1.2. Types

On rappelle que les types `tuple`, `str` par exemples sont immuables (impossible donc de modifier leurs éléments en place).

Types & Conversions

`type(x)` : type des de la variable x
`int` (entier), `float` (réel), `complex` (complexe)
`bool` (booléen : `True` ou `False`)
`list` (liste), `dict` (dictionnaire), `tuple` (tuple), `str` (chaîne de caractères), `array` (tableau),
Exemples de conversions de types.

```
>>> int(2.5)
2
>>> float(2)
2.0
>>> str(2)
'2'
>>> int("3")
3
```

Affichage et récupération de variables

`x = int(input("Valeur de x"))` : demander une valeur
`a,b,c = 1,2.5,"blabla"` : affectation multiple
`print("la valeur de x est", x)` : afficher, `del(x)` : effacer la variable x,

Manipulations de listes. La liste des fonctions/méthodes suivantes est à connaître.

Commande	Effet
`len(L)`	longueur
`L[0]`	premier élément
`L[-1]`	dernier élément
`L[i:j]`	liste extraite des éléments d'indices entre i (inclus) et j (exclus)
`L[i:]`	liste extraite à partir de l'indice i (inclus)
`L.append(v)`	ajoute l'élément v à la fin de la liste
`L.remove(v)`	supprimer le premier élément v apparaissant dans la liste,
	retourne une erreur s'il n'est pas présent
`L.extend(s)`	ajoute la liste s à la fin de la liste
`L.instert(i,v)`	insert l'objet v à l'indice i
`del L[i]`	supprime l'élément d'indice i
`L.pop()`	supprime le dernier élément et retourne l'élément supprimé

Manipulations de chaînes. La liste des fonctions/méthodes suivantes est à connaître.

Commande	Effet
`len(s)`	longueur
`s[0]`	premier élément
`s[-1]`	dernier élément
`s[i:j]`	chaîne extraite des éléments d'indices entre i (inclus) et j (exclus)
`s[i:]`	chaîne extraite à partir de l'indice i (inclus)
`s.join`	concatène à la chaîne de départ la nouvelle
`L = s.split()`	affecte à `L` la chaîne `s` dont les mots ont été séparés (et toute la chaîne si elle n'a pas de trou)

1.3. Structures de contrôle

TESTS Voici un bref panorama des test.

Tests : `==`, `!=`, `<=`, `>=`, `<`, `>`
Opérateurs logiques : **`and`**, **`or`**, **`not`**

Version courte

```
if test:
    instructions
    ...
```

Avec alternative

```
if test:
    instructions 1
    ...
else:
    instructions 2
    ...
```

Avec plusieurs tests

```
if test1:
    instructions 1
    ...
elif test2:
    instructions 2
    ...
else :
    instructions 3
    ...
```

BOUCLES.

Itérateurs sur des entiers/Listes d'entiers

`range(n)` : entiers entre 0 et $n-1$
`range(a,b)` : entiers entre a et $b-1$
`range(a,b,k)` : entiers de k en k entre a et $b-1$

Attention

On rappelle que les objets précédents ne sont pas des listes ! Pour les convertir en liste, on utilise `list(range(a,b))` par exemple.

Boucle for

```
for k in sequence:
    instructions
```

Boucle while

```
while test:
    instructions
```

Application : savoir calculer une somme, un produit, les termes d'une suite.

2. SCRIPTS USUELS

2.1. Sur les listes

Appartenance d'un élément dans une liste par balayage

```
def appartient(x, L):
    """
    retourne True si x est dans L
    """
    for y in L:
        if y == x:
            return True
```

```
    return False
```

Calcul d'une somme

```
def somme(L):
    """
    Retourne la somme des éléments d'une liste
    """
    S = 0
    for x in L:
        S += x
    return S
```

Calcul du maximum

```
def maximum(L):
    """
    Retourne la valeur du maximum de L
    """
    maxi = L[0]
    for k in range(1, len(L)):
        if L[k] > maxi:
            maxi = L[k]
    return maxi
```

Calcul du maximum + positions où il est réalisé

```
def maximum_occur(L):
    """
    Retourne le maximum de L, et renvoie la liste des occurences où
    il apparaît
    """
    maxi = L[0]
    ind_maxi = []
    for k in range(1, len(L)):
        if L[k] > maxi:
            maxi = L[k]
    for k in range(len(L)):
        if L[k] == maxi:
            ind_maxi.append(k)
    return maxi, ind_maxi

def maximum_occur_bis(L):
    """
    Retourne le maximum de L, et renvoie la liste des occurences où
    il apparaît. Un seul parcours de la liste ici.
    """
    maxi = L[0]
    ind_maxi = [0]
    for k in range(1, len(L)):
        if L[k] == maxi:
            ind_maxi.append(k)
        if L[k] > maxi:
            maxi = L[k]
            ind_maxi = [k]
    return maxi, ind_maxi
```

Décalage d'éléments

```
def decalage(L):
    """
    Décale les éléments de L vers la droite, ajoute zéro au début
    (par modif. d'une copie)
    """
    tete = L[0]
    del L[0]
    L.append(tete)
```

```python
def decalage_bis(L):
    """
    Décale les éléments de L vers la droite, ajoute zéro au début
    (par boucle for sur une copie)
    """
    M = [0 for _ in L]
    for k in range (len(L)):
        M[k-1] = L[k] # pour k=0 c'est le dernier élément M[-1]
    return M

def decalage_bisbis(L):
    """
    Décale les éléments de L vers la droite, ajoute zéro au début
    (par concaténation)
    """
    return L[1:len(L)] + [L[0]]
```

2.2. Sur les chaînes de caractère

Recherche d'un mot dans une chaîne

```python
def cherche_mot(m,s):
    """
    Recherche le mot m dans une chaîne s
    Renvoie un booléen
    """
    n = len(m) # Taille du mot
    N = len(s) # Taille de la chaine
    for k in range(N-n+1): # On parcourt les positions possibles
        if s[k:k+n] == m: # Si on trouve le mot a la position k
            return True
    return False
```

2.3. Sur les tris

Tri par sélection du minimum (non en place)

```python
def minimum(L):
    """
    Cherche le minimum de L, et renvoie le premier indice
    Sert pour le tri par sélection du minimum
    """
    mini = L[0]
    ind_mini = 0
    for k in range(1, len(L)):
        if L[k] < mini:
            mini = L[k]
            ind_mini = k
    return mini, ind_mini

def tri_selection_nn_en_place(L):
    """
    Trie la liste L en créant une copie selon le tri par sélection
    (du min)
    """
    L_trie = []
    for _ in range(len(L)):
        mini, ind_mini = minimum(L)
        L_trie.append(mini)
        del L[ind_mini]
    return L_trie
```

Tri par sélection du minimum (en place)

```python
def tri_selection(L):
    """
    Trie la liste L en place selon le tri par sélection
```

```
    """
    for i in range(len(L)):
        # Recherche du minimum de la liste démarrée à i
        mini, ind_mini = minimum(L[i:])
        # On le place au début
        L[i], L[i + ind_mini] = L[i + ind_mini], L[i]
```

Tri rapide

```
def tri_rapide_rec(L):
    """
    Retourne une liste triée par ordre croissant des éléments de
 ↪  L,
    selon le tri rapide. Version récursive.
    Tri non en place
    """
    if L == []:
        return []
    pivot = L[0]
    inf_pivot = []
    sup_pivot = []
    for x in L[1:]:
        if x < pivot:
            inf_pivot.append(x)
        else:
            sup_pivot.append(x)
    return tri_rapide_rec(inf_pivot) + [pivot] +
     ↪  tri_rapide_rec(sup_pivot)

def tri_rapide_rec_aux(L):
    """
    Retourne une liste triée par ordre croissant des éléments de
 ↪  L,
    selon le tri rapide. Version récursive.
    Les listes inf/sup sont construites par compréhension ici
    Tri non en place
    """
    if L == []:
        return []
    pivot = L[0]
    inf_pivot = [L[i] for i in range(1, len(L)) if L[i] < pivot]
    sup_pivot = [L[i] for i in range(1, len(L)) if L[i] >= pivot]
    return tri_rapide_rec(inf_pivot) + [pivot] +
     ↪  tri_rapide_rec(sup_pivot)
```

Tri par insertion

```
def tri_insertion(L):
    """
    Retourne une version de L triée, selon le tri à insertion
    Tri non en place
    """
    L_tri = [L[0]]
    for k in range(1, len(L)):
        i = 0
        while (i < len(L_tri)) and (L_tri[i] < L[k]):
            i += 1
        # insertion a la bonne place dans Lla version triée
        L_tri.insert(i, L[k])
    return L_tri
```

Tri à bulles

```
def tri_bulles(L):
    """
    Modifie la liste L pour la trier, selon le tri à bulles
    Tri en place
    """
```

```
    echange_fait = True
    while echange_fait == True:
        echange_fait = False
        for j in range(0, len(L)-1):
            if L[j] > L[j+1]:
                echange_fait = True
                L[j], L[j+1] = L[j+1], L[j]
```

Chapitre ANN12.

Synthèse pour le numérique

Résumé & Plan

Un condensé de commandes importantes axées ici sur le calcul numérique et la simulation.

1. GRAPHISMES

1.1. Généralités

On importe le sous-module `matplotlib.pyplot` avec

```
import matplotlib.pyplot as plt
```

Commande	Effet
`plt.plot(X, Y)`	Trace le nuage de points d'abscisses `X` et ordonnées `Y`
`plt.plot(X, Y, color = 'r', linewi`	Idem, mais en précisant une couleur, une épaisseur

Voici quelques autres options pour les paramètres optionnels de la command `plt.plot`

Propriétés	Rôle	Valeurs possibles
`color`	Trace le nuage de points	`'red'`, `'blue'` *etc.* d'abscisses `X` et ordonnées `Y`
`label`	Légende de la courbe	une chaîne
`linestyle`	type de ligne (pointillés, etc.)	`'-'`, `'-.'`, `':'`
`linewidth`	épaisseur du trait	un entier
`marker`	forme des points	`'+'`, `','`, `'o'`, `'1'`, `'2'` *etc.* (`'o'` adapté aux suites)

Commandes de formatage des titres, légendes, axes

```
plt.legend(loc = 'upper left') # permet de placer les légendes
plt.title('Titre') # permet de placer un titre
plt.axhline(color = 'black') # permet de rajouter un axe
 ↪ horizontal.
plt.axvline(color = 'black')
plt.axis(xmin = 0, xmax = 1.6, ymin = -1, ymax = 1)
plt.xlabel('x')
```

1.2. Tracé d'une suite ou d'une fonction

Soit f une fonction définie sur $[a, b]$, avec $a, b \in \mathbf{R}, a < b$, définie sur cet intervalle et que l'on souhaite tracer.

Tracer une fonction

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(a, b, 10**3) # création d'une subdivision des
 ↪ abscisses
Y = [f(x) for x in X] # les ordonnées
# OU, si et seulement si f ne contient que des fonctions numpy
Y = f(X)
plt.plot(X, Y, options)
plt.show()
```

Soit (u_n) une suite définie sur $\mathbf{N}$, que l'on souhaite tracer par exemple sur $[\![0, 10]\!]$. On suppose cette codée par une fonction Python notée u.

Tracer une fonction

```
import numpy as np
import matplotlib.pyplot as plt

X = list(range(0, 11))
Y = [u(n) for n in X] # les ordonnées
# OU : (si et seulement si u ne contient que des fonctions numpy)
Y = u(X)
plt.plot(X, Y, marker = 'o', autres options)
# OU puisque l'on veut tracer à partir de 0, on peut seulement
 ↪ écrire
plt.plot(Y, marker = 'o', autres options)
plt.show()
```

2. MÉTHODES NUMÉRIQUES

Méthode d'Euler

```
    """
    f : fonction, y0 : valeur en zéro, tau : borne max de
 ↪ l'intervalle,
    N : nombre de points de la subdivision->
    retourne un couple (abs, solution approchée)
    """
    h = tau/N
    T = np.linspace(0, tau, N+1)
    Y = np.zeros(N+1)
    Y[0] = y0
    for k in range(N):
        Y[k+1] = Y[k] + h * f(T[k], Y[k])
    return T, Y


import numpy as np
def euler_vec(f, y0, tau, N):
    """
```

```
    f : fonction, y0 : vecteur en zéro, tau : borne max de
↪   l'intervalle,
    N : nombre de points de la subdivision->
    retourne un couple (abs, solution approchée)
    solution approchée est une matrice (en chaque point de la
↪   subdivison, k
    coordonnées)
    """
    h = tau/N
    T = np.linspace(0, tau, N+1)
    Y = np.zeros((N+1, len(y0)))
    Y[0] = y0
    for k in range(N):
        Y[k+1] = Y[k] + h * f(T[k], Y[k])
    return T, Y
```

Dichotomie

```
def dicho(a, b, f, prec):
    """
    Retourne une valeur approchée d'un zéro de f entre a et b avec
↪   précision prec
    Retourne faux si aucune racine n'existe
    """
    while b - a > prec:
        c = (a + b)/2
        if f(a)*f(c) <= 0:
            # changement de signe sur [a,c]
            b = c
        else:
            # pas de changement de signe sur [a,c]
            a = c
    return (a + b)/2

def dicho_rec(a, b, f, prec):
    """
    Retourne une valeur approchée d'un zéro de f entre a et b avec
↪   précision prec
    selon un principe dichotomique
    Retourne faux si aucune racine n'existe
    """
    if f(a)*f(b) > 0:
        return False
    if b - a <= prec:
        return (a + b)/2
    else:
        c = (a + b)/2
        if f(a)*f(c) <= 0:
            # changement de signe sur [a,c]
            return dichotomie(a, c, prec)
        else:
            # pas de changement de signe sur [a,c]
            return dichotomie(c, b, prec)
```

Intégrales

```
def rectangle_RG(f, a, b, n):
    """
    Calcule la somme des rectangles gauche associée à f
    """
    S = 0
    h = (b-a)/n
    for i in range(n):
        S += f(a+h*i)
    return S*h

def rectangle_RD(f, a, b, n):
    """
    Calcule la somme des rectangles droite associée à f
    """
```

```
    S = 0
    h = (b-a)/n
    for i in range(1,n+1):
        S += f(a+h*i)
    return S*h
```

3. ALÉATOIRE & STATISTIQUES

On importe le module `random` avec

```
import random as rd
```

Commande	Effet
`rd.random()`	Tirer un nombre entre 0 et 1 selon une $\mathscr{U}[0,1]$
`rd.randint(i,j)`	Tirer un entier entre i et j selon une $\mathscr{U}[\![i,j]\!]$
`rd.choice(L)`	Tirer un élément au hasard dans `L`
`rd.shuffle(L)`	Mélange les éléments de `L` de manière aléatoire

3.1. Simulations de lois classiques

Lois discrètes

```
import numpy as np
import random as rd
def unif_entier(a,b):
    """
    Renvoie une simulation de l'uniforme sur [a,b] entier
    """
    return np.floor(a+(b+1-a)*rd.random())


import random as rd
def bernoulli(p):
    """
    simule une bernoulli
    """
    if rd.random() < p:
        return 1
    else:
        return 0

def binomiale(n,p):
    """
    simule une binomiale
    """
    S = 0
    for i in range(n):
        if rd.random() < p:
            S += 1
    return S


import random as rd
def hypergeometrique(N, n, p):
    """
    simule une hypergeometrique
    N : nombre total d'éléments
    n : nombre d'éléments piochés
    p : proportion éléments type 1
    """
    Nb_type1 = N * p
    Nb_type2 = N - Nb_type1
```

```
    S = 0
    for i in range(n):
        if rd.random() < p:
            Nb_type1 -= 1
            S += 1
        else:
            Nb_type2 -= 1
        p = Nb_type1/(Nb_type1 + Nb_type2) # proportion boules de
        ↪ type 1
    return S

import random as rd
def geometrique(p):
    """
    simule une geometrique
    """
    S = 1
    while rd.random() > p:
        S += 1
    return S


import random as rd
def poisson(lamb):
    """
    Simule une loi de Poisson par inversion
    """
    U = rd.random()
    F = 0
    i = 0
    while F < U:
        i += 1
        F += np.exp(-lamb)*lamb**i/ma.factorial(i)
    return i-1
```

Lois à densité

```
import random as rd
def unif_reel(a,b):
    """
    Renvoie une simulation de l'uniforme sur [a,b]
    """
    return a+(b+1-a)*rd.random()

import random as rd
from math import log
def expo(lamba):
    """
    lamba -> une simulation de l'exponentielle de paramètre lamba
    """
    return -(1/lamba)*log(rd.random())

    ->une simulation de la N(0,1)
    """
    return stat.norm.ppf(rd.random())
```

3.2. Statistiques descriptives

Calculs de grandeurs statistiques

```
def moyenne(L):
    """
    Renvoie l'espérance
    """
    S = 0
    for x in L:
        S += x
    return S/len(L)
```

```python
def etendue(L):
    """
    Cherche l'étendue de L, sans la trier
    """
    min = L[0]
    max = L[0]
    for i in range(1, len(L)):
        if L[i] < min:
            min = L[i]
        elif L[i] > max:
            max = L[i]
    return max - min

from tri_rapide_rec import *
def mediane(L):
    """
    Cherche la médiane d'une liste, après tri rapide des
 ↪  observations
    """
    L = tri_rapide_rec(L)
    n = len(L)
    if n % 2 == 1:
        # Nombre impair d'observations, on prend le milieu
        return L[n//2]
    else:
        # Nombre pair d'observations, on prend la moyenne des deux
         ↪  termes du milieu
        return (L[n//2-1] + L[n//2])/2

from moyenne import *
def variance(L):
    """
    Renvoie la variance, version KH
    """
    esp = moyenne(L)
    V = 0
    for x in L:
        V += x**2
    return V/len(L) - esp**2

def covariance(L, M):
    """
    Renvoie la covariance des deux séries
    """
    Prod = [L[i]*M[i] for i in range(len(M))]
    return esperance(Prod) - esperance(L)*esperance(M)
```

Chapitre ANN13.

Fichiers

Résumé & Plan

L'objectif de ce chapitre est de présenter brièvement comment importer des données extérieures dans son fichier Python principal. Ces données peuvent être dans un autre fichier `.py`, ou bien un fichier texte `.txt` ou encore une base donnée contenue dans un fichier `.csv`.

1. FICHIER .PY : IMPORTATION DE MODULE

Si l'on souhaite importer les données contenus dans un fichier python auxiliaire `fichier_aux.py` contenant

```
import numpy as np
CHAINE = "bonjour les BCPST1"
```

dans un fichier principal, on place le fichier `fichier_aux.py` dans le même dossier que le fichier principal et on tape dans le fichier principal

```
>>> from fichier_aux import *
```

Dans le fichier principal, on voit que tout a été correctement importé.

```
>>> np.sin # le module numpy a donc correctement été importé par
 ↪  la même occasion
<ufunc 'sin'>
>>> CHAINE # la variable CHAINE existe bien dans le fichier
 ↪  principal
'bonjour les BCPST1'
```

2. FICHIER .CSV

Pour échanger, partager et analyser les données collectées lors d'une expérience, elles doivent être enregistrées dans un fichier au format informatique ouvert. Le plus utilisé est le format de fichier CSV (**C**omma-**S**eparated **V**alues).

Le module `csv` contient des outils permettant de lire et manipuler des fichiers dont le format est `CSV`. On l'importe de la manière classique suivante.

```
>>> import csv
```

Dans la suite, nous allons manipuler la base de données stockée dans le fichier `.csv` disponible au lieu ci-après :

https://opendata.bordeaux-metropole.fr/explore/dataset/bor_arbres/export/

Ensuite, on doit au choix :

- déposer le fichier de données dans le même répertoire que le fichier Python principal (solution à préférer), puis exécuter **en tant que script**
- ou bien utiliser le chemin absolu du fichier dans la suite. Pour le connaître *via* Pyzo, il suffit de glisser déspoer ledit fichier dans la fenêtre de la console, puis indiquer ce chemin complet dans la commande `open` ci-après.

Supposons que la première option a été choisie. On commence par ouvrir le fichier en mode lecture, puis à le décoder en précisant le délimiteur (le symbole séparant chaque ligne de la base de données).

```
fichier = open('nomdufichier.csv',"r")
data = csv.reader(fichier,delimiter=";")
```

Donc dans notre cas

```
>>> fichier = open('bor_arbres.csv',"r")
>>> data = csv.reader(fichier, delimiter=";")
```

On constate que `data` possède un type particulier.

```
>>> type(data)
<class '_csv.reader'>
```

Mais c'est un type itérable, au sens où l'on peut parcourir chacune des lignes à l'aide d'une simple boucle `for`. Dans l'exemple qui suit, on l'arrête au bout de 3 itérations, car la base est très grande.

```
>>> N = 0
>>> for ligne in data:
...     print(ligne)
...     N += 1
...     if N == 3:
...         break
...
['Geo Point', 'Geo Shape', 'gid', 'geom_o', 'geom_err',
 ↪ 'localisation', 'tranche_age', 'nom', 'circonference',
 ↪ 'hauteur', 'typo_espace', 'statut', 'diametre', 'famille',
 ↪ 'genre', 'variete', 'geographie', 'date_plantation', 'cdate',
 ↪ 'mdate', 'age_tranche_basse']
['44.8451436,-0.5719386', '{"type": "Point", "coordinates":
 ↪ [-0.5719386, 44.8451436]}', '3034', '0', '', 'des Quinconces
 ↪ (place)', '49 - 74', 'Platanus x hispanica', '147', '20',
 ↪ 'ESPACE_PUBLIC_VEGETALISE', 'VIVANT', '47', 'Platanaceae',
 ↪ 'Platanus sp.', '', 'Hybride Platanus orientalis x P.
 ↪ occidentalis', '', '2019-04-02T00:00:00+02:00',
 ↪ '2019-04-02T00:00:00+02:00', '49']
['44.8451677,-0.5716563', '{"type": "Point", "coordinates":
 ↪ [-0.5716563, 44.8451677]}', '3036', '0', '', 'des Quinconces
 ↪ (place)', '83 - 124', 'Platanus x hispanica', '248', '21',
 ↪ 'ESPACE_PUBLIC_VEGETALISE', 'VIVANT', '79', 'Platanaceae',
 ↪ 'Platanus sp.', '', 'Hybride Platanus orientalis x P.
 ↪ occidentalis', '', '2021-05-20T00:00:00+02:00',
 ↪ '2021-05-20T00:00:00+02:00', '83']
```

La première ligne correspond donc aux champs de la base, le premier champ est donc le lieu géographique de plantation. Si l'on souhaite par exemple afficher l'ensemble des coordonnées uniquement, on fera :

```
>>> N = 0
>>> for ligne in data:
...     print(ligne[0])
...     N += 1
...     if N == 3:
...         break
```

Une fois nos données récupérées, on ferme le fichier.

```
>>> fichier.close()
```

3. FICHIER .TXT

On souhaite ici importer un texte long contenu dans `texte.txt`. Cela va globalement se passer comme avec les csv, mais sans module supplémentaire. On suppose qu'un fichier `texte.txt` contenant :

283 133 47 30 21 16 13 : les températures sur une année 1000 855 798
760 732 710 692 : les températures sur l'année suivante

est présent dans le même répertoire que le fichier principal. Ou alors utiliser un chemin complet du texte comme déjà dit avant.

```
>>> fichier = open("texte.txt", "r")
>>> fichier.read() # convertit le tout en une grosse chaîne
"283 133 47 30 21 16 13 : les températures sur une année\n1000 855
 ↪  798 760 732 710 692 : les températures sur l'année suivante"
>>> fichier.close()
```

Sauts de ligne sont codés par la chaîne \n. On peut aussi préférer lire le fichier ligne par ligne

```
>>> fichier = open("texte.txt", "r")
>>> ligne_1 = fichier.readline()
>>> ligne_1
'283 133 47 30 21 16 13 : les températures sur une année\n'
>>> ligne_2 = fichier.readline()
>>> ligne_2
"1000 855 798 760 732 710 692 : les températures sur l'année
 ↪  suivante"
>>> fichier.close()
```

On peut également lire toutes les lignes d'un coup, et créer ainsi une liste des lignes.

```
>>> fichier = open("texte.txt", "r")
>>> lignes = fichier.readlines()
>>> lignes
['283 133 47 30 21 16 13 : les températures sur une année\n',
 ↪  "1000 855 798 760 732 710 692 : les températures sur l'année
 ↪  suivante"]
>>> fichier.close()
```

Chapitre ANN14.

Erreurs courantes en Python

Résumé & Plan

L'objectif de ce chapitre est le suivant : vous permettre d'analyser et de comprendre les erreurs que vous obtenez en Python. «Monsieur, ça ne marche pas» n'est à prononcer qu'après l'analyse de l'erreur.

Avec Python, lorsque le lancement d'une instruction ou d'un programme provoque une erreur, un message s'affiche en rouge. Ce message vous informe :

- de l'endroit où l'erreur s'est produite dans le programme (numéro de la ligne dans le programme),
- de la cause de l'erreur.

Concernant cette dernière information, le message paraîtra parfois obscur pour un non-anglophone. Il est donc important que vous puissiez comprendre par vous-même ce qu'il faut corriger à votre programme pour qu'il tourne correctement.

- `SyntaxError` regroupe toutes les situations où ce que vous avez demandé n'a pas de sens en Python. Cela peut venir d'une parenthèse en trop, d'un crochet au lieu d'une parenthèse, d'une virgule au lieu d'un point …. Exemple :

```
((1+2]
```

- `IndentationError: unexpected indent` signifie qu'il y a un problème d'alinéa. En général, une ou plusieurs lignes ne sont pas alignées correctement avec leur bloc. Il suffit d'un espace en trop !

```
x = 1 #BIEN
  x = 1 #PAS BIEN
```

- `IndexError: list index out of range` signifie que vous avez demandé d'extraire un élément d'une liste avec une position plus grande que la taille de la liste.

```
L = [2,3,9]
L[5]
```

- `TypeError: list indices must be integers or slices, not float` même chose mais si vous demandez une position qui n'est pas un nombre entier.

```
A = [1, 8, 4]
A[3.3]
```

- `NameError: name 'BCPST' is not defined` signifie que vous demandez à Python d'utiliser une variable alors que celle-ci n'existe pas. Souvent, ça arrive quand on a mal recopié le nom de notre variable.

```
BCPST *= 2
BCPST *= 8
```

Ca peut arriver aussi si on utilise une fonction d'une bibliothèque qu'on a oublié d'importer au préalable. Par exemple si on fait :

```
a = cos(pi)
# sans avoir fait from math import *
```

- `ZeroDivisionError : float division by zero` assez facile à comprendre … vous avez demandé de faire une division par zéro.
- `TypeError: ... object is not callable` en général signifie que vous avez pris pour une fonction une variable qui n'en était pas une. *To call a function* signifie «exécuter une fonction». Seules les variables de type **function** peuvent être exécutées, ceci avec la syntaxe `f(x)`. Par conséquent si `f` n'est pas une fonction ça fait une erreur.

```
a = 1
a(3)
```

Cela arrive aussi si vous avez oublié un signe * pour multiplier une parenthèse. Exemple :

```
2(3+8)
```

- `TypeError: ... object is not subscriptable ...` Même chose avec les crochets. La syntaxe `L[2]` sert à extraire l'élément en position 2 de `L`. Quand `L` n'est pas une liste ou une chaîne de caractère cela posera problème. Exemple :

```
L = 3
L[2]
```

- `OSError: [Errno 2] No such file or directory: 'C:/photos/vacances_Blanquer/ibiza.jpg'` ceci arrive quand vous cherchez à importer un fichier qui n'existe pas. Vérifiez alors que votre fichier est bien situé là où vous croyez, et qu'il n'y a pas d'erreur dans l'orthographe des dossiers, sous-dossiers, du fichier.

Chapitre ANN15.

Bonnes pratiques en Python

Résumé & Plan

Ce chapitre est transverse à tous les autres, et est d'une importance capitale pour la rédaction de vos projets. Le jury est attentif à tous ses éléments de présentation du code, ne le négligez donc pas. Il faut garder la chose suivante en tête : les examinateurs ont énormément de projets à lire pour l'oral, plus le vôtre sera clair et compréhensible, plus vous aurez de chance d'être compris. Pour ce faire, il y a un certain nombre de conseils simples à prendre en compte.

Code is read much more often than it is written.

— Guido van Rossum

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

— Martin Fowler

Comme vous l'avez constaté dans tous les chapitres précédents, la syntaxe de Python est très permissive. Afin d'uniformiser l'écriture de code en Python, la communauté des développeurs Python recommande un certain nombre de règles afin qu'un code soit lisible. Lisible par quelqu'un d'autre, mais également, et surtout, par soi-même. Essayez de relire un code que vous avez écrit «rapidement» il y a un 1 mois, 6 mois ou un an. Si le code ne fait que quelques lignes, il se peut que vous vous y retrouviez, mais s'il fait plusieurs dizaines voire centaines de lignes, vous serez perdus. Avec l'expérience, vous vous rendrez compte que cela est parfaitement vrai. Alors plus de temps à perdre, voyons en quoi consistent ces bonnes pratiques.

*Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts. Special cases aren't special enough to break the rules. Although practicality beats purity. Errors should never pass silently. Unless explicitly silenced. In the face of ambiguity, refuse the temptation to guess. There should be one– and preferably only one –obvious way to do it. Although that way may not be obvious at first unless you're Dutch. Now is better than never. Although never is often better than *right* now. If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea. Namespaces are one honking great idea – let's do more of those!*

Un condensé de toutes les règles ci-dessous peut être consulté ici :
https://larlet.fr/david/biologeek/archives/20080511-bonnes-pratiques-et-astuces-python/

1. LA PEP8

Plusieurs choses sont nécessaires pour écrire un code lisible : la syntaxe, l'organisation du code, le découpage en fonctions, mais souvent, aussi, le bon sens. Pour cela, les PEP (comme « Python Enhancement Proposal »), qui sont des agrégats de propositions dont la plus connue est la PEP8. Voici une synthèse des principales préconisations.

1.1. Indentation

Déjà constaté en première année : l'indentation des programmes est un incontournable des scripts en Python. Cela vient d'un constat simple, l'indentation améliore la lisibilité d'un code. Dans la PEP8, la recommandation pour la syntaxe de chaque niveau d'indentation est très simple : **4 espaces**. N'utilisez pas autre chose, c'est le meilleur compromis.

Attention

Cela ne correspond pas nécessairement à la touche de tabulation de votre clavier, mais c'est le cas dans Pyzo.

1.2. Importations de modules

- Le chargement d'un module se fait avec l'instruction `import module` ou éventuellement avec un préfixe `import module as blabla` plutôt qu'avec `from module import *`.
- L'importation doit également se faire par ordre alphabétique de nom.

Si on souhaite ensuite utiliser une fonction d'un module, la première syntaxe conduit à `module.fonction()` ce qui rend explicite la provenance de la fonction. Avec la seconde syntaxe, il faudrait écrire `fonction()` ce qui peut causer divers problèmes (conflits notamment).

1.3. Règles de nommage

- Les `noms_de_variable` doivent être écrites en minuscules, avec éventuellement le symbole underscore comme séparateur de blocs de noms, avec un nombre de caractère petit (pour les lire facilement).
- Les `CONSTANTES` globales sont écrites en majuscules, avec éventuellement le symbole underscore comme séparateur de blocs de noms, avec un nombre de caractère raisonnable (plus facile à retenir).
- Nommage des variables globales : utiliser des noms explicites dans votre code plutôt que le codage utilisé.

Par exemple :

```
ma_variable
fonction_test_27()
```

```
mon_module
VITESSE_LUMIERE = ...
```

On essaiera toujours de donner des noms de variable explicites : il est complètement inutile de perdre de l'énergie à répondre en permanence à des questions du type « que représente telle ou telle variable » ? Ainsi : dans le cas d'un plateau de jeu recouvert de pions blancs (codés par des 0) et noirs (codés par des 1), utiliser en préambule

```
NOIR = 1
BLANC = 0
```

et utiliser `NOIR`, `BLANC` plutôt que `1`, `0` dans tout le projet.

1.4. Espacement

- La PEP 8 recommande d'entourer les opérateurs (+, -, /, *, ==, !=, >=, **not**, **in**, **and**, **or**...) d'un espace avant et d'un espace après.

Par exemple :

```
# code recommandé :
ma_variable = 3+7
mon_texte = "souris"
mon_texte == ma_variable
# code non recommandé :
ma_variable = 3+7
mon_texte = "souris"
mon_texte == ma_variable
```

Mais inutile d'aligner pour « faire joli ».

```
# code recommandé :
x1 = 1
x2 = 3
x_old = 5
# code NON recommandé :
x1    = 1
x2    = 3
x_old = 5
```

Et c'est tout. On n'espace pas entre un nom de fonction et son argument par exemple, dans l'indexage des éléments d'une liste, *etc.*.

- On met un espace après les caractères « : » et « , » (mais pas avant).

1.5. Aération & Retours à la ligne

Aération. Dans un script, les lignes vides sont utiles pour séparer visuellement les différentes parties du code.

Il est recommandé de laisser deux lignes vides avant la définition d'une fonction ou d'une classe et de laisser une seule ligne vide avant la définition d'une méthode (dans une classe).

On peut aussi laisser une ligne vide dans le corps d'une fonction pour séparer les sections logiques de la fonction, mais cela est à utiliser avec parcimonie.

Retour à la ligne. Pyzo matérialise une ligne sur la droite, qu'il convient de ne pas dépasser pour des questions de lisibilité. Bien entendu, comme sur l'image ci-dessous, des très légers débordements à droite peuvent être tolérés.

```
def Kmeans(cellule_1,cellule_2,distance):
    """
    Cette fonction sépare la liste de cellules en deux groupes à partir
    d'une cellule de type 2 et une cellule de type 4
    """
    Barycentre_1_i = cellule_1
    Barycentre_2_i = cellule_2

    Barycentre_1_j = calcul_barycentre(cellule_1,cellule_2,0,distance)
    Barycentre_2_j = calcul_barycentre(cellule_1,cellule_2,1,distance)

    while distance(Barycentre_1_j,Barycentre_1_i) > 1 and \
        distance(Barycentre_2_j,Barycentre_2_i) > 1 :

        Barycentre_1_i = Barycentre_1_j
        Barycentre_2_i = Barycentre_2_j

        Barycentre_1_j = calcul_barycentre(Barycentre_1_j,Barycentre_2_j,0,distance)
```

FIG. ANN15.1. : Réglage de longueur de lignes sour Pyzo

- Tout code qui dépasse cette ligne doit être coupé, pour des questions de lisibilité évidente.[a]

[a] Votre examinateur n'aura aucune envie de *«scroller»* de gauche à droite (en plus de haut en bas)

1.6. Commentaires

Les commentaires débutent toujours par le symbole `##` **suivi d'un espace**. Ils donnent des explications claires sur l'utilité du code et doivent être synchronisés avec le code, c'est-à-dire que si le code est modifié, les commentaires doivent l'être aussi (le cas échéant).

- Les commentaires sont sur le même niveau d'indentation que le code qu'ils commentent. Les commentaires sont constitués de phrases complètes, avec une majuscule au début (sauf si le premier mot est une variable qui s'écrit sans majuscule) et un point à la fin.
- Les commentaires qui suivent le code sur la même ligne sont à éviter le plus possible et doivent être séparés du code par au moins deux espaces.

Remarque 1 (Risques d'obscurcir le programme)
- En ajoutant du verbiage (paraphrase),
- en dispensant le programmeur d'écrire proprement,
- en induisant le lecteur en erreur.

Par exemple, est-ce pertinent d'écrire cela ?

```
x += 1 # on augmente x, COMMENTAIRE TOTALEMENT INUTILE
```

- Un commentaire n'est pas là pour expliquer une notion de Python, mais pour faire gagner du temps au lecteur de votre code. Il n'a de l'intérêt que s'il est nécessaire de devoir expliquer quelque chose à l'endroit donné.

Un commentaire peut être également utilisé pour expliquer, par exemple, les structures de données utilisées dans votre projet. Par exemple en début de projet :

```
# Les coordonnées sur l'échiquier seront représentées par
# des couples (i, j) (numéro de ligne, numéro de colonne), # la
 ↪ numérotation commençant à zéro.
# On représente la solution en cours de construction par un #
 ↪ tableau t de longueur au plus N donnant les coordonnées
# de chaque dame déjà placée.
```

2. AUTRES RECOMMANDATIONS

2.1. À propos des conditionnelles

SIMPLICITÉ DES `return`. La gestion des booléens doit être optimale. En particulier :

- Les booléens sont des objets comme les autres.
- `return x > 2` doit être aussi naturel et habituel que `return x + 2`.
- Ainsi, ce type de chose est à éviter :

```
if x > 0:
    return True
else:
    return False
```

Utiliser plutôt : `return x > 0`.

2.1.1. Imbrication

- Éviter d'imbriquer des conditionnelles. En particulier un `if` juste après un `if`, `elif` ou `else` doit être changé.

Par exemple :

```
if i==1:
if j==1: action(1) elif j==17: action(2) else: action(3)
elif i==17:
if j==1: action(4) elif j==17: action(5) else: action(6)
else:
if j==1: action(7) elif j==17: action(8) else: action(9)
```

doit **être changé en**

```
if (i, j) == ( 1, 1):
    action(1)
elif (i, j) == ( 1, 17):
    action(2)
elif i == 1 :
    action(3)
elif (i, j) == (17, 1):
    action(4)
elif (i, j) == (17, 17):
    action(5)
else:
    action(6)
```

3. LES DOCSTRINGS ET LA PEP 257

De manière générale, écrivez des docstrings pour les modules, les fonctions, les classes et les méthodes. Lorsque l'explication est courte et compacte comme dans certaines fonctions ou méthodes simples, utilisez des docstrings d'une ligne. Les éléments que vous allez renseigner dans la docstring sont d'importance capitale pour la personne qui va exécuter votre programme (le jury de projets pour vous en l'occurence). Elles comprennent notamment :

1. ce que fait la fonction ou la méthode,
2. ce qu'elle prend en argument,
3. ce qu'elle renvoie.

Il existe certaines règles pour la rédaction de ces aides, mais nous n'irons pas jusque là. Voici un titre informatif un exemple de «bon formatage» de docstring pour une fonction :

Version longue

```
def exemple_fonction(parametres):
    """
    Ce que fait la fonction
    Parameters
    ----------
    par1 : type de par1
    par2 : type de par2

        Avec une description plus longue.
        Sur plusieurs lignes.

    Returns
    -------
    type1
        Le return associé à type 1

    """
```

```
    corps de la fonction
```

Mais la plupart du temps, pour les fonctions de taille moyenne et petites, nous utiliserons plutôt la version courte ci-dessous.

Version courte (à utiliser)

```
def exemple_fonction(parametres):
    """
    parametres -> ce qu'elle renvoie
    """
    corps de la fonction
```

4. OUTIL DE VÉRIFICATION DE CODE

Il existe différentes méthodes pour cela. Je vous conseille de la faire de manière régulière pendant toute la rédaction de votre projet, en utilisant l'outil en ligne suivant :

http://pep8online.com

On copie-colle ensuite le code à vérifier. On peut aussi directement uploader un fichier. Le programme se charge de vérifier la typographie de votre code.