Introduction à Python

Romain Tavenard

sept. 17, 2023

CONTENTS

Ce document est un polycopié qui sert de support pour certains cours enseignés dans la filière MIASHS de l'Université de Rennes 2. Il est distribué librement (sous licence CC BY-NC-SA plus précisément) et se veut évolutif, n'hésitez donc pas à faire vos remarques à son auteur dont vous trouverez le contact sur sa page web. Ce polycopié a notamment bénéficié des apports d'Aurélie Lemaitre, d'Agnès Maunoury et de Jean-Christophe Burnel.

Durant la lecture de ce polycopié, vous trouverez des blocs de code tels que celui-ci :

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

x = 5
y = f(3 * x + 2)
print(y)
```

```
289
```

Le bloc situé sous le code correspond à la sortie générée dans le terminal par le code en question.

Dans ce document, nous allons donc nous intéresser au langage Python. Pour tester les exemples présentés au fil de ce document ou réaliser les exercices proposés, vous aurez deux possibilités. La première consiste à ouvrir une **console Python**, à l'aide de la commande suivante (si vous êtes sous Unix, en supposant que le symbole `$` corresponde au prompt de votre *shell*) :

```
$ python
Python 3.5.1 (default, Dec  9 2015, 11:28:16)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.1.76)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Lors de l'exécution de cette commande, on peut remarquer plusieurs choses. Tout d'abord, au démarrage, la console Python nous indique la version de Python qui est exécutée. Cela est important, car il existe notamment une importante différence entre les versions 2 (2.x.y) et 3 (3.x.y) de Python. Dans ce document, nous supposons l'utilisation de Python dans sa version 3, comme dans la console affichée plus haut. Enfin, une fois la console démarrée, on voit apparaître un prompt Python (`>>>`) qui indique que vous pouvez, à partir de ce point, entrer du code Python et en demander l'exécution en appuyant sur la touche *retour chariot* (ou « Entrée ») de votre clavier.

L'autre façon de programmer en Python, plus adaptée dès lors que l'on souhaite conserver une trace de ce qu'on a écrit, consiste à enregistrer vos commandes dans un fichier texte (en respectant la convention qui consiste à utiliser l'extension `.py` pour le nom de fichier) puis à faire exécuter votre programme par l'interpréteur Python :

```
$ python nom_de_mon_fichier.py
[...]
```

CHAPTER 1

STRUCTURES DE DONNÉES ET STRUCTURES DE CONTRÔLE

Dans ce chapitre, on s'intéresse aux éléments de base de la syntaxe Python : les structures de données d'une part et les structures de contrôle d'autre part. Les structures de données vont permettre de stocker dans la mémoire de l'ordinateur (dans le but de les traiter ensuite) des données tandis que les structures de contrôle vont servir à définir nos interactions avec ces données.

1.1 Variables

En Python, les données sont stockées dans des variables. On ne peut pas, comme c'est le cas dans d'autres langages, définir de constante (qui sont, dans ces langages, des moyens de stocker des valeurs n'ayant pas vocation à être modifiées au cours de l'exécution du programme). Une variable est une association entre un symbole (le nom de la variable) et une valeur, cette dernière pouvant varier au cours de l'exécution du programme.

1.1.1 Types des variables Python

Les types de base existant en Python sont les suivants :

- `int` : entier ;
- `float` : nombre à virgule ;
- `complex` : nombre complexe (peu utilisé en pratique dans ce cours) ;
- `str` : chaîne de caractères ;
- `bool` : booléen (pouvant prendre les valeurs `True` ou `False`).

De plus, il existe un type spécial (`NoneType`) ne permettant qu'une seule valeur : la valeur `None` qui signifie « pas de valeur » ou « valeur manquante ».

Le choix du type utilisé pour une variable impliquera :

- une certaine façon d'encoder les données en mémoire (mais cette partie vous sera largement masquée, c'est l'interpréteur qui s'en chargera) ;
- un certain nombre d'opérations autorisées sur la variable (opérations arithmétiques sur les variables numériques, concaténation sur les chaînes de caractères, *etc.*)

Les variables Python sont typées dynamiquement, ce qui signifie qu'une variable, à un moment donné de l'exécution d'un programme, a un type précis qui lui est attribué, mais que celui-ci peut évoluer au cours de l'exécution du programme. En Python, le type d'une variable n'est pas déclaré par l'utilisateur : il est défini par l'usage (la valeur effective que l'on décide de stocker dans la variable en question).

Par exemple, l'instruction suivante (dite opération d'affectation) en Python attribue la valeur `12` à la variable `v`, qui devient donc automatiquement de type entier :

```
v = 12
```

Ainsi, les instructions suivantes ont toutes une incidence sur le type des variables considérées :

```
v = 12      # v est alors de type entier
c = "abc"   # c est de type chaîne de caractères
d = 'abc'   # d est également de type chaîne de caractères
            # les contenus de c et d sont identiques
v = 12.     # v change de type et est désormais de type nombre à virgule
```

Pour vérifier le type d'une variable, il suffit d'utiliser la fonction `type` de la librairie standard :

```
print(type(v))   # la fonction print(.) permet d'afficher
                 # une information dans le terminal
```

```
<class 'float'>
```

1.1.2 Opération d'affectation

Comme le montrent les exemples précédents, pour pouvoir utiliser des variables, on doit leur donner un nom (placé à gauche du signe égal dans l'opération d'affectation). Ces noms de variables doivent respecter certaines contraintes :

- ils doivent débuter par une lettre (minuscule ou majuscule, peu importe) ou par le symbole _ ;
- ils ne doivent contenir que des lettres, des chiffres et des symboles _ ;
- ils ne doivent pas correspondre à un quelconque mot réservé du langage Python, dont voici la liste :

```
and del for is raise assert elif from lambda return break else global
not try nonlocal True False class except if or while continue import
pass yield None def finally in as with
```

- ils ne doivent pas correspondre à des noms de fonction de la librairie standard de Python (cette dernière condition n'est en fait qu'une bonne pratique à observer) : vous apprendrez au fur et à mesure les noms de ces fonctions.

Les noms de variable en Python sont sensibles à la casse, ainsi les variables `maVariable` et `mavariable` ne pointent pas sur les mêmes données en mémoire. Pour s'en convaincre, on peut exécuter le code suivant :

```
mavariable = 12
maVariable = 15
print(mavariable)
```

```
12
```

```
print(maVariable)
```

```
15
```

Comme on l'a vu plus haut, on utilise en Python l'opérateur = pour affecter une valeur à une variable. La sémantique de cet opérateur est la suivante : « affecter la valeur contenue dans le membre de droite à la variable du membre de gauche ». Ainsi, il est tout à fait valide d'écrire, en Python :

```
x = 3.9 * x * (1 - x)
```

Pour exécuter cette instruction, l'interpréteur Python commencera par évaluer le membre de droite en utilisant la valeur courante de la variable `x`, puis affectera la valeur correspondant au résultat de l'opération `3.9 * x * (1 - x)` dans la variable `x`.

Ainsi, voici le résultat de l'exécution suivante :

```
x = 2
print(x)
```

```
2
```

```
x = 3.9 * x * (1 - x)
print(x)
```

```
-7.8
```

Si l'on souhaite obtenir un affichage plus riche, on pourra utiliser la méthode `format` comme suit :

```
x = 2
x = 3.9 * x * (1 - x)
print("La valeur courante de x est {}".format(x))
```

```
La valeur courante de x est -7.8
```

1.1.3 Opérateurs et priorité

On le voit dans l'exemple précédent, pour manipuler des variables, on utilisera des opérateurs (dont les plus connus sont les opérateurs arithmétiques). Le tableau suivant dresse une liste des opérateurs définis pour les variables dont le type est l'un des types numériques (entier, nombre à virgule, nombre complexe) :

Opérateur	Opération
+	Addition
-	Soustraction
*	Multiplication
/	Division
**	Élévation à la puissance
%	Modulo (non défini pour les nombres complexes)
//	Division entière

De plus, pour chacun de ces opérateurs, il existe un opérateur associé qui réalise successivement l'opération demandée puis l'affectation de la nouvelle valeur à la variable en question. Ainsi, l'instruction suivante :

```
x = x + 2
```

qui ajoute 2 à la valeur courante de `x` puis stocke le résultat du calcul dans `x` peut se réécrire :

```
x += 2
```

Ceci est purement un raccourci de notation, s'il ne vous semble pas évident à maîtriser au premier abord, vous pouvez vous en passer et toujours utiliser la notation `x = x + 2`.

Enfin, lorsque l'évaluation d'une expression implique plusieurs opérateurs, les règles de priorité sont les suivantes (de la priorité maximale à la priorité minimale) :

1. parenthèses ;
2. élévation à la puissance ;
3. multiplication / division ;
4. addition / soustraction ;
5. de gauche à droite.

Pour prendre un exemple concret, pour évaluer l'expression :

```
3.9 * x * (1 - x)
```

l'interpréteur Python commencera par évaluer le contenu de la parenthèse puis, les 2 opérations restantes étant toutes des multiplications, il les effectuera de gauche à droite.

De plus, lorsqu'une opération est effectuée entre deux variables de types différents, le type le plus générique est retenu. Par exemple, si l'on multiplie un entier par un nombre à virgule, le résultat sera de type `float`. De même, le résultat de l'addition entre un nombre complexe et un nombre à virgule est un complexe.

Attention. Comme indiqué en introduction, ce polycopié suppose que vous utilisez Python dans sa version 3. Il est à noter qu'il existe une différence importante entre Python 2 et Python 3 dans la façon d'effectuer des opérations mêlant nombres entiers et flottants. Par exemple, l'opération suivante :

```
x = 2 / 3
```

stockera, en Python 2, la valeur 0 (résultat de la division **entière** de 2 par 3) dans la variable `x` alors qu'en Python 3, la division **flottante** sera effectuée et ainsi `x` contiendra `0.666666...` En Python 3, si l'on souhaite effectuer une division entière, on pourra utiliser l'opérateur `//` :

```
print(2 // 3)
```

```
0
```

1.2 Structures de contrôle

Un programme est une séquence d'instructions dont l'ordre doit être respecté. Au-delà de cet aspect séquentiel, on peut souhaiter :

- n'effectuer certaines instructions que si une condition est vérifiée ;
- répéter certaines instructions ;
- factoriser une sous-séquence d'instructions au sein d'une fonction pour pouvoir y faire appel à plusieurs reprises dans le programme.

Les structures de contrôle associées à ces différents comportements sont décrits dans la suite de cette section.

1.2.1 Structures conditionnelles

On peut indiquer à un programme de n'exécuter une instruction (ou une séquence d'instructions) que si une certaine condition est remplie, à l'aide du mot-clé `if` :

```
x = 12
if x > 0:
    print("X est positif")
    print("X n'est pas négatif")
```

```
X est positif
X n'est pas négatif
```

On remarque ici que la condition est terminée par le symbole `:`, de plus, la séquence d'instructions à exécuter si la condition est remplie est **indentée**, cela signifie qu'elle est décalée d'un « cran » (généralement une tabulation ou 4 espaces) vers la droite. Cette indentation est une bonne pratique recommandée quel que soit le langage que vous utilisez, mais en Python, c'est même une obligation (sinon, l'interpréteur Python ne saura pas où commence et où se termine la séquence à exécuter sous condition).

Dans certains cas, on souhaite exécuter une série d'instructions si la condition est vérifiée et une autre série d'instructions si elle ne l'est pas. Pour cela, on utilise le mot-clé `else` comme suit :

```
x = -1
if x > 0:
    print("X est positif")
    print("X n'est pas négatif")
else:
    print("X est négatif")
```

```
X est négatif
```

Là encore, on remarque que l'indentation est de rigueur pour chacun des deux blocs d'instructions. On note également que le mot-clé `else` se trouve au même niveau que le `if` auquel il se réfère.

Enfin, de manière plus générale, il est possible de définir plusieurs comportements en fonction de plusieurs tests successifs, à l'aide du mot-clé `elif`. `elif` est une contraction de `else if`, qui signifie sinon si.

```
x = -1
if x > 0:
    print("X est positif")
    x = 4
elif x > -2:
    print("X est compris entre -2 et 0")
elif x > -4:
    print("X est compris entre -4 et -2")
else:
    print("X est inférieur à -4")
```

```
X est compris entre -2 et 0
```

Pour utiliser ces structures conditionnelles, il est important de maîtriser les différents opérateurs de comparaison à votre disposition en Python, dont voici une liste non exhaustive :

Opérateur	Comparaison effectuée	Exemple
`<`	Plus petit que	`x < 0`
`>`	Plus grand que	`x > 0`
`<=`	Plus petit ou égal à	`x <= 0`
`>=`	Plus grand ou égal à	`x >= 0`
`==`	Égal à	`x == 0`
`!=`	Différent de	`x != 0`
`is`	Test d'égalité pour le cas de la valeur `None`	`x is None`
`is not`	Test d'inégalité pour le cas de la valeur `None`	`x is not None`
`in`	Test de présence d'une valeur dans une liste	`x in [1, 5, 7]`

Il est notamment important de remarquer que, lorsque l'on souhaite tester l'égalité entre deux valeurs, l'opérateur à utiliser est == et non = (qui sert à affecter une valeur à une variable).

Exercice

Exercice 2.1 : Température de l'eau

Écrivez une expression conditionnelle, qui à partir d'une température d'eau stockée dans une variable `t` affiche dans le terminal si l'eau à cette température est à l'état liquide, solide ou gazeux.

Solution

```
t = -5
if t <= 0:
        print("l'eau est sous forme solide")
elif t < 100:
        print("l'eau est sous forme liquide")
else :
        print("l'eau est sous forme gazeuse")
```

1.2.2 Boucles

Il existe, en Python comme dans une grande majorité des langages de programmation, deux types de boucles :

- les boucles qui s'exécutent tant qu'une condition est vraie ;
- les boucles qui répètent la même série d'instructions pour différentes valeurs d'une variable (appelée **variable de boucle**).

Boucles `while`

Les premières ont une syntaxe très similaire à celle des structures conditionnelles simples :

```
x = 0
while x <= 10:
    print(x)
    x = 2 * x + 2
```

```
0
2
6
```

On voit bien ici, en analysant la sortie produite par ces quelques lignes, que le contenu de la boucle est répété plusieurs fois. En pratique, il est répété jusqu'à ce que la variable `x` prenne une valeur supérieure à 10 (14 dans notre cas). Il faut être très prudent avec ces boucles `while` car il est tout à fait possible de créer une boucle dont le programme ne sortira jamais, comme dans l'exemple suivant :

```
x = 2
y = 0
while x > 0:
    y = y - 1
    print(y)
print("Si on arrive ici, on a fini")
```

En effet, on a ici une boucle qui s'exécutera tant que `x` est positif, or la valeur de cette variable est initialisée à 2 et n'est pas modifiée au sein de la boucle, la condition sera donc toujours vérifiée et le programme ne sortira jamais de la boucle. Pour information, si vous vous retrouvez dans un tel cas, vous pourrez interrompre l'exécution du programme à l'aide de la combinaison de touches `Ctrl + C`.

Boucles `for`

Information

Si vous avez appris à programmer dans un autre langage que Python, il est possible que vous ayez été habitué(e) à ce que les boucles `for` soient utilisées pour itérer sur des entiers (par exemple les indices des éléments d'une liste). En Python, le principe de base est légèrement différent : par défaut, on itère sur les éléments d'une liste. Vous verrez dans le chapitre de ce polycopié dédié aux listes *comment faire lorsque l'on souhaite itérer sur les indices*.

Le second type de boucle repose en Python sur l'utilisation de listes (ou, plus généralement, d'itérables) dont nous reparlerons plus en détail dans la suite de cet ouvrage. Sachez pour le moment qu'une liste est un ensemble ordonné d'éléments. On peut alors exécuter une série d'instructions pour toutes les valeurs d'une liste :

```
for x in [1, 5, 7]:
    print(x)
print("Fin de la boucle")
```

```
1
5
7
Fin de la boucle
```

Cette syntaxe revient à définir une variable `x` qui prendra successivement pour valeur chacune des valeurs de la liste `[1, 5, 7]` dans l'ordre et à exécuter le code de la boucle (ici, un appel à la fonction `print`) pour cette valeur de la variable `x`.

Il est fréquent d'avoir à itérer sur les `n` premiers entiers. Cela peut se faire à l'aide de la fonction `range` (qui est abordée plus en détail dans *la section de ce cours sur les listes*) :

```
for i in range(1, 10):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
```

Quelle boucle choisir ?

Lorsque l'on débute la programmation en Python, il peut être difficile de choisir, pour un problème donné, entre les boucles `for` et `while` présentées ici. De manière générale, dès lors que l'on souhaite parcourir les éléments d'une liste ou d'un dictionnaire, on utilisera la boucle `for`. De même, si l'on connait à l'avance le nombre d'itérations que l'on souhaite effectuer, on utilisera une boucle `for` (couplée avec un appel à la fonction `range`). Comme son nom l'indique, la boucle `while` sera utilisée dès lors que l'on souhaite répéter une action **tant qu'** une condition est vérifiée.

Exercice

Exercice 2.2 : Nombres impairs

Écrivez une boucle permettant d'afficher tous les nombres impairs inférieurs à une valeur `n` initialement fixée.

Solution

- Version avec utilisation du modulo

```
n = 8
i = 0
```

(suite sur la page suivante)

(suite de la page précédente)

```
while(i < n):
    if i % 2 != 0:
        print(i)
    i+=1
```

- Version sans utilisation du modulo

```
n = 8
i = 1
while(i < n):
    print(i)
    i += 2
```

1.2.3 Fonctions

Nous avons déjà vu dans ce qui précède, sans le dire, des fonctions. Par exemple, lorsque l'on écrit :

```
print(x)
```

```
7
```

on demande l'appel à une fonction, nommée `print` et prenant un **argument** (ici, la variable `x`). La fonction `print` ne retourne pas de valeur, elle ne fait qu'afficher la valeur contenue dans `x` sur le terminal. D'autres fonctions, comme `type` dont nous avons parlé plus haut, **retournent une valeur** et cette valeur peut être utilisée dans la suite du programme, comme dans l'exemple suivant :

```
x = type(1)  # On stocke dans x la valeur retournée par type
y = type(2.)
if x == y:
    print("types identiques")
else:
    print("types différents")
```

```
types différents
```

En pratique, dans un programme, on aura recours à la définition de fonctions pour décomposer un problème global en sous-problèmes, chaque sous-problème étant géré par une fonction qui pourra elle-même, au besoin, faire appel à d'autres fonctions.

Définition d'une fonction

Lorsqu'un ensemble d'instructions est susceptible d'être utilisé à plusieurs occasions dans un ou plusieurs programmes, il est recommandé de l'isoler au sein d'une fonction. Cela présentera les avantages suivants :

- en donnant un nom à la fonction et en listant la liste de ses arguments, on explicite la sémantique de l'ensemble d'instructions en question, ses entrées et sorties éventuelles, ce qui rend le code beaucoup plus lisible ;
- s'il est nécessaire d'adapter à l'avenir le code pour résoudre un *bug* ou le rendre plus générique, vous n'aurez à modifier le code qu'à un endroit (dans le corps de la fonction) et non pas à chaque fois que le code est répété.

Pour définir une fonction en Python, on utilise le mot-clé `def` :

```
def f(x):
    y = 5 * x + 2
    z = x + y
    return z // 2
```

On a ici défini une fonction

- dont le nom est `f` ;
- qui prend un seul argument, noté `x` ;
- qui retourne une valeur, comme indiqué dans la ligne débutant par le mot-clé `return`.

Il est possible, en Python, d'écrire des fonctions retournant plusieurs valeurs. Pour ce faire, ces valeurs seront séparées par des virgules dans l'instruction `return` :

```
def f(x):
    y = 5 * x + 2
    z = x + y
    return z // 2, y
```

Enfin, en l'absence d'instruction `return`, une fonction retournera la valeur `None`.

Il est également possible d'utiliser le nom des arguments de la fonction lors de l'appel, pour ne pas risquer de se tromper dans l'ordre des arguments. Par exemple, si l'on a la fonction suivante :

```
def affiche_infos_personne(poids, taille):
    print("Poids: ", poids)
    print("Taille: ", taille)
```

Les trois appels suivants sont équivalents :

```
affiche_infos_personne(80, 180)
```

```
Poids:  80
Taille:  180
```

```
affiche_infos_personne(taille=180, poids=80)
```

```
Poids:  80
Taille:  180
```

```
affiche_infos_personne(poids=80, taille=180)
```

```
Poids:  80
Taille:  180
```

Notons qu'il est alors possible d'interchanger l'ordre des arguments lors de l'appel d'une fonction si on précise leur nom. Évidemment, pour que cela soit vraiment utile, il est hautement recommandé d'utiliser des **noms d'arguments explicites** lors de la définition de vos fonctions.

Argument(s) optionnel(s) d'une fonction

Certains arguments d'une fonction peuvent avoir une valeur par défaut, décidée par la personne qui a écrit la fonction. Dans ce cas, si l'utilisateur ne spécifie pas explicitement de valeur pour ces arguments lors de l'appel à la fonction, c'est la valeur par défaut qui sera utilisée dans la fonction, dans le cas contraire, la valeur spécifiée sera utilisée.

Par exemple, la fonction `print` dispose de plusieurs arguments facultatifs, comme le caractère par lequel terminer l'affichage (par défaut, un retour à la ligne, `"\n"`) :

```
print("La vie est belle")
```

```
La vie est belle
```

```
print("Life is beautiful")
```

```
Life is beautiful
```

```
print("La vie est belle", end="--")
```

```
La vie est belle--
```

```
print("Life is beautiful", end="*-*")
```

```
Life is beautiful*-*
```

Lorsque vous définissez une fonction, la syntaxe à utiliser pour donner une valeur par défaut à un argument est la suivante :

```
def f(x, y=0):  # La valeur par défaut pour y est 0
    return x + 5 * y
```

Attention toutefois, les arguments facultatifs (*ie.* qui disposent d'une valeur par défaut) doivent impérativement se trouver, dans la liste des arguments, après le dernier argument obligatoire. Ainsi, la définition de fonction suivante **n'est pas correcte** :

```
def f(x, y=0, z):
    return x - 2 * y + z
```

1.2.4 Exercices

Exercice 2.3 : Triangle équilatéral

Écrivez une fonction en Python qui prenne en argument une longueur `long` et retourne l'aire du triangle équilatéral de côté `long`.

Solution

```python
import math

def aire_equi(long):
    base = long
    hauteur = long * math.sin(math.pi / 3)
    return base * hauteur / 2

print(aire_equi(1.))
```

Exercice 2.4 : Suite récurrente

Écrivez une fonction en Python qui affiche tous les termes plus petits que 1000 de la suite (u_n) définie comme :

$$\begin{aligned} u_0 &= 2 \\ \forall n \geq 1,\ u_n &= u_{n-1}^2 \end{aligned}$$

Solution

- Version itérative (avec une boucle)

```python
def affiche_u_n():
    u = 2
    while u < 1000:
        print(u)
        u = u ** 2

affiche_u_n()
```

- Version récursive (avec des appels de fonction)

```python
def affiche_u_n(u=2):
    if u < 1000:
        print(u)
        affiche_u_n(u ** 2)

affiche_u_n()
```

Ici, on a fixé une valeur par défaut à l'argument `u` correspondant à l'initialisation de la suite, pour que l'appel initial se fasse comme pour la version itérative de la fonction (`affiche_u_n()`).

1.3 Les modules en Python

Jusqu'à présent, nous avons utilisé des fonctions (comme `print`) issues de la librairie standard de Python. Celles-ci sont donc chargées par défaut lorsque l'on exécute un script Python. Toutefois, il peut être nécessaire d'avoir accès à d'autres fonctions et/ou variables, définies dans d'autres librairies. Pour cela, il sera utile de charger le **module** correspondant.

Prenons l'exemple du module `math` qui propose un certain nombre de fonctions mathématiques usuelles (`sin` pour le calcul du sinus d'un angle, `sqrt` pour la racine carrée d'un nombre, *etc.*) ainsi que des constantes mathématiques très utiles comme `pi`. Le code suivant charge le module en mémoire puis fait appel à certaines de ses fonctions et/ou variables :

```
import math

print(math.sin(0))
```

```
0.0
```

```
print(math.pi)
```

```
3.141592653589793
```

```
print(math.cos(2 * math.pi))
```

```
1.0
```

```
print(math.sqrt(2))
```

```
1.4142135623730951
```

Vous remarquerez ici que l'instruction d'import du module se trouve nécessairement avant les instructions faisant référence aux fonctions et variables de ce module, faute de quoi ces dernières ne seraient pas définies. De manière générale, vous prendrez la bonne habitude d'écrire les instructions d'import en tout début de vos fichiers Python, pour éviter tout souci.

Enfin, il est possible de renommer le module au moment où vous l'importez, ce qui peut être pratique pour les modules dont le nom est long ou ceux que vous allez utiliser de manière particulièrement fréquente dans votre code. Typiquement, il est de coutume de renommer le module `numpy` en `np` (*cf.* *ce chapitre* pour plus de détails sur ce module) lorsqu'on l'importe :

```
import numpy as np

print(np.zeros(5))  # Ici on utilise alors la forme abrégée
```

```
[0. 0. 0. 0. 0.]
```

1.4 Liste des exercices de ce chapitre

1. *Température de l'eau*
2. *Nombres impairs*
3. *Triangle équilatéral*
4. *Suite récurrente*

CHAPTER 2

LES DATES

Dans la partie précédente, nous avons présenté les types de base du langage Python et les opérateurs associés aux types numériques. Lorsque l'on souhaite manipuler des dates et des heures, on devra avoir recours à un autre type de données, défini dans le module `datetime`. Pour cela, il faudra commencer par charger ce module en ajoutant l'instruction :

```
import datetime
```

en en-tête de votre script Python.

Pour créer une nouvelle variable de ce type, on utilisera la syntaxe :

```
d = datetime.datetime(annee, mois, jour, heure, minutes)
```

La syntaxe `datetime.datetime`, qui peut vous sembler bizarre au premier coup d'oeil signifie à l'interpréteur Python qu'il doit chercher dans le module `datetime` une fonction dont le nom est `datetime` et l'appeler.

En fait, on pourrait rajouter lors de l'appel de `datetime.datetime` un argument pour spécifier les secondes, puis éventuellement un autre pour les microsecondes, si l'on avait besoin d'une heure plus précise. Si, au contraire, on ne spécifie pas l'heure lors de l'appel de la fonction, l'heure `00h00` sera choisie par défaut.

Par exemple :

```
import datetime  # Cette commande doit se trouver en début de fichier

# [...] (ici, du code concernant autre chose si besoin)

d = datetime.datetime(2020, 8, 27, 17, 23)
print(d)
```

```
2020-08-27 17:23:00
```

```
d = datetime.datetime(2020, 8, 27, 17, 23, 32)
print(d)
```

```
2020-08-27 17:23:32
```

```
d = datetime.datetime(2020, 8, 27)
print(d)
```

```
2020-08-27 00:00:00
```

Les opérateurs de comparaison vus au chapitre précédent (<, >, <=, >=, ==, !=) fonctionnent de manière naturelle avec ce type de données :

```
d1 = datetime.datetime(2020, 8, 27, 17, 23)
d2 = datetime.datetime(2020, 8, 27, 17, 28)
d3 = datetime.datetime(2020, 8, 27, 17, 23)
print(d1 < d2)
```

```
True
```

```
print(d1 == d3)
```

```
True
```

```
print(d1 > d3)
```

```
False
```

Il existe d'autres moyens de construire des variables de type date. On peut générer une date correspondant à l'heure actuelle avec la fonction `datetime.now` du module `datetime` :

```
date_actuelle = datetime.datetime.now()
```

2.1 Transformation d'une date en chaîne de caractères

Si l'on souhaite transformer une date en chaîne de caractères (par exemple pour l'afficher), on peut lui appliquer la fonction `str` :

```
print(str(datetime.datetime(2020, 8, 27)))
```

```
2020-08-27 00:00:00
```

Dans ce cas, on ne peut pas gérer la façon dont se fait cette transformation. Pour contourner cette limitation, il convient alors d'utiliser `strftime` :

```
d1 = datetime.datetime(...)
s = d1.strftime(format)
```

L'attribut `format` que l'on passe à cette fonction va servir à définir comment on souhaite représenter la date en question. Il s'agit d'une chaîne de caractères qui pourra contenir les éléments suivants :

Code	Signification
`%Y`	Année
`%m`	Mois
`%d`	Jour
`%H`	Heure
`%M`	Minutes

Remarquez que la casse n'est pas neutre pour les codes à utiliser : `%M` et `%m` ont des significations tout à fait différentes. Notez également qu'il existe d'autres codes permettant de générer des chaînes de caractères plus variées encore. Une liste de ces codes est disponible sur la page d'aide du module `datetime`.

Vous pouvez vous référer aux exemples ci-dessous pour mieux comprendre le fonctionnement de la fonction `strftime` :

```
d = datetime.datetime(2020, 8, 27, 17, 23)

print(d.strftime("%d-%m-%Y, %H:%M"))
```

```
27-08-2020, 17:23
```

```
print(d.strftime("%d-%m-%Y"))
```

```
27-08-2020
```

```
print(d.strftime("%H:%M"))
```

```
17:23
```

```
print(d.strftime("%d/%m/%Y %Hh%M"))
```

```
27/08/2020 17h23
```

Attention !

Attention aux confusions possibles entre `datetime.strftime` et `datetime.strptime` :

- `datetime.strftime` : le `f` signifie *format*, il s'agit donc de mettre en forme une date, selon un format donné, dans une chaîne de caractères
- `datetime.strptime` : le `p` signifie *parse* (en anglais), il s'agit donc de reconnaître une date dans une chaîne de caractères et de retourner la date en question

Il est également possible d'effectuer l'opération inverse (lire une date contenue dans une chaîne de caractères, étant donné un format connu). Cela se fait avec la fonction `datetime.strptime` :

```
d1 = datetime.datetime.strptime(chaine_a_lire, format)
```

Voici deux exemples d'utilisation de cette fonction :

```
d1 = datetime.datetime.strptime("2020/8/27, 17:23", "%Y/%m/%d, %H:%M")
print(d1)
```

```
2020-08-27 17:23:00
```

```
d2 = datetime.datetime.strptime("27-08-2020", "%d-%m-%Y")
print(d2)
```

```
2020-08-27 00:00:00
```

2.2 Attributs des objets `datetime`

Lorsque l'on définit une date de type `datetime.datetime`, on peut accéder à certains de ses attributs directement :

```
d1 = datetime.datetime.strptime("2020/8/27, 17:23", "%Y/%m/%d, %H:%M")
print(d1)
```

```
2020-08-27 17:23:00
```

```
print(d1.year)
```

```
2020
```

```
print(d1.month)
```

```
8
```

```
print(d1.day)
```

```
27
```

```
print(d1.hour)
```

```
17
```

```
print(d1.minute)
```

```
23
```

```
print(d1.second)
```

```
0
```

2.3 Calcul de temps écoulé

On peut ensuite souhaiter calculer la différence entre deux dates. Le résultat de cette opération est une **durée**, représentée en Python par le type `timedelta` (lui aussi défini dans le module `datetime`).

```
d1 = datetime.datetime(2020, 8, 27, 17, 23)
d2 = datetime.datetime(2020, 8, 27, 17, 28)
intervalle_de_temps = d1 - d2
print(type(intervalle_de_temps))
```

```
<class 'datetime.timedelta'>
```

Une autre façon de créer une durée au format `timedelta` est d'utiliser la fonction du même nom :

```
duree = datetime.timedelta(weeks=0, days=10, hours=3, minutes=10, seconds=23)
print(duree)
```

```
10 days, 3:10:23
```

Très souvent, il est utile pour manipuler une durée de la convertir en un nombre de secondes et de manipuler ce nombre ensuite. Cela se fait à l'aide de la commande :

```
d1 = datetime.datetime(2020, 8, 27, 17, 23)
d2 = datetime.datetime(2020, 8, 27, 17, 28)
intervalle_de_temps = d1 - d2
print(intervalle_de_temps.total_seconds())
```

```
-300.0
```

On remarque ici que l'intervalle obtenu est négatif, ce qui était prévisible car il s'agit de l'intervalle `d1 - d2` et on a `d1 < d2`.

Notez enfin que l'on peut tout à fait ajouter une durée à une date :

```
d1 = datetime.datetime(2020, 8, 27, 17, 23)
d2 = datetime.datetime(2020, 8, 27, 17, 28)
d3 = datetime.datetime(2020, 8, 27, 18, 00)
intervalle_de_temps = d2 - d1
print(d3 + intervalle_de_temps)
```

```
2020-08-27 18:05:00
```

2.4 Exercices

Exercice 3.1

S'est-il écoulé plus de temps (i) entre le 2 Janvier 1920 à 7h32 et le 4 Mars 1920 à 5h53 ou bien (ii) entre le 30 Décembre 1999 à 17h12 et le 1er Mars 2000 à 15h53 ?

Solution

```
import datetime

interv1_date1 = datetime.datetime(1920, 1, 2, 7, 32)
interv1_date2 = datetime.datetime(1920, 3, 4, 5, 53)
duree1 = interv1_date2 - interv1_date1

interv2_date1 = datetime.datetime(1999, 12, 30, 17, 12)
interv2_date2 = datetime.datetime(2000, 3, 1, 15, 53)
duree2 = interv2_date2 - interv2_date1

if duree1 > duree2:
    print("Le premier intervalle est le plus grand.")
else:
    print("Le second intervalle est le plus grand.")
```

Exercice 3.2

À l'aide des fonctions du module `datetime` vues plus haut, affichez, pour chaque année civile comprise entre 2010 et 2030, si elle est bissextile ou non.

Solution

```
import datetime

duree_annee_normale = 365 * 24 * 60 * 60

for annee in range(2010, 2031):
    date_debut = datetime.datetime(annee, 1, 1)
    date_fin = datetime.datetime(annee + 1, 1, 1)
    duree_anne_courante = date_fin - date_debut
    if duree_anne_courante.total_seconds() > duree_annee_normale:
        print(annee, "est bissextile")
    else:
        print(annee, "n'est pas bissextile")
```

CHAPTER 3

LES LISTES

Définition

Une liste est une collection ordonnée de valeurs. Dans une liste, chaque valeur occupe une position bien définie que l'on repère par un entier appelé **indice**. La première valeur est associée à l'indice 0, la seconde à l'indice 1, *etc.* Une liste a une longueur (*i.e.* un nombre d'éléments) finie, ainsi la liste vide a pour longueur 0.

En Python, il n'est pas nécessaire que tous les éléments d'une liste soient du même type, même si dans les exemples que nous considérerons, ce sera souvent le cas.

On peut trouver des informations précieuses sur le sujet des listes dans l'aide en ligne de Python disponible à l'adresse : https://docs.python.org/3/tutorial/datastructures.html.

3.1 Avant-propos : listes et itérables

Dans la suite, nous parlerons de listes, qui est un type de données bien spécifique en Python. Toutefois, une grande partie de notre propos pourra se transposer à l'ensemble des itérables en Python (c'est-à-dire l'ensemble des objets Python dont on peut parcourir les éléments un à un).

Il existe toutefois une différence majeure entre listes et itérables : nous verrons dans la suite de ce chapitre que l'on peut accéder au i-ème élément d'une liste simplement, alors que ce n'est généralement pas possible pour un itérable (pour ce dernier, il faudra parcourir l'ensemble de ses éléments et s'arrêter lorsque l'on est effectivement rendu au i-ème).

Toutefois, si l'on a un itérable `iterable`, il est possible de le transformer en liste simplement à l'aide de la fonction `list` :

```
liste = list(iterable)
```

3.2 Création de liste

Pour créer une liste contenant des éléments définis (par exemple la liste contenant les entiers 1, 5 et 7), il est possible d'utiliser la syntaxe suivante :

```
liste = [1, 5, 7]
```

De la même façon, on peut créer une liste vide (ne contenant aucun élément) :

```
liste = []
print(len(liste))
```

```
0
```

On voit ici la fonction `len` qui retourne la taille d'une liste passée en argument (ici 0 puisque la liste est vide).

Toutefois, lorsque l'on souhaite créer des listes longues (par exemple la liste des 1000 premiers entiers), cette méthode est peu pratique. Heureusement, il existe des fonctions qui permettent de créer de telles listes. Par exemple, la fonction `range(a, b)` retourne un itérable contenant les entiers de `a` (inclus) à `b` (exclu) :

```
it = range(1, 10)      # it = [1, 2, 3, ..., 9]
it = range(10)         # it = [0, 1, 2, ..., 9]
it = range(0, 10, 2)   # it = [0, 2, 4, ..., 8]
```

On remarque que, si l'on ne donne qu'un argument à la fonction `range`, l'itérable retourné débute à l'entier 0. Si, au contraire, on passe un troisième argument à la fonction `range`, cet argument correspond au pas utilisé entre deux éléments successifs.

3.3 Accès aux éléments d'une liste

Pour accéder au i-ème élément d'une liste, on utilise la syntaxe :

```
liste[i]
```

Attention, toutefois, le premier indice d'une liste est 0, on a donc :

```
liste = [1, 5, 7]
print(liste[1])
```

```
5
```

```
print(liste[0])
```

```
1
```

On peut également accéder au dernier élément d'une liste en demandant l'élément d'indice `-1` :

```
liste = [1, 5, 7]
print(liste[-1])
```

```
7
```

```
print(liste[-2])
```

```
5
```

```
print(liste[-3])
```

```
1
```

De la même façon, on peut accéder au deuxième élément en partant de la fin *via* l'indice `-2`, *etc.*

Ainsi, pour une liste de taille n, les valeurs d'indice valides sont les entiers compris entre $-n$ et $n-1$ (inclus).

Il est également à noter que l'accès aux éléments d'une liste peut se faire en lecture (lire l'élément stocké à l'indice `i`) comme en écriture (modifier l'élément stocké à l'indice `i`) :

```
liste = [1, 5, 7]
print(liste[1])
```

```
5
```

```
liste[1] = 2
print(liste)
```

```
[1, 2, 7]
```

Enfin, on peut accéder à une sous-partie d'une liste à l'aide de la syntaxe `liste[d:f]` où `d` est l'indice de début et `f` est l'indice de fin (exclu). Ainsi, on a :

```
liste = [1, 5, 7, 8, 0, 9, 8]
print(liste[2:4])
```

```
[7, 8]
```

Lorsque l'on utilise cette syntaxe, si l'on omet l'indice de début, la sélection commence au début de la liste et si l'on omet l'indice de fin, elle s'étend jusqu'à la fin de la liste :

```
liste = [1, 5, 7, 8, 0, 9, 8]
print(liste[:3])
```

```
[1, 5, 7]
```

```
print(liste[5:])
```

```
[9, 8]
```

3.4 Parcours d'une liste

Lorsque l'on parcourt une liste, on peut vouloir accéder :

- aux éléments stockés dans la liste uniquement ;
- aux indices de la liste uniquement (même si c'est rare) ;
- aux indices de la listes et aux éléments associés.

Ces trois cas de figure impliquent trois parcours de liste différents, décrits dans ce qui suit.

Attention. Quel que soit le parcours de liste utilisé, il est fortement déconseillé de supprimer ou d'insérer des éléments dans une liste pendant le parcours de celle-ci.

3.4.1 Parcours des éléments

Pour parcourir les éléments d'une liste, on utilise une boucle `for` :

```
liste = [1, 5, 7]
for elem in liste:
    print(elem)
```

```
1
5
7
```

Dans cet exemple, la variable `elem` va prendre successivement pour valeur chacun des éléments de la liste.

3.4.2 Parcours par indices

Pour avoir accès aux indices (positifs) de la liste, on devra utiliser un subterfuge. On sait que les indices d'une liste sont les entiers compris entre 0 (inclus) et la taille de la liste (exclu). On va donc utiliser la fonction `range` pour cela :

```
liste = [1, 5, 7]
n = len(liste)  # n = 3 ici
for i in range(n):
    print(i, liste[i])
```

```
0 1
1 5
2 7
```

3.4.3 Parcours par éléments et indices

Dans certains cas, enfin, on a besoin de manipuler simultanément les indices d'une liste et les éléments associés. Cela se fait à l'aide de la fonction `enumerate` :

```
liste = [1, 5, 7]
for i, elem in enumerate(liste):
    print(i, elem)
```

```
0 1
1 5
2 7
```

On a donc ici une boucle `for` pour laquelle, à chaque itération, on met à jour les variables `i` (qui contient l'indice courant) et `elem` (qui contient l'élément se trouvant à l'indice `i` dans la liste `liste`).

Pour tous ces parcours de listes, il est conseillé d'utiliser des noms de variables pertinents, afin de limiter les confusions dans la nature des éléments manipulés. Par exemple, on pourra utiliser `i` ou `j` pour noter des indices, mais on préfèrera `elem` ou `val` pour désigner les éléments de la liste.

3.4.4 Exercice

Exercice 4.1 : Argmax

Écrivez une fonction en Python qui permette de calculer l'argmax d'une liste, c'est-à-dire l'indice auquel est stockée la valeur maximale de la liste. Si cette valeur maximale est présente plusieurs fois dans la liste, on retournera l'indice de sa première occurrence.

Solution

```
def argmax(liste):
    i_max = None
    # On initialise elem_max à une valeur
    # qui n'est clairement pas le max
    if len(liste) > 0:
        elem_max = liste[0] - 1
    for i, elem in enumerate(liste):
        if elem > elem_max:
            i_max = i
            elem_max = elem
    return i_max

print(argmax([1, 6, 2, 4]))
```

3.5 Manipulations de listes

Nous présentons dans ce qui suit les opérations élémentaires de manipulation de listes.

3.5.1 Insertion d'élément

Pour insérer un nouvel élément dans une liste, on peut :

- rajouter un élément à la fin de la liste à l'aide de la méthode `append` ;
- insérer un élément à l'indice `i` de la liste à l'aide de la méthode `insert`.

Comme vous pouvez le remarquer, il est ici question de méthodes et non plus de fonctions. Pour l'instant, sachez que les méthodes sont des fonctions spécifiques à certains objets, comme les listes par exemple. L'appel de ces méthodes est un peu particulier, comme vous pouvez le remarquer dans ce qui suit :

```
liste = [1, 5, 7]
liste.append(2)
print(liste)
```

```
[1, 5, 7, 2]
```

```
liste.insert(2, 0)  # insère la valeur 0 à l'indice 2
print(liste)
```

```
[1, 5, 0, 7, 2]
```

3.5.2 Suppression d'élément

Si l'on souhaite, maintenant, supprimer un élément dans une liste, deux cas de figures peuvent se présenter. On peut souhaiter :

- supprimer l'élément situé à l'indice `i` dans la liste, à l'aide de l'instruction `del` ;
- supprimer la première occurrence d'une valeur donnée dans la liste à l'aide de la méthode `remove`.

```
liste = [1, 5, 7]
del liste[1]  # l'élément d'indice 1 est le deuxième élément de la liste !
print(liste)
```

```
[1, 7]
```

```
liste = [7, 1, 5, 1]
liste.remove(1) # supprime la première occurrence de 1 dans la liste
print(liste)
```

```
[7, 5, 1]
```

3.5.3 Recherche d'élément

Pour trouver l'indice de la première occurrence d'une valeur dans une liste, on utilisera la méthode `index` :

```
liste = [1, 5, 7]
print(liste.index(7))
```

```
2
```

Si l'on ne cherche pas à connaître la position d'une valeur dans une liste mais simplement à savoir si une valeur est présente dans la liste, on peut utiliser le mot-clé `in` :

```
liste = [1, 5, 7]
if 5 in liste:
    print("5 est dans liste")
```

```
5 est dans liste
```

3.5.4 Création de listes composites

On peut également concaténer deux listes (c'est-à-dire mettre bout à bout leur contenu) à l'aide de l'opérateur + :

```
liste1 = [1, 5, 7]
liste2 = [3, 4]
liste = liste1 + liste2
print(liste)
```

```
[1, 5, 7, 3, 4]
```

Dans le même esprit, l'opérateur `*` peut aussi être utilisé pour des listes :

```
liste1 = [1, 5]
liste2 = 3 * liste1
print(liste2)
```

```
[1, 5, 1, 5, 1, 5]
```

Bien entendu, vu le sens de cet opérateur, on ne peut multiplier une liste que par un entier.

3.5.5 Tri de liste

Enfin, on peut trier les éléments contenus dans une liste à l'aide de la fonction `sorted` :

```
liste = [4, 5, 2]
liste2 = sorted(liste)
print(liste2)
```

```
[2, 4, 5]
```

Il est à noter que l'on peut trier une liste dès lors que celle-ci contient des éléments du même type (ou de types assimilables, par exemple des valeurs numériques, entières ou flottantes) à partir du moment où une relation d'ordre est définie sur ce type. On peut donc par exemple trier des listes de chaînes de caractères :

```
liste = ["a", "zzz", "c"]
print(sorted(liste))
```

```
['a', 'c', 'zzz']
```

Sachant que les émoticones sont des caractères comme les autres, on peut ainsi (enfin) obtenir une réponse à un problème vieux comme le monde :

```
liste = ["□", "□"]
print(sorted(liste))
```

```
['□', '□']
```

3.5.6 Exercices

Exercice 4.2 : Intersection de listes

Écrivez une fonction qui prenne deux listes en entrée et retourne l'intersection des deux listes (c'est-à-dire une liste contenant tous les éléments présents dans les deux listes).

Solution

```
def intersection(liste1, liste2):
    liste_intersection = []
    for elem in liste1:
        if elem in liste2 and not elem in liste_intersection:
            liste_intersection.append(elem)
    return liste_intersection

print(intersection([1, 6, 2, 4], [2, 7, 6]))
```

Exercice 4.3 : Union de listes

Écrivez une fonction qui prenne deux listes en entrée et retourne l'union des deux listes (c'est-à-dire une liste contenant tous les éléments présents dans au moins une des deux listes) sans doublon.

Solution

```
def union_sans_doublon(liste1, liste2):
    liste_union = []
    for elem in liste1 + liste2:
        if elem not in liste_union:
```

(suite sur la page suivante)

(suite de la page précédente)

```
            liste_union.append(elem)
    return liste_union

print(union_sans_doublon([1, 6, 2, 4], [2, 7, 6, 2]))
```

3.6 Copie de liste

Pour la plupart des variables, en Python, la copie ne pose pas de problème :

```
a = 12
b = a
a = 5
print(a, b)
```

```
5 12
```

Cela ne se passe pas de la même façon pour les listes. En effet, si `liste` est une liste, lorsque l'on écrit :

```
liste2 = liste
```

on ne recopie pas le contenu de `liste` dans `liste2`, mais on crée une variable `liste2` qui va « pointer » vers la même position dans la mémoire de votre ordinateur que `liste`. La différence peut sembler mince, mais cela signifie que si l'on modifie `liste` même après l'instruction `liste2 = liste`, la modification sera répercutée sur `liste2` :

```
liste = [1, 5, 7]
liste2 = liste
liste[1] = 2
print(liste, liste2)
```

```
[1, 2, 7] [1, 2, 7]
```

Lorsque l'on souhaite éviter ce comportement, il faut effectuer une copie explicite de liste, à l'aide par exemple de la fonction `list` :

```
liste = [1, 5, 7]
liste2 = list(liste)
liste[1] = 2
print(liste, liste2)
```

```
[1, 2, 7] [1, 5, 7]
```

3.7 Bonus : listes en compréhension

Il est possible de créer des listes en filtrant et/ou modifiant certains éléments d'autres listes ou itérables. Supposons par exemple que l'on souhaite créer la liste des carrés des 10 premiers entiers naturels. Le code qui suit présente deux façons équivalentes de créer une telle liste :

```
# Façon "classique"
liste = []
for i in range(10):
    liste.append(i ** 2)
print(liste)
```

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

```
# En utilisant les listes en compréhension
liste = [i ** 2 for i in range(10)]
print(liste)
```

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

On remarque que la syntaxe de liste en compréhension est plus compacte. On peut également appliquer un filtre sur les éléments de la liste de départ (ici `range(10)`) à considérer à l'aide du mot-clé `if` :

```
liste = [i ** 2 for i in range(10) if i % 2 == 0]
print(liste)
```

```
[0, 4, 16, 36, 64]
```

Ici, on n'a considéré que les entiers pairs.

3.8 Super Bonus : le parcours simultané de plusieurs listes

Il peut arriver que l'on ait à parcourir simultanément plusieurs listes. Par exemple, supposons que l'on ait une liste stockant les prénoms d'enfants d'une classe et une autre liste contenant leurs noms. Si l'on veut afficher les noms et prénoms de ces enfants, on peut effectuer un parcours par indice :

```
prenoms = ["Jeanne", "Anne", "Camille"]
noms = ["Papin", "Bakayoko", "Drogba"]

for i in range(len(prenoms)):
    print(prenoms[i], noms[i])
```

```
Jeanne Papin
Anne Bakayoko
Camille Drogba
```

On peut aussi se dire que, en toute logique, on n'a pas réellement besoin ici d'accéder aux indices des éléments et que l'on voudrait juste effectuer un parcours simultané des deux listes. C'est ce que permet la fonction `zip()` :

```
prenoms = ["Jeanne", "Anne", "Camille"]
noms = ["Papin", "Bakayoko", "Drogba"]

for p, n in zip(prenoms, noms):
    print(p, n)
```

```
Jeanne Papin
Anne Bakayoko
Camille Drogba
```

On peut même parcourir plus de deux listes simultanément :

```
prenoms = ["Jeanne", "Anne", "Camille"]
noms = ["Papin", "Bakayoko", "Drogba"]
ages = [12, 11, 12]

for p, n, a in zip(prenoms, noms, ages):
    print(p, n, a)
```

```
Jeanne Papin 12
Anne Bakayoko 11
Camille Drogba 12
```

Bien entendu, pour pouvoir utiliser `zip()`, il faut que les listes soient de même taille.

3.9 Liste des exercices de ce chapitre

1. *Argmax*
2. *Intersection de listes*
3. *Union de listes*

CHAPTER 4

LES CHAÎNES DE CARACTÈRES

Nous nous intéressons maintenant à un autre type de données particulier du langage Python : les chaînes de caractères (type `str`). Pour créer une chaîne de caractères, il suffit d'utiliser des guillemets, simples ou doubles (les deux sont équivalents) :

```
s1 = "abc"
s2 = 'bde'
```

Comme pour les listes (et peut-être même plus encore), il est fortement conseillé de se reporter à l'aide en ligne dédiée lorsque vous avez des doutes sur la manipulation de chaînes de caractères : https://docs.python.org/3/library/stdtypes.html#string-methods

4.1 Conversion d'une chaîne en nombre

Si une chaîne de caractères représente une valeur numérique (comme la chaîne `"10.2"` par exemple), on peut la transformer en un entier ou un nombre à virgule, afin de l'utiliser ensuite pour des opérations arithmétiques. On utilise pour cela les fonctions de conversion, respectivement `int` et `float`.

```
s = '10.2'
f = float(s)
print(f)
```

```
10.2
```

```
print(f == s)
```

```
False
```

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

```
12.2
```

```
s = '10'
i = int(s)
print(i)
```

```
10
```

```
print(i == s)
```

```
False
```

```
print(i - 1)
```

```
9
```

4.2 Analogie avec les listes

Les chaînes de caractères se manipulent en partie comme des listes. On peut ainsi obtenir la taille d'une chaîne de caractères à l'aide de la fonction `len`, ou accéder à la i-ème lettre d'une chaîne de caractères avec la notation `s[i]`. Comme pour les listes, il est possible d'indicer une chaîne de caractères en partant de la fin, en utilisant des indices négatifs :

```
s = "abcdef"
print(len(s))
```

```
6
```

```
print(s[0])
```

```
a
```

```
print(s[-1])
```

```
f
```

De même, on peut sélectionner des sous-parties de chaînes de caractères à partir des indices de début et de fin de la sélection. Comme pour les listes, l'indice de fin correspond au premier élément exclu de la sélection :

```
s = "abcdef"
print(s[2:4])
```

```
cd
```

Comme pour les listes, on peut concaténer deux chaînes de caractères à l'aide de l'opérateur + ou répéter une chaîne de caractères avec l'opérateur * :

```
s = "ab" + ('cde' * 3)
print(s)
```

```
abcdecdecde
```

On peut également tester la présence d'une sous-chaîne de caractères dans une chaîne avec le mot-clé `in` :

```
s = "abcde"
print("a" in s)
```

```
True
```

```
print("bcd" in s)
```

```
True
```

```
print("bCd" in s)
```

```
False
```

Attention. Toutefois, l'analogie entre listes et chaînes de caractères est loin d'être parfaite. Par exemple, on peut accéder au i-ème élément d'une chaîne de caractères en lecture, mais pas en écriture. Si `s` est une chaîne de caractères, on ne peut pas exécuter `s[2] = "c"` par exemple.

4.3 Principales méthodes de la classe `str`

La liste de méthodes de la classe `str` qui suit n'est pas exhaustive, il est conseillé de consulter l'aide en ligne de Python pour plus d'informations.

- `ch.count(sub)` : Retourne le nombre d'occurrences de `sub` dans `ch`
- `ch.endswith(suffix)` : Retourne `True` si `ch` se termine par `suffix`
- `ch.startswith(prefix)` : Retourne `True` si `ch` commence par `prefix`
- `ch.find(sub)` : Retourne l'indice du début de la première occurrence de `sub` dans `ch`
- `ch.rfind(sub)` : Retourne l'indice du début de la dernière occurrence de `sub` dans `ch`
- `ch.islower()` : Retourne `True` si `ch` est constituée uniquement de caractères minuscules
- `ch.isupper()` : Retourne `True` si `ch` est constituée uniquement de caractères majuscules
- `ch.isnumeric()` : Retourne `True` si `ch` est constituée uniquement de chiffres
- `ch.lower()` : Retourne la version minuscule de `ch`
- `ch.upper()` : Retourne la version majuscule de `ch`
- `ch.replace(old, new)` : Retourne une copie de `ch` dans laquelle toutes les occurrences de `old` ont été remplacées par `new`

- `ch.split(sep=None)` : Retourne une liste contenant des morceaux de `ch` découpée à chaque occurrence de `sep` (par défaut, la chaîne est decoupée à chaque espace ou retour à la ligne)
- `ch.strip()` : Retourne une version « nettoyée » de `ch` dans laquelle on a enlevé tous les espaces en début et en fin de chaîne
- `ch.format(...)` : Remplace les caractères `{}` dans la chaîne `ch` par le contenu des variables passées en argument

4.4 Formatage des chaînes de caractères

Lorsque l'on souhaite ajouter, dans une chaîne de caractères, du contenu stocké dans une variable, on pourra utiliser la méthode format listée ci-dessus.

Commençons par un exemple :

```
age = 12
prenom = "Micheline"
s = "{} a {} ans".format(prenom, age)
print(s)
```

```
Micheline a 12 ans
```

Ainsi, la méthode `.format()` recherche dans la chaîne de caractères les `{}` et les remplace par les valeurs des variables fournies. Il est possible de maîtriser plus finement la mise en forme de ces variables, et même de les nommer (ce qui peut s'avérer très utile si la chaîne de caractères est longue et inclut de nombreuses variables).

Voici quelques exemples :

```
age_enfant = 12
prenom_enfant = "Micheline"
s = "{prenom} a {age} ans".format(prenom=prenom_enfant, age=age_enfant)
print(s)
```

```
Micheline a 12 ans
```

```
age_enfant = 12
prenom_enfant = "Micheline"
s = "{prenom} a {age:.3f} ans".format(prenom=prenom_enfant, age=age_enfant)
print(s)
```

```
Micheline a 12.000 ans
```

Vous trouverez une présentation plus exhaustive de ces questions dans la documentation Python sur ce point.

4.4.1 Pour aller plus loin : les f-strings

Pour info

Les f-strings existent en Python depuis la version 3.6.

Il existe une autre façon de mettre en forme les chaînes de caractères, qui consiste en l'utilisation de f-strings. Pour définir une f-string, il suffit d'ajouter un f avant la chaîne de caractères :

```
s = f"Ceci est une f-string"
print(s)
```

```
Ceci est une f-string
```

Jusqu'ici, rien de bien révolutionnaire. Mais ces f-strings deviennent fort pratiques dès lors que l'on souhaite ajouter des données issues de variables précédemment définies :

```
age = 12
prenom = "Micheline"
s = f"{prenom} a {age} ans"
print(s)
```

```
Micheline a 12 ans
```

Cette syntaxe est beaucoup plus concise que ce que l'on pouvait avoir en utilisant la méthode `.format()`. On peut même effectuer des calculs à la volée dans les f-strings :

```
age_chat = 12
s = f"Ce chat a {age_chat} ans, ce qui lui fait {age_chat * 6} ans en âge équivalent␣
↪humain"
print(s)
```

```
Ce chat a 12 ans, ce qui lui fait 72 ans en âge équivalent humain
```

En utilisant f-strings et options de formatage, on peut obtenir des sorties alignées avec un code relativement succinct comme dans les exemples suivants :

```
# Exemple 1 : alignement
# Syntaxe : {variable : alignement largeur}
# Alignement < gauche | > droite | ^ centré|
ville = "Rennes"
print(f"La ville de {ville:<10} est en Bretagne.")
print(f"La ville de {ville:>10} est en Bretagne.")
print(f"La ville de {ville:^10} est en Bretagne.")
```

```
La ville de Rennes     est en Bretagne.
La ville de     Rennes est en Bretagne.
La ville de   Rennes   est en Bretagne.
```

```
# Exemple 2 : Affichage des entiers
# Syntaxe : {variable : largeur symbole}
# le symbole d (digit) ou n (numeric) représente un entier
vils = ["Rennes", "Chantepie", "Noyal-sur-Vilaine"]
pops = [215346, 10445, 5820 ]
for v, p in zip(vils, pops):
    print(f"A {v:<17} : il y a {p:7n} habitants." )
```

```
A Rennes            : il y a  215346 habitants.
A Chantepie         : il y a   10445 habitants.
A Noyal-sur-Vilaine : il y a    5820 habitants.
```

```
# Exemple 3 : Affichage des nombres ayant une partie décimale
# Syntaxe : {variable : longtotale.longdeci f}
# Remarque le nombre affiché est arrondi (et non tronqué)
x, y =10/3, 25/7
for longdeci in range(5):
    print(f"Avec {longdeci} décimales, le 1er nombre est |{x:7.{longdeci}f}| le␣
 ↪second est|{y:7.{longdeci}f}|")
```

```
Avec 0 décimales, le 1er nombre est |      3| le second est|      4|
Avec 1 décimales, le 1er nombre est |    3.3| le second est|    3.6|
Avec 2 décimales, le 1er nombre est |   3.33| le second est|   3.57|
Avec 3 décimales, le 1er nombre est |  3.333| le second est|  3.571|
Avec 4 décimales, le 1er nombre est | 3.3333| le second est| 3.5714|
```

4.5 Exercices

Exercice 5.1

Écrivez une fonction qui prenne en argument deux chaînes de caractères `s` et `prefix` et retourne le nombre de mots de la chaîne `s` qui débutent par la chaîne `prefix`.

Solution

```
def compte_prefix(s, prefix):
    compteur = 0
    for mot in s.split():
        if mot.startswith(prefix):
            compteur += 1
    return compteur

print(compte_prefix("la vie est belle au bord du lac", "la"))
```

Exercice 5.2

Écrivez une fonction qui prenne en argument deux chaînes de caractères `s` et `mot_cible` et retourne le nombre

d'occurrences du mot `mot_cible` dans la chaîne `s` en ne tenant pas compte de la casse.

Solution

```
def compte_sans_casse(s, mot_cible):
    compteur = 0
    mot_cible_minuscules = mot_cible.lower()
    for mot in s.split():
        if mot.lower() == mot_cible_minuscules:
            compteur += 1
    return compteur

print(compte_sans_casse("la vie est LA aussi", "la"))
```

Exercice 5.3

Écrivez une fonction qui prenne en argument une liste d'entiers et l'affiche sous le format suivant :

```
L'entier d'indice  0 est |           1|
L'entier d'indice  1 est |          12|
L'entier d'indice  2 est |         123|
L'entier d'indice  3 est |        1234|
L'entier d'indice  4 est |       12345|
L'entier d'indice  5 est |      123456|
L'entier d'indice  6 est |     1234567|
L'entier d'indice  7 est |    12345678|
L'entier d'indice  8 est |   123456789|
L'entier d'indice  9 est |  1234567890|
L'entier d'indice 10 est | 12345678901|
```

Cet affichage correspond à la liste `[1, 12, 123, 1234, 12345, 123456, 1234567, 12345678, 123456789, 1234567890, 12345678901]`

Solution

```
def affiche_liste(liste):
    for i, elt in enumerate(listent):
        print(f"L'entier d'indice {i:2n} est |{elt:12n}|")

listent = [1, 12, 123, 1234, 12345, 123456, 1234567, 12345678, 123456789, 1234567890,␣
 ↪12345678901]
affiche_liste(listent)
```

CHAPTER 5

LES DICTIONNAIRES

Comme une liste, un dictionnaire est une collection de données. Mais, à la différence des listes, les dictionnaires ne sont pas ordonnés (ou, tout du moins, ils le sont dans un ordre qui ne nous est pas naturel). Chaque entrée dans un dictionnaire est une association entre une **clé** (équivalente à un indice pour une liste) et une **valeur**. Alors que les indices d'une liste sont forcément les entiers compris entre 0 et la taille de la liste exclue, les clés d'un dictionnaire sont des valeurs quelconques, la seule contrainte étant qu'on ne peut pas avoir deux fois la même clé dans un dictionnaire. Notamment, ces clés ne sont pas nécessairement des entiers, on utilisera en effet souvent des dictionnaires lorsque l'on souhaite stocker des valeurs associées à des chaînes de caractères (qui seront les clés du dictionnaire).

Pour définir un dictionnaire par ses paires clé-valeur en Python, on peut utiliser la syntaxe suivante :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
print(mon_dico)
```

```
{'a': 123, 'z': 7, 'bbb': None}
```

Ainsi, un dictionnaire vide sera créé comme suit :

```
mon_dico = {}
print(mon_dico)
```

```
{}
```

Attention, pour toutes les versions de Python antérieures à 3.7, l'ordre dans lequel on a entré des paires clé-valeur n'est pas conservé lors de l'affichage ni lors du parcours.

5.1 Modification du contenu d'un dictionnaire

Pour modifier la valeur associée à une clé d'un dictionnaire, la syntaxe est similaire à celle utilisée pour les listes, en remplaçant les indices par les clés :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
mon_dico["a"] = 1000
print(mon_dico)
```

```
{'a': 1000, 'z': 7, 'bbb': None}
```

De même, on peut créer une nouvelle paire clé-valeur en utilisant la même syntaxe :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
mon_dico["c"] = -1
print(mon_dico)
```

```
{'a': 123, 'z': 7, 'bbb': None, 'c': -1}
```

Enfin, pour supprimer une paire clé-valeur d'un dictionnaire, on utilise le mot-clé `del` :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
del mon_dico["a"]
print(mon_dico)
```

```
{'z': 7, 'bbb': None}
```

5.2 Lecture du contenu d'un dictionnaire

Pour lire la valeur associée à une clé du dictionnaire, on peut utiliser la même syntaxe que pour les listes :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
print(mon_dico["a"])
```

```
123
```

Par contre, si la clé demandée n'existe pas, cela génèrera une erreur. Pour éviter cela, on peut utiliser la méthode `get` qui permet de définir une valeur par défaut à retourner si la clé n'existe pas :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
print(mon_dico.get("a", 0))
```

```
123
```

```
print(mon_dico.get("b", 0))
```

```
0
```

5.3 Parcours d'un dictionnaire

Pour parcourir le contenu d'un dictionnaire, il existe, comme pour les listes, trois possibilités.

5.3.1 Parcours par valeurs

Si l'on souhaite uniquement accéder aux valeurs stockées dans le dictionnaire, on utilisera la méthode `values` :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
for val in mon_dico.values():
    print(val)
```

```
123
7
None
```

5.3.2 Parcours par clés

Si l'on souhaite uniquement accéder aux clés stockées dans le dictionnaire, on utilisera la méthode `keys` :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
for cle in mon_dico.keys():
    print(cle)
```

```
a
z
bbb
```

5.3.3 Parcours par couples clés/valeurs

Si l'on souhaite accéder simultanément aux clés stockées dans le dictionnaire et aux valeurs associées, on utilisera la méthode `items` :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
for cle, valeur in mon_dico.items():
    print(cle, valeur)
```

```
a 123
z 7
bbb None
```

Liste ou dictionnaire ?

Lorsque l'on doit stocker des séries de valeurs, on a donc plusieurs choix, et notamment, on peut utiliser une liste ou un dictionnaire. En pratique, on utilisera un dictionnaire dès lors que l'on souhaitera mettre en évidence une correspondance entre clés et valeurs, comme dans l'exemple suivant dans lequel on **associe** à chaque numéro étudiant un entier (pouvant représenter une note, un nombre de photocopies restantes auorisées, ou autre) :

```
{"2170000": 13, "2180000": 23}
```

Si l'on souhaite stocker une **série ordonnée de valeurs**, on préférera l'utilisation d'une liste, comme dans ce qui suit pour une liste de prénoms :

```
["Chloé", "Ali", "Margot", "Antoine"]
```

5.4 Exercices

Exercice 6.1

Écrivez une fonction qui compte le nombre d'occurrences de chacun des mots d'une chaîne de caractères et retourne le résultat sous forme de dictionnaire :

```
# [...]
print(compte_occurrences("la vie est belle c'est la vie"))
# [Sortie] {"c'est": 1, 'la': 2, 'belle': 1, 'est': 1, 'vie': 2}
```

Solution

```
def compte_occurrences(s):
    d = {}
    for mot in s.split():
        d[mot] = d.get(mot, 0) + 1
    return d

print(compte_occurrences("la vie est belle c'est la vie"))
```

Exercice 6.2

Écrivez une fonction qui retourne la somme des *valeurs* d'un dictionnaire fourni en argument, en supposant que toutes les valeurs stockées dans ce dictionnaire soient numériques.

Solution

```
def somme_valeurs(d):
    s = 0
    for v in d.values():
        s += v
    return s

print(somme_valeurs({"a": 12, "zz": 1.5, "AAA": 0}))
```

CHAPTER 6

LES *SETS* ET LES *TUPLES*

Nous avons vu jusqu'à présent deux types qui permettent de stocker des collections de données : les listes et les dictionnaires. Dans ce chapitre, nous nous intéressons à deux autres structures qui permettent elles aussi de stocker des collections : les *sets* et les *tuples*.

6.1 Les *tuples*

Commençons par les tuples, car vous en avez déjà manipulé sans même vous en rendre compte. Prenons l'exemple de la fonction suivante :

```
def comment_tu_tappelles():
    prenom = "Pierre"
    nom = "Lapin"
    return prenom, nom
```

Vous remarquez que cette fonction retourne deux valeurs, séparées par une virgule. Ces deux valeurs forment un *tuple*.

Pour définir un nouveau tuple, on peut écrire :

```
# Les parenthèses sont facultatives
a = 1, 2, 3
b = ("a", "b", "c")

print(a)
print(b)
```

```
(1, 2, 3)
('a', 'b', 'c')
```

Un *tuple* ressemble en de nombreux points à une liste, mais il est impossible de modifier les données d'un *tuple* une fois défini. Il n'est donc pas possible d'ajouter / supprimer des données (comme on le faisait avec `l.append()` ou `l.remove()` pour les listes par exemple). Il n'est pas non plus possible de modifier un élément d'un *tuple* (si `a` est un *tuple*, on ne peut donc pas écrire `a[0] = 5` comme on l'aurait fait pour une liste).

6.2 Les *sets*

Les *sets*, pour leur part, permettent de stocker des ensemble de valeurs, sans ordre et sans répétition. On peut ainsi voir les *sets* comme des dictionnaires qui auraient des clés, mais pas de valeurs associées. Cela fait d'ailleurs écho à la syntaxe utilisée en Python pour définir un set, qui utilise les accolades, comme pour les dictionnaires :

```
c = {1, 2, 3}
d = {"truc", "machin", "chose"}

print(c)
print(d)
```

```
{1, 2, 3}
{'machin', 'truc', 'chose'}
```

Vous pouvez remarquer, dans l'exemple ci-dessus, que, comme pour les dictionnaires, l'ordre dans lequel les éléments sont déclarés dans un *set* n'est pas préservé lors de l'affichage.

Puisque les *sets* ne peuvent pas stocker de valeurs répétées, lorsqu'on convertit une liste en set, on obtient une liste sans doublon (mais on perd la notion d'ordre) :

```
ma_liste = ["truc", "machin", "truc", "chose", "machin", "machin"]
mon_set = set(ma_liste)  # Convertit la liste en set

print(mon_set)
```

```
{'machin', 'truc', 'chose'}
```

Une fois un *set* défini, on ne peut plus modifier les valeurs qu'il contient, mais on peut lui supprimer / ajouter des valeurs :

```
mon_set = {"truc", "machin", "chose"}

# Ajouter un élément
mon_set.add("bidule")

# Supprimer un élément
mon_set.remove("truc")

print(mon_set)
```

```
{'machin', 'bidule', 'chose'}
```

6.3 Tableau récapitulatif

Voici un tableau de ce qu'on peut / ne peut pas faire avec l'un ou l'autre des types de base pour les collections de données :

	Listes	Dictionnaires	*Tuples*	*Sets*
Définition	`[1, 4]`	`{"a": 1, "b": 4}`	`(1, 4)`	`{1, 4}`
Possibilité d'insérer / supprimer des éléments	✓	✓	✗	✓
Possibilité de modifier des éléments	✓	valeurs : ✓ clés : ✗	✗	✗
Possibilité de stocker des valeurs dupliquées	✓	✓*	✓	✗
Valeurs ordonnées	✓	✗**	✓	✗

- * : dans un dictionnaire, les valeurs peuvent être dupliquées, mais pas les clefs
- ** : en fait, à partir de la version 3.7 de Python, les dictionnaires sont ordonnés, mais pour que votre code tourne de manière identique quelle que soit la version de Python utilisée, il est préférable de ne pas faire d'hypothèse sur l'ordre des paires clés-valeurs dans les dictionnaires

CHAPTER 7

INTERACTION AVEC DES FICHIERS

Dans ce chapitre, nous allons présenter des outils de manipulation de fichiers en Python. Un fichier est une unité logique d'informations placée sur une mémoire secondaire (un disque dur, une clef USB, *etc.*).

Quand vous consultez une dossier à l'aide de votre gestionnaire de fichiers, ce dossier peut comprendre plusieurs fichiers. En accédant depuis un programme Python au contenu de fichiers de données, on va pouvoir alimenter le programme avec des données provenant d'un fichier de données indépendant du programme (et non plus seulement des données écrites en dur dans le programmes ou saisies par l'utilisateur directement).

7.1 Manipulation de fichiers en Python avec le module `os`

Lorsque l'on lit ou écrit des fichiers, il est fréquent de vouloir répéter la même opération sur plusieurs fichiers, par exemple sur tous les fichiers avec l'extension `".txt"` d'un répertoire donné. Pour ce faire, on peut utiliser en Python le module `os` qui propose un certain nombre de fonctions standard de manipulation de fichiers. On utilisera notamment la fonction `listdir` de ce module qui permet de lister l'ensemble des fichiers et sous-répertoires contenus dans un répertoire donné :

```
import os

for nom_fichier in os.listdir("donnees"):
    print(nom_fichier)
```

```
truc.txt
machin.csv
```

La fonction `listdir` peut prendre indifféremment un chemin absolu ou relatif (dans notre exemple, il s'agit d'un chemin relatif qui pointe sur le sous-répertoire `"donnees"` contenu dans le répertoire de travail courant du programme).

Si vous exécutez le code ci-dessus et que votre répertoire `"donnees"` n'est pas vide, vous remarquerez que le nom du fichier stocké dans la variable `nom_fichier` ne contient pas le chemin vers ce fichier. Or, si l'on souhaite ensuite ouvrir ce fichier (que ce soit en lecture ou en écriture), il faudra bien spécifier ce chemin. Pour cela, on utilisera la syntaxe suivante :

```
import os

repertoire = "donnees"
for nom_fichier in os.listdir(repertoire):
    nom_complet_fichier = os.path.join(repertoire, nom_fichier)
    print(nom_fichier)
    print(nom_complet_fichier)
    # Ouverture et traitement du fichier dont le nom est contenu dans
    # nom_complet_fichier
```

La fonction `path.join` du module `os` permet d'obtenir le chemin complet vers le fichier à partir du nom du répertoire dans lequel il se trouve et du nom du fichier isolé. Il est préférable d'utiliser cette fonction plutôt que d'effectuer la concaténation des chaînes de caractères correspondantes car la forme des chemins complets dépend du système d'exploitation utilisé, ce que gère intelligemment `path.join`.

7.2 Lecture et écriture de fichiers textuels

7.2.1 Qu'est-ce qu'un fichier textuel ?

Dans ce qui suit, nous traiterons uniquement de la lecture de fichiers textuels. Pour faire simple, nous appellerons dans ce qui suit « fichier textuel » tout fichier dont le contenu est lisible en clair en ouvrant le fichier avec un éditeur de fichiers bruts (comme Notepad++ sous windows, ou gedit sous linux). Pour se faire une idée, nous pouvons (sous unix) utiliser la commande `head` qui permet d'afficher le début d'un fichier :

```
!head entete.csv
```

```
NOM;PRENOM;AGE
Lemarchand;John;23
Trias;Anne;
```

Et si l'on avait cherché à lire le contenu d'un fichier non textuel, on aurait obtenu des caractères spéciaux ne pouvant être interprétés comme du texte.

Dans ce chapitre, nous ne nous intéressons donc pas à ce dernier type de fichiers mais nous nous concentrerons sur la lecture/écriture de fichiers textuels par un programme Python.

7.2.2 Encodage des fichiers

Un premier élément qu'il est nécessaire de maîtriser pour lire ou écrire des fichiers textuels est la notion d'encodage. Il faut savoir qu'il existe plusieurs façons d'encoder un texte. Nous nous focaliserons ici sur les deux encodages que vous êtes les plus susceptibles de rencontrer (mais sachez qu'il en existe bien d'autres) :

- l'encodage Unicode 8 bits (UTF-8), dont le code en python est `"utf-8"` ;
- l'encodage Latin-1 (ISO-8859-1) dont le code en python est `"iso-8859-1"`.

La principale différence entre ces deux encodage réside dans leur façon de coder les accents. Ainsi, si le texte que vous lisez/écrivez ne contient aucun accent ou caractère spécial, il est probable que la question de l'encodage ne soit pas problématique dans votre cas. Au contraire, s'il est possible que vous utilisiez de tels caractères, il faudra bien faire attention à l'encodage utilisé, que vous spécifierez à l'ouverture du fichier. Si votre programme doit lire un fichier, il faudra donc vous assurer de l'encodage associé à ce fichier (en l'ouvrant par exemple avec un éditeur de texte qui soit suffisamment avancé pour vous fournir cette information). Si vous écrivez un programme qui écrit un fichier, il faudra

vous poser la question de l'utilisation future qui sera faite de ce fichier : s'il est amené à être ouvert par un autre utilisateur, il serait pertinent de vous demander quel encodage sera le moins problématique pour cet utilisateur, par exemple.

Si vous n'avez pas de contrainte extérieure pour ce qui est de l'encodage, vous utiliserez l'encodage UTF-8 par défaut.

7.2.3 Lecture de fichiers textuels

Ce que nous appelons lecture de fichiers textuels en Python consiste à copier le contenu d'un fichier dans une (ou plusieurs) chaîne(s) de caractères. Cela implique deux étapes en Python :

1. ouvrir le fichier en lecture ;
2. parcourir le contenu du fichier.

La première étape d'ouverture du fichier en lecture est commune à tous les types de fichiers textuels. En supposant que le nom du fichier à ouvrir soit stocké sous forme de chaîne de caractères dans la variable `nom_fichier`, le code suivant ouvre un fichier en lecture avec l'encodage UTF-8 et stocke dans la variable `fp` un pointeur sur l'endroit où nous sommes rendus dans notre lecture du fichier (pour l'instant, le début du fichier) :

```
fp = open(nom_fichier, "r", encoding="utf-8")
```

Le second argument (`"r"`) indique que le fichier doit être ouvert en mode *read*, donc en lecture.

Fichiers textuels génériques

Nous appelons ici « fichier textuel générique » un fichier dont le contenu n'a pas de structure particulière (par opposition aux fichiers CSV ou JSON présentés plus bas, par exemple). C'est le cas par exemple de fichiers contenant du texte libre. Ces fichiers, une fois ouverts en lecture, peuvent être lus ligne par ligne à l'aide de la boucle suivante :

```
fp = open(nom_fichier, "r", encoding="utf-8")
for ligne in fp.readlines():
    print(ligne)
```

Ici, la variable `ligne`, de type chaîne de caractères, contiendra successivement le texte de chacune des lignes du fichier considéré.

Fichiers *Comma-Separated Values* (CSV)

Les fichiers *Comma-Separated Values* (CSV) permettent de stocker des données organisées sous la forme de tableaux dans des fichiers textuels. À l'origine, ces fichiers étaient organisés par ligne et au sein de chaque ligne les cellules du tableau (correspondant aux différentes colonnes) étaient séparées par des virgules (d'où le nom de ce type de fichiers). Aujourd'hui, la définition de ce format (lien) est plus générale que cela et différents délimiteurs sont acceptés. Pour manipuler ces fichiers, il existe en Python un module dédié, appelé `csv`. Ce module contient notamment une fonction `reader` permettant de simplifier la lecture de fichiers CSV. La syntaxe d'utilisation de cette fonction est la suivante (vous remarquerez la présence de l'attribut `delimiter`) :

```
import csv

nom_fichier = "simple.csv"

# Contenu supposé du fichier :
# 1;2;3;4;5
# a;b;c;d;e
```

(suite sur la page suivante)

(suite de la page précédente)

```
# xx;xx;xx;xx;xx
# 0.4;0.5;0.7;0.8;0.9

fp = open(nom_fichier, "r", encoding="utf-8")
for ligne in csv.reader(fp, delimiter=";"):
    for cellule in ligne:
        print(cellule)
    print("Fin de ligne")
```

```
1
2
3
4
5
Fin de ligne
a
b
c
d
e
Fin de ligne
xx
xx
xx
xx
xx
Fin de ligne
0.4
0.5
0.7
0.8
0.9
Fin de ligne
```

On remarque ici que, contrairement au cas de fichiers textuels génériques, la variable de boucle `ligne` n'est plus une chaîne de caractères mais une liste de chaînes de caractères. Les éléments de cette liste sont les cellules du tableau représenté par le fichier CSV.

Cas des fichiers à en-tête

Souvent, les fichiers CSV comprennent une première ligne d'en-tête, comme dans l'exemple suivant :

```
NOM;PRENOM;AGE
Lemarchand;John;23
Trias;Anne;
```

Si l'on souhaite que, lors de la lecture du fichier CSV, chaque ligne soit représentée par un dictionnaire dont les clés sont les noms de colonnes (lus dans l'en-tête) et les valeurs associées sont celles lues dans la ligne courante, on utilisera `csv.DictReader` au lieu de `csv.reader` :

```
import csv

nom_fichier = "entete.csv"
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Contenu supposé du fichier :
# NOM;PRENOM;AGE
# Lemarchand;John;23
# Trias;Anne;

fp = open(nom_fichier, "r", encoding="utf-8")
for ligne in csv.DictReader(fp, delimiter=";"):
    for cle, valeur in ligne.items():
        print(cle, valeur)
    print("--Fin de ligne--")
```

```
NOM Lemarchand
PRENOM John
AGE 23
--Fin de ligne--
NOM Trias
PRENOM Anne
AGE
--Fin de ligne--
```

Un peu de magie...

Dans certains cas, on ne sait pas à l'avance quel délimiteur est utilisé pour le fichier CSV à lire. On peut demander au module CSV de deviner le *dialecte*[1] d'un fichier en lisant le début de ce fichier. Dans ce cas, la lecture du fichier se fera en 4 étapes :

1. Ouverture du fichier en lecture ;
2. Lecture des *n* premiers caractères du fichier pour tenter de deviner son dialecte ;
3. « Rembobinage » du fichier pour recommencer la lecture au début ;
4. Lecture du fichier en utilisant le dialecte détecté à l'étape 2.

Le choix du paramètre *n* doit être un compromis : il faut lire suffisamment de caractères pour que la détection de dialecte soit fiable, tout en sachant que lire beaucoup de caractères prendra du temps. En pratique, lire les 1000 premiers caractères d'un fichier est souvent suffisant pour déterminer son dialecte.

On obtient alors une syntaxe du type :

```
import csv

nom_fichier = "simple.csv"

# Contenu supposé du fichier :
# 1,2,3
# a,b

fp = open(nom_fichier, "r", encoding="utf-8")  # Étape 1.
dialecte = csv.Sniffer().sniff(fp.read(1000))  # Étape 2.
fp.seek(0)                                     # Étape 3. À ne pas oublier !
for ligne in csv.reader(fp, dialect=dialecte): # Étape 4.
```

(suite sur la page suivante)

[1] Le *dialecte* d'un fichier CSV définit, en fait, bien plus que le caractère de séparation des cellules, comme décrit dans ce document.

(suite de la page précédente)

```
    for cellule in ligne:
        print(cellule)
    print("Fin de ligne")
```

```
1
2
3
4
5
Fin de ligne
a
b
c
d
e
Fin de ligne
xx
xx
xx
xx
xx
Fin de ligne
0.4
0.5
0.7
0.8
0.9
Fin de ligne
```

Fichiers *JavaScript Object Notation* (JSON)

Les fichiers *JavaScript Object Notation* (JSON) permettent de stocker des données structurées (par exemple avec une organisation hiérarchique). Un document JSON s'apparente à un dictionnaire en Python (à la nuance près que les clés d'un document JSON sont forcément des chaînes de caractères). Voici un exemple de document JSON :

```
{
    "num_etudiant": "21300000",
    "notes": [12, 5, 14],
    "date_de_naissance": {
        "jour": 1,
        "mois": 1,
        "annee": 1995
    }
}
```

En Python, pour lire de tels fichiers, on dispose du module `json` qui contient une fonction `load` :

```
import json

nom_fichier = "simple.json"

# Contenu supposé du fichier :
# {
```

(suite sur la page suivante)

(suite de la page précédente)

```
#     "num_etudiant": "21300000",
#     "notes": [12, 5, 14],
#     "date_de_naissance": {
#         "jour": 1,
#         "mois": 1,
#         "annee": 1995
#     }
# }

fp = open(nom_fichier, "r", encoding="utf-8")
d = json.load(fp)
print(d)
```

```
{'num_etudiant': '21300000', 'notes': [12, 5, 14], 'date_de_naissance': {'jour': 1,
↪ 'mois': 1, 'annee': 1995}}
```

Il est à noter qu'un fichier JSON peut également contenir une liste de dictionnaires, comme dans l'exemple suivant :

```
[{
    "num_etudiant": "21300000",
    "notes": [12, 5, 14],
    "date_de_naissance": {
        "jour": 1,
        "mois": 1,
        "annee": 1995
    }
},
{
    "num_etudiant": "21300001",
    "notes": [14],
    "date_de_naissance": {
        "jour": 1,
        "mois": 6,
        "annee": 1989
    }
}]
```

Dans ce cas, `json.load` retournera une liste de dictionnaires au lieu d'un dictionnaire, bien évidemment.

Enfin, si l'on a stocké dans une variable une chaîne de caractères dont le contenu correspond à un document JSON, on peut également la transformer en dictionnaire (ou en liste de dictionnaires) à l'aide de la fonction `json.loads` (attention au « s » final) :

```
ch = '{"num_etudiant": "21300000",  "notes": [12, 5, 14]}'
d = json.loads(ch)  # loads : load (from) string
print(d)
```

```
{'num_etudiant': '21300000', 'notes': [12, 5, 14]}
```

7.2.4 Écriture de fichiers textuels

Ce que nous appellons écriture de fichiers textuels en Python consiste à copier le contenu d'une (ou plusieurs) chaîne(s) de caractères dans un fichier. Cela implique trois étapes en Python :

1. ouvrir le fichier en écriture ;
2. ajouter du contenu dans le fichier ;
3. fermer le fichier.

La première étape d'ouverture du fichier en écriture est commune à tous les types de fichiers textuels. En supposant que le nom du fichier à ouvrir est stocké sous forme de chaîne de caractères dans la variable `nom_fichier`, le code suivant ouvre un fichier en écriture avec l'encodage UTF-8 et stocke dans la variable `fp` un pointeur sur l'endroit où nous sommes rendus dans notre écriture du fichier (pour l'instant, le début du fichier) :

```
fp = open(nom_fichier, "w", encoding="utf-8", newline="\n")
```

Le second argument (`"w"`) indique que le fichier doit être ouvert en mode *write*, donc en écriture.

Si le fichier en question existait déjà, son contenu est tout d'abord écrasé et on repart d'un fichier vide. Si l'on souhaite au contraire ajouter du texte à la fin d'un fichier existant, on utilisera le mode *append*, symbolisé par la lettre `"a"` :

```
fp = open(nom_fichier, "a", encoding="utf-8", newline="\n")
```

Une fois les instructions d'écriture exécutées (voir plus bas), on doit fermer le fichier pour s'assurer que l'écriture sera effective :

```
fp.close()
```

Il est à noter que l'on peut, dans certains cas, se dispenser de fermer explicitement le fichier. Par exemple, si notre code est inclus dans un script Python, dès la fin de l'exécution du script, tous les fichiers ouverts en écriture par le script sont automatiquement fermés.

Fichiers textuels génériques

Pour ajouter du contenu à un fichier pointé par la variable `fp`, il suffit ensuite d'utiliser la méthode `write` :

```
fp.write("La vie est belle\n")
```

Notez que, contrairement à la fonction `print` à laquelle vous êtes habitué, la méthode `write` ne rajoute pas de caractère de fin de ligne après la chaîne de caractères passée en argument, il faut donc inclure ce caractère `"\n"` à la fin de la chaîne de caractères passée en argument, si vous souhaitez inclure un retour à la ligne.

Fichiers CSV

Le module `csv` déjà cité plus haut contient également une fonction `writer` permettant de simplifier l'écriture de fichiers CSV. La syntaxe d'utilisation de cette fonction est la suivante :

```
import csv

nom_fichier = "ecriture.csv"

fp = open(nom_fichier, "w", encoding="utf-8", newline="\n")
```

(suite sur la page suivante)

(suite de la page précédente)

```
csvfp = csv.writer(fp, delimiter=";")
csvfp.writerow([1, 5, 7])
csvfp.writerow([2, 3])
fp.close()
# Après cela, le fichier contiendra les lignes suivantes :
# 1;5;7
# 2;3
```

La méthode `writerow` prend donc une liste en argument et écrit dans le fichier les éléments de cette liste, séparés par le délimiteur `";"` spécifié lors de l'appel à la fonction `writer`. Le retour à la ligne est écrit directement par la méthode `writerow`, vous n'avez pas à vous en occuper.

Fichiers JSON

Le module `json` déjà cité plus haut contient également une fonction `dump` permettant d'écrire le contenu d'un dictionnaire (ou d'une liste de dictionnaires) dans un fichier JSON. La syntaxe d'utilisation de cette fonction est la suivante :

```
import json

nom_fichier = "ecriture.json"

liste = [
    {"a": 5},
    {"b": 3, "a": 7}
]

fp = open(nom_fichier, "w", encoding="utf-8", newline="\n")
json.dump(liste, fp)
fp.close()
# Après cela, le fichier contiendra la ligne suivante :
# [{"a": 5}, {"b": 3, "a": 7}]
```

Vous pouvez vous référer à la documentation de cette fonction pour maîtriser plus finement la mise en forme du contenu du fichier de sortie.

7.3 Bonus : utilisation de `with`

Il existe en Python un moyen très efficace de maîtriser la durée de vie d'une variable : il s'agit de la directive `with`, dont la syntaxe est la suivante :

```
with EXPRESSION as VARIABLE:
    BLOC_DE_CODE
```

Le code ci-dessus signifie que l'on va créer une variable `VARIABLE` qui va stocker la valeur de retour de `EXPRESSION` (ceci n'est pas tout à fait exact, mais dans l'exemple qui nous intéresse, cela revient au même), puis le code situé dans `BLOC_DE_CODE` sera exécuté et, à la fin de ce bloc de code, `VARIABLE` sera détruite.

Cette syntaxe peut s'avérer fort utile lorsque l'on manipule des fichiers, parce que les pointeurs de fichiers, lorsqu'ils sont détruits, ferment automatiquement le fichier en question, et il n'est donc pas nécessaire de faire appel à la méthode `.close()` explicitement.

On pourra donc écrire :

```
with open("simple.csv", "r") as fp:
    for texte in fp.readlines():
        print(texte)
```

```
1;2;3;4;5

a;b;c;d;e

xx;xx;xx;xx;xx

0.4;0.5;0.7;0.8;0.9
```

et le fichier sera tout de même fermé correctement dès que l'on sortira du bloc `with`.

7.4 Exercices

Exercice 8.1

Écrivez une fonction qui affiche, pour chaque fichier d'extension `".txt"` d'un répertoire passé en argument, le nom du fichier ainsi que son nombre de lignes.

Solution

```
import os

def nb_lignes(nom_fichier):
    n = 0
    fp = open(nom_fichier, "r")
    for ligne in fp.readlines():
        n += 1
    return n

def nb_lignes_repertoire(repertoire):
    for nom_fichier in os.listdir(repertoire):
        if nom_fichier.endswith(".txt"):
            nom_complet_fichier = os.path.join(repertoire, nom_fichier)
            n = nb_lignes(nom_complet_fichier)
            print(nom_complet_fichier, n)

nb_lignes_repertoire(".")
```

Exercice 8.2

Écrivez une fonction qui retourne le nombre de fichiers présents dans un répertoire dont le nom est passé en argument. Vous pourrez vous aider pour cela de la documentation du sous-module `path` du module `os` (lien).

Solution

```python
import os

def compte_fichiers(repertoire):
    compteur = 0
    for f in os.listdir(repertoire):
        if os.path.isfile(os.path.join(repertoire, f)):
            compteur += 1
    return compteur

print(compte_fichiers("."))
```

CHAPTER 8

RÉCUPÉRATION DE DONNÉES À PARTIR D'API WEB

De nombreux services web fournissent des API (*Application Programming Interface*) pour mettre des données à disposition du grand public. Le principe de fonctionnement de ces API est le suivant : l'utilisateur effectue une requête sous la forme d'une requête HTTP, le service web met en forme les données correspondant à la requête et les renvoie à l'utilisateur, dans un format défini à l'avance.

Voici une liste (très loin d'être exhaustive) d'API web d'accès aux données :

- Google Maps
  - Directions API : permet de calculer des itinéraires ;
  - Elevation API : permet de calculer l'altitude d'un point sur le globe terrestre ;
  - Distance Matrix API : permet de calculer des distances entre points du globe ;
  - Geocoding API : permet d'associer une coordonnée GPS à une adresse.
- Twitter
  - Twitter API : permet de récupérer des informations sur les utilisateurs du réseau et leurs *tweets*.
- Facebook
  - Facebook Graph API : permet de récupérer des informations sur des utilisateurs Facebook .
- STAR (Transports en commun rennais)
  - Horaires des bus ;
  - Disponibilité des vélos dans les relais VéloStar.

Pour manipuler en Python de telles données, il faudra donc être capable :

1. d'envoyer une requête HTTP et de récupérer le résultat ;
2. de transformer le résultat en une variable Python facilement manipulable.

Pour ce qui est du second point, la plupart des API web offrent la possibilité de récupérer les données au format JSON. Nous avons vu précédemment dans ce cours que ce format était facilement manipulable en Python, notamment parce qu'il est très proche de la notion de dictionnaire. Ce chapitre se focalise donc sur la réalisation de requêtes HTTP en Python.

8.1 Requêtes HTTP en Python

8.1.1 Format d'une requête HTTP

Dans un premier temps, étudions le format d'une requête HTTP, telle que vous en effectuez des dizaines chaque jour, par l'intermédiaire de votre navigateur web. Lorsque vous entrez dans la barre d'adresse de votre navigateur l'URL suivante :

```
http://people.irisa.fr/Romain.Tavenard/index.php?page=3
```

votre navigateur va envoyer une requête au serveur concerné (cette requête ne contiendra pas uniquement l'URL visée mais aussi d'autres informations sur lesquelles nous ne nous attarderons pas ici). Dans l'URL précédente, on distingue 4 sous parties :

- `http://` indique le protocole à utiliser pour effectuer la requête (ici HTTP). Dans ce chapitre, nous ne nous intéresserons qu'aux protocoles HTTP et HTTPS (version sécurisée du protocole HTTP) ;
- `people.irisa.fr` est le nom de domaine du serveur (*ie.* de la machine) à contacter pour obtenir une réponse ;
- `/Romain.Tavenard/index.php` indique le chemin du fichier à récupérer sur cette machine ;
- `?page=3` indique que l'on doit passer la valeur `3` au paramètre `page` lors de la requête.

De la même façon, lors d'un appel à une API web, on spécifiera le protocole à utiliser, la machine à contacter, le chemin vers la ressource voulue et un certain nombre de paramètres qui décriront notre requête. Voici un exemple de requête à une API web (l'API Google Maps Directions en l'occurrence) :

```
https://maps.googleapis.com/maps/api/directions/json?origin=Toronto&
↪destination=Montreal
```

Vous pouvez copier/coller cette URL dans la barre d'adresse de votre navigateur et observer ce que vous obtenez en retour. Observez que le résultat de cette requête est au format JSON. En fait, si vous étudiez plus précisément l'URL fournie, vous verrez que c'est nous qui avons demandé à obtenir le résultat dans ce format. De plus, on a spécifié dans l'URL que l'on souhaitait obtenir les informations d'itinéraire pour aller de Toronto (paramètre `origin`) à Montreal (paramètre `destination`).

Vous devez aussi remarquer que, en réponse à cette requête, l'API Google Maps renvoie en fait un message d'erreur. En effet, pour être autorisé à utiliser cette API, il faut disposer d'une clé d'API et renseigner cette clé sous la forme d'un paramètre supplémentaire (nommé `key` dans les API Google Maps par exemple). Ainsi, la requête précédente deviendrait :

```
https://maps.googleapis.com/maps/api/directions/json?origin=Toronto&
↪destination=Montreal&key=VOTRE_CLE
```

dans laquelle vous devrez remplacer `VOTRE_CLE` par une clé que vous aurez préalablement générée et qui vous permettra d'utiliser le service web de manière authentifiée. Pour créer une clef d'API, il faut se rendre sur l'interface développeur de l'API concernée (ici pour l'API *Google Maps Directions* par exemple).

8.1.2 Utilisation du module `requests`

Les requêtes HTTP en (très) bref

Dans le protocole HTTP, il existe plusieurs types de requêtes pour réaliser l'échange entre le client et le serveur. En particulier les requêtes de type GET sont très utilisées lorsque le client demande une ressource au serveur. Il s'agit d'une requête de téléchargement d'un document. Il est possible de transmettre des paramètres pour filtrer la réponse ; dans ce cas, les paramètres seront transférés « en clair » (dans l'URL utilisée pour la requête).

Les requêtes de type POST permettent comme GET de télécharger un document du serveur vers le client mais avec un plus de sophistication : les paramètres sont masqués et il est possible de demander de mettre à jour des données sur le serveur à l'occasion de la requête.

Il existe d'autres requêtes HTTP que nous ne détaillons pas ici.

Installer un paquet Python

Le module `requests` ne fait pas partie de la librairie standard en Python. Il faut donc l'installer avant de pouvoir l'utiliser. Pour ce faire, on peut utiliser le gestionnaire de paquets `pip`.

Si vous utilisez Anaconda, la documentation en ligne disponible à cette adresse explique la marche à suivre pour installer de nouveaux paquets dans votre version de Python fournie par Anaconda. Si vous utilisez plutôt l'IDE PyCharm et une version de Python non fournie par Anaconda, vous pourrez trouver de la documentation à cette adresse.

La section précédente proposait un rappel sur le format des requêtes HTTP et vous avez été invités à effectuer des requêtes HTTP à l'aide de votre navigateur. Si maintenant on souhaite récupérer de manière automatique le résultat d'une requête HTTP pour le manipuler en Python, le plus commode est d'effectuer la requête HTTP depuis Python. Pour cela, on utilise le module `requests`. Ce module contient notamment une fonction `get` qui permet d'effectuer des requêtes HTTP de type GET (je vous laisse deviner le nom de la fonction qui permet d'effectuer des requêtes HTTP POST :) :

```
import requests

url = "http://my-json-server.typicode.com/rtavenar/fake_api/tasks"

reponse = requests.get(url)
print(reponse)
```

```
<Response [200]>
```

On voit ici que l'on a reçu une réponse de code 200, ce qui signifie que la requête s'est déroulée correctement.

Codes de retour HTTP

Voici quelques codes de retour de requêtes HTTP qui peuvent vous être utiles :

- 20x : la transaction s'est bien déroulée
  - *ex.* 200 : la requête s'est effectuée correctement
- 40x : erreur « due au client »
  - *ex.* 404 : page non trouvée

- 50x : erreur « due au serveur »
  - *ex.* 504 : Temps imparti écoulé

```
contenu_txt = reponse.text
print(type(contenu_txt))
```

```
<class 'str'>
```

```
contenu = reponse.json()
print(type(contenu))
```

```
<class 'list'>
```

```
print(contenu)
```

```
[{'userId': 1, 'id': 1, 'title': 'delectus aut autem', 'completed': False}, {
↪'userId': 1, 'id': 2, 'title': 'quis ut nam facilis et officia qui', 'completed
↪': False}, {'userId': 1, 'id': 3, 'title': 'fugiat veniam minus', 'completed':␣
↪False}, {'userId': 1, 'id': 4, 'title': 'et porro tempora', 'completed': True}, {
↪'userId': 1, 'id': 8, 'title': 'quo adipisci enim quam ut ab', 'completed': True}
↪, {'userId': 3, 'id': 44, 'title': 'cum debitis quis accusamus doloremque ipsa␣
↪natus sapiente omnis', 'completed': True}, {'userId': 3, 'id': 45, 'title':
↪'velit soluta adipisci molestias reiciendis harum', 'completed': False}, {'userId
↪': 3, 'id': 46, 'title': 'vel voluptatem repellat nihil placeat corporis',
↪'completed': False}]
```

On voit ici qu'il est possible d'obtenir le résultat de notre requête sous deux formes : le texte brut du résultat qui est stocké dans `reponse.text` et la version mise en forme (sous la forme de dictionnaire ou de liste) de ce résultat que l'on obtient via `reponse.json()`.

De plus, si l'on souhaite passer des paramètres à la requête HTTP (ce qui se trouvait après le symbole `?` dans les URL ci-dessus), il est possible de le faire lors de l'appel à `requests.get` :

```
import requests

url = "http://my-json-server.typicode.com/rtavenar/fake_api/tasks"

reponse = requests.get(url, params="userId=3")
contenu = reponse.json()
print(contenu)
```

```
[{'userId': 3, 'id': 44, 'title': 'cum debitis quis accusamus doloremque ipsa␣
↪natus sapiente omnis', 'completed': True}, {'userId': 3, 'id': 45, 'title':
↪'velit soluta adipisci molestias reiciendis harum', 'completed': False}, {'userId
↪': 3, 'id': 46, 'title': 'vel voluptatem repellat nihil placeat corporis',
↪'completed': False}]
```

Le code ci-dessus correspond ainsi à ce que vous obtiendriez dans votre navigateur en entrant l'URL http://my-json-server.typicode.com/rtavenar/fake_api/tasks?userId=3.

En pratique, dans de nombreux cas, des modules Python existent pour permettre d'utiliser les API grand public sans avoir à gérer les requêtes HTTP directement. C'est par exemple le cas des modules `tweepy` (pour l'API Twitter) ou `graphh`

(qui permet d'accéder à l'API GraphHopper qui est un équivalent libre de Google Maps)[1].

8.2 Exercice

Exercice 9.1

Écrivez une fonction qui prenne en entrée une liste de `userId` et affiche l'ensemble des entrées de l'API http://my-json-server.typicode.com/rtavenar/fake_api/tasks pour lesquelles l'attribut `completed` vaut `True`.

Solution

- Solution 1

```
import requests

def affiche_api(liste_userId):
    url = "http://my-json-server.typicode.com/rtavenar/fake_api/tasks"
    contenu = requests.get(url)
    list_taches = contenu.json()
    for tache in list_taches:
        if tache["completed"] and tache["userId"] in liste_userId:
            print(tache)

affiche_api([1, 3])
```

- Solution 2 : en utilisant les paramètres d'URL

```
import requests

def affiche_api(liste_userId):
    url = "http://my-json-server.typicode.com/rtavenar/fake_api/tasks"
    contenu = requests.get(url, params="completed=true")
    list_taches = contenu.json()
    for tache in list_taches:
        if tache["userId"] in liste_userId:
            print(tache)

affiche_api([1, 3])
```

[1] Notez que le module `graphh` a été développé par d'anciens étudiants de Licence 2 et Licence 3 MIASHS de l'Université de Rennes 2.

CHAPTER 9

NUMPY ET LE CALCUL MATRICIEL

Ce chapitre traite d'un module en particulier : le module `numpy`. `numpy` est un raccourci pour *Numerical Python* : cette librairie a donc pour vocation de fournir des outils de calcul numérique en Python. Ce module permet donc de manipuler des vecteurs et des matrices (voire des tenseurs d'ordre supérieur).

Avant toute chose, vous devrez importer ce module :

```
import numpy as np
```

Vous remarquez ici que lors de l'import, le nom du module est renommé en `np` : il s'agit d'une habitude très répandue qui permet de ne pas surcharger inutilement la suite de votre code.

De plus, sachez que ce chapitre est très succinct et très loin de couvrir l'ensemble des fonctionnalités `numpy`, vous êtes donc fortement incités à utiliser la documentation `numpy` pour trouver ce qui pourrait vous être utile pour votre usage.

9.1 Tableaux multi-dimensionnels

Les tableaux multi-dimensionnels sont les objets de base en `numpy`. On peut créer un vecteur comme suit :

```
vec = np.array([1, 4, 6, 7])

print(vec.ndim)
```

```
1
```

Dans ce chapitre, nous allons nous concentrer sur des vecteurs (`ndim = 1`, comme dans l'exemple ci-dessus) et des matrices (`ndim = 2`), mais il faut savoir que les tableaux multi-dimensionnels `numpy` peuvent stocker des tenseurs d'ordre quelconque.

Voici quelques exemples de manipulations élémentaires sur les tableaux `numpy` :

```
# Multiplication par une constante
print(2.5 * vec)

# Accès au type des données stockées
print(vec.dtype)

# Accès à la taille du vecteur
print(vec.shape)

# Définition d'une matrice
A = np.array([[0, 1], [2, 3]])
print(A)

# Transposition
print(A.T)
```

```
[ 2.5 10.  15.  17.5]
int64
(4,)
[[0 1]
 [2 3]]
[[0 2]
 [1 3]]
```

On remarque d'ores et déjà que les tableaux `numpy` ont un type associé. On ne pourra donc pas stocker dans un tableau `numpy` des données de types hétérogènes, comme on peut le faire dans le cas des listes Python par exemple.

9.2 Produit matriciel et opérations « élément à élément »

Une chose importante à comprendre en `numpy` est que le produit par défaut entre deux tableaux est le produit élément à élément, et non pas le produit matriciel, comme on peut le voir dans cet exemple :

```
A = np.array([[0, 1], [2, 3]])
print(A)
```

```
[[0 1]
 [2 3]]
```

```
I = np.array([[1, 0], [0, 1]])
print(I)
```

```
[[1 0]
 [0 1]]
```

```
A * I
```

```
array([[0, 0],
       [0, 3]])
```

Il est toutefois possible d'effectuer un produit matriciel à l'aide de l'opérateur `@`, et alors on retrouve bien la propriété attendue qui est que le produit de `A` par la matrice identité retourne la matrice `A` :

```
A @ I
```

```
array([[0, 1],
       [2, 3]])
```

De même, lorsqu'on écrit `A ** 2`, on obtient l'élévation au carré de chacun des éléments de `A` et non pas le produit de `A` par lui-même :

```
A ** 2
```

```
array([[0, 1],
       [4, 9]])
```

```
A @ A
```

```
array([[ 2,  3],
       [ 6, 11]])
```

Les choses sont plus simples pour l'addition puisqu'il n'y a alors pas de confusion possible :

```
A + A
```

```
array([[0, 2],
       [4, 6]])
```

9.3 Constructeurs de tableaux usuels

`numpy` permet de définir très simplement des tableaux remplis de 0, de 1, la matrice identité, ou des séquences de valeurs :

```
np.zeros((2, 3))  # (2, 3) est la taille de la matrice à produire
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
np.ones((2, 3))  # (2, 3) est la taille de la matrice à produire
```

```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

```
np.eye(2)  # eye -> matrice identité
```

```
array([[1., 0.],
       [0., 1.]])
```

```
np.arange(10)  # arange -> équivalent de range pour les listes
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
# Vecteur de 11 valeurs espacées régulièrement entre 0 et 1
np.linspace(0, 1, 11)
```

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

```
m = np.arange(10)
# Redimensionner m pour qu'il ait 5 lignes et 2 colonnes
m.reshape((5, 2))
```

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
```

9.4 Accès à des sous-parties des tableaux

Comme pour les listes, les tableaux `numpy` peuvent être accédés par « tranches » (*slice*), comme dans les exemples suivants :

```
M = np.array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14], [15, 16, 17, 18,
 ↪ 19]])
M
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

```
# Indices de ligne jusqu'à 2 (exclu)
# Indices de colonne à partir de 3 (inclus)
M[:2, 3:]
```

```
array([[3, 4],
       [8, 9]])
```

```
# Indices de colonne de 1 (inclus) à 3 (exclu)
# Tous les indices de colonne
M[1:3, :]
```

```
array([[ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

On peut également utiliser des listes (ou des `ndarray`) d'indices pour accéder aux éléments d'un tableau situés aux indices correspondants :

```
liste_indices = [0, 4]
v = np.array([1, 5, 7, 9, 12])
v[liste_indices]
```

```
array([ 1, 12])
```

Ici, `v[liste_indices]` est un `ndarray` constitué des éléments `[v[0], v[4]]`. De la même façon, si les indices sont stockés dans une structure à deux dimensions :

```
indices = np.array([[0, 4],
                    [1, 3]])
v[indices]
```

```
array([[ 1, 12],
       [ 5,  9]])
```

On voit que, ici, `v[indices]` est un `ndarray` à 2 dimensions constitué des éléments

```
v[0] v[4]
v[1] v[3]
```

9.5 Opérations élémentaires sur les tableaux

Une fois un tableau défini, on peut très facilement calculer :

- la somme de ses éléments :

```
np.sum(M)  # Peut aussi s'écrire M.sum()
```

```
190
```

- sa plus petite / plus grande valeur :

```
np.min(M)  # Peut aussi s'écrire M.min()
```

```
0
```

```
np.max(M)  # Peut aussi s'écrire M.max()
```

```
19
```

- la moyenne / l'écart-type de ses éléments :

```
np.mean(M)  # Peut aussi s'écrire M.mean()
```

```
9.5
```

```
np.std(M)  # Peut aussi s'écrire M.std()
```

```
5.766281297335398
```

Il est à noter que pour toutes ces opérations, deux syntaxes co-existent :

```
print(np.min(M))
print(M.min())
```

```
0
0
```

De plus, on peut également effectuer ces opérations ligne par ligne, ou colonne par colonne, comme ci-dessous :

```
# On somme sur les lignes (dimension numéro 0)
# Donc on obtient un résultat par colonne
M.sum(axis=0)
```

```
array([30, 34, 38, 42, 46])
```

Enfin, on peut très facilement créer des masques binaires, tels que :

```
M > 5  # Vaut True à chaque position telle que l'élément correspondant dans M est > 5
```

```
array([[False, False, False, False, False],
       [False,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True]])
```

Ce qui permet de compter simplement le nombre de valeurs d'un tableau vérifiant une condition :

```
np.sum(M > 5)
```

```
14
```

9.6 Bonnes pratiques

Vous devrez, tant que faire se peut, utiliser les fonctions prédéfinies en `numpy` pour vos manipulations de tableaux multi-dimensionnels, plutôt que de recoder les opérations élémentaires. Il est notamment fortement déconseillé de parcourir les valeurs d'un tableau `numpy` au sein d'une boucle, pour des raisons d'efficacité (autrement dit, de temps de calcul), comme illustré ci-dessous :

```
vec = np.ones((100, 10))
```

```
%%timeit
# timeit permet de mesurer le temps d'exécution d'un morceau de code
vec.sum()
```

```
2.28 µs ± 7.98 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%%timeit
# timeit permet de mesurer le temps d'exécution d'un morceau de code
s = 0
for v in vec:  # À ne JAMAIS faire !
    s += v
```

```
86.7 µs ± 30.6 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

9.7 Exercices

Exercice 10.1

Calculez, en `numpy`, la somme des `n` premiers entiers, pour `n` fixé.

Solution

```
import numpy as np

n = 10
print(np.arange(n).sum())
```

Exercice 10.2

Supposons qu'on ait stocké dans le tableau suivant les notes reçues par 2 étudiants à 3 examens :

```
notes = np.array(
  [[10, 12],
   [15, 16],
   [18, 12]]
)
```

1. Calculez la moyenne de chacun des deux étudiants.
2. Calculez le nombre de notes supérieures à 12 contenues dans ce tableau

Solution

```
import numpy as np
```

(suite sur la page suivante)

(suite de la page précédente)

```
notes = np.array(
  [[10, 12],
   [15, 16],
   [18, 12]]
)

moyennes = notes.mean(axis=0)
n_notes_sup_12 = np.sum(notes > 12)
```

CHAPTER 10

LA PROGRAMMATION ORIENTÉE OBJET

Dans ce chapitre, nous allons parler de programmation orientée objet. Pour cela, nous allons tout d'abord donner une description de ce qu'est un objet et revenir sur des objets que vous avez déjà manipulé en Python. Nous verrons ensuite comment définir vos propres objets et les utiliser.

10.1 Les objets du quotidien

En Python, toutes les variables que vous manipulez sont en fait des objets. Dans la suite de ce chapitre, nous allons prendre l'exemple d'un type que vous utilisez souvent, le type *chaîne de caractères* (`str`). En termes de vocabulaire, on dit que `"abc"` est un **objet** de la **classe** `str`.

Nous avons vu dans le chapitre dédié que l'on disposait, pour les chaînes de caractères, de fonctions permettant des manipulations élémentaires, comme par exemple passer la chaîne de caractères en minuscule :

```
s = "abcDEf"
print(s.upper())
```

```
ABCDEF
```

Vous vous êtes peut-être déjà habitué à cette syntaxe, pourtant il s'agit bien d'une syntaxe spécifique aux objets. En fait, ici, vous demandez d'appeler la **méthode** `upper()` de l'objet `s`. Une méthode est une fonction rattachée à un objet.

En plus des méthodes, les objets peuvent avoir des **attributs**, qui les décrivent. Jetons un oeil à un type un peu particulier, le type *nombre complexe* :

```
nombre_complexe = 10 + 5j
```

Notez qu'ici `j` permet d'identifier la partie imaginaire. Les objets de ce type ont, en Python, un attribut qui stocke leur partie entière et un autre pour leur partie imaginaire :

```
print(nombre_complexe.real)
print(nombre_complexe.imag)
```

```
10.0
5.0
```

On dit que `real` et `imag` sont des **attributs** (on parle aussi de propriétés) de l'objet nombre complexe.

En résumé, nos objets ont des méthodes (qui sont des fonctions) et des attributs, et pour y accéder, on utilise la notation `objet.methode()` ou `objet.attribut`. Il est à noter qu'une méthode peut avoir des arguments, comme toute fonction, comme dans l'exemple suivant :

```
print(s.find("b"))
```

```
1
```

10.2 Définir vos propres objets

La librairie Python standard propose déjà un nombre important de classes (c'est-à-dire de types) pré-définies. Pour définir une nouvelle classe, on utilise la syntaxe suivante :

```
class Vecteur:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def translation(self, delta_x, delta_y):
        self.x += delta_x
        self.y += delta_y
```

On a défini ici une nouvelle classe : la classe `Vecteur`. Les objets de cette classe ont deux attribut : `x` et `y` et une méthode `translation(self, delta_x, delta_y)`.

Voyons comment créer un nouvel objet de cette classe :

```
mon_vecteur = Vecteur(0., 2.)
```

Lors de la définition d'un nouvel objet, la méthode `__init__()` est appelée pour « construire » ce nouvel objet, et lui attribuer les bonnes propriétés. On peut accéder aux attributs de `mon_vecteur` pour s'en convaincre :

```
print(mon_vecteur.x, mon_vecteur.y)
```

```
0.0 2.0
```

De même, on peut utiliser ses méthodes :

```
mon_vecteur.translation(delta_x=1., delta_y=0.)
print(mon_vecteur.x, mon_vecteur.y)
```

```
1.0 2.0
```

Le mot-clé self

Dans tous les cas que vous rencontrerez dans ce cours, le premier argument d'une méthode sera `self`. Cet argument dénote l'objet sur lequel on est en train de travailler. Ainsi, lorsqu'on écrit :

```
    def translation(self, delta_x, delta_y):
        self.x += delta_x
        self.y += delta_y
```

le sens de ce code est le suivant :

- on définit une méthode `translation` qui aura **deux** arguments (il ne faut pas compter `self` qui est un argument spécial)
- lorsque l'on appelle cette méthode avec une syntaxe du type `mon_vecteur.translation(delta_x=1., delta_y=0.)`, cette méthode a pour effet de modifier la valeur des attributs `x` et `y` de l'objet `mon_vecteur` (celui sur lequel la méthode est appelée)

10.2.1 Les méthodes spéciales

Il existe des opérations « spéciales » que l'on peut vouloir effectuer sur des objets. On peut par exemple vouloir les afficher via `print()`, ou les sommer. Pour cela, on fait appel à des **méthodes spéciales**.

Définissons par exemple un nouveau vecteur en spécifiant ses coordonnées dans le plan, et affichons-le :

```
v0 = Vecteur(1.5, -1.)
print(v0)
```

```
<__main__.Vecteur object at 0x7fa94423d5d0>
```

Ici, on a bien défini notre nouveau vecteur, mais par contre l'affichage laisse à désirer (en tout cas, il ne nous permet pas de savoir ce que contient notre vecteur). On va donc ajouter une nouvelle méthode dont le nom nous est imposé : la méthode `__repr__()` qui permet de définir la **représentation** sous forme de chaîne de caractères d'un objet :

```
class Vecteur:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Vecteur({self.x}, {self.y})"


v0 = Vecteur(1.5, -1.)
print(v0)
```

```
Vecteur(1.5, -1.0)
```

On voit bien ici que, même si on n'appelle pas explicitement la méthode `__repr__`, elle est appelée dès lors que l'on doit obtenir une représentation d'un objet sous la forme d'une chaîne de caractères.

De la même façon, on peut vouloir définir le résultat de l'opération `v0 + v1` où `v0` et `v1` sont des vecteurs. On devra pour cela définir une méthode `__add__` :

```
class Vecteur:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Vecteur({self.x}, {self.y})"

    def __add__(self, autre_vecteur):
        nouveau_vecteur = Vecteur(x=self.x + autre_vecteur.x,
                                  y=self.y + autre_vecteur.y)
        return nouveau_vecteur


v0 = Vecteur(1.5, -1.)
v1 = Vecteur(1., 0.)
v_somme = v0 + v1
print(v_somme)
```

```
Vecteur(2.5, -1.0)
```

Ici, lorsque l'on écrit `v0 + v1`, tout se passe comme si cette expression était remplacée par `v0.__add__(v1)`.

De nombreuses méthodes spéciales peuvent ainsi être définies :

Méthode spéciale	Opérateur
`__add__(self, o)`	`+`
`__sub__(self, o)`	`-`
`__mul__(self, o)`	`*`
`__truediv__(self, o)`	`/`
`__pow__(self, o)`	`**`

10.2.2 Les attributs calculés

Dans certains cas, on aimerait rajouter à nos objets des attributs qui pourraient être calculés à la volée. Si l'on reprend l'exemple de la classe `Vecteur` plus haut, on peut d'ores et déjà accéder à ses attributs `x` et `y` :

```
v1 = Vecteur(1., 0.)
print(v1.x, v1.y)
```

```
1.0 0.0
```

On pourrait vouloir accéder à la norme de ce vecteur *via* un attribut qui serait calculé à la volée, en fonction des valeurs des attributs `x` et `y`, et cela est possible en Python. Pour cela, il faut définir une méthode `norme` qui ne prenne que `self` comme argument et la « décorer » avec le décorateur `@property` (on rappelle que les attributs peuvent également être qualifiés de propriétés) :

```
from math import sqrt

class Vecteur:
    def __init__(self, x, y):
        self.x = x
```

(suite sur la page suivante)

(suite de la page précédente)

```
        self.y = y

    @property
    def norme(self):
        return sqrt(self.x ** 2 + self.y ** 2)

    def __repr__(self):
        return f"Vecteur({self.x}, {self.y})"

    def __add__(self, autre_vecteur):
        nouveau_vecteur = Vecteur(x=self.x + autre_vecteur.x,
                                  y=self.y + autre_vecteur.y)
        return nouveau_vecteur


v1 = Vecteur(1., 0.)
print(v1.norme)
```

```
1.0
```

10.3 La notion d'héritage

Dès lors que l'on va introduire plusieurs nouvelles classes dans nos programmes, il arrivera que nos classes partagent un certain nombre d'attributs, voire de méthodes.

Prenons pour cela un nouvel exemple. Imaginons que l'on souhaite représenter des véhicules, qui pourront être des vélos ou bien des voitures. Dans ce cas, on va définir une classe **mère**, nommée `Polygone`, et deux classes filles `Rectangle` et `Triangle` comme suit :

```
class Polygone:
    def __init__(self, cotes):
        self.cotes = cotes

    def perimetre(self):
        return sum(self.cotes)


class Triangle(Polygone):
    def __init__(self, cotes):
        super().__init__(cotes)


class Rectangle(Polygone):
    def __init__(self, largeur, longueur):
        super().__init__(cotes=[largeur, longueur, largeur, longueur])

    def perimetre(self):
        return 2 * sum(self.cotes[:2])
```

Dans le code ci-dessus, on décide que lors de la construction d'un nouveau polygone, la liste des longueurs de ses côtés sera initialisée vide. On fait aussi le choix de dire que les classes `Triangle` et `Rectangle` **héritent** de la classe `Polygone` (on le spécifie en écrivant `class Triangle(Polygone)`). En héritant de cette classe, elles récupèrent tout ce qui existait pour cette classe (l'initialisation par défaut et la méthode pour calculer le périmètre). La classe `Rectangle`

redéfinit en outre la méthode périmètre car dans le cas des rectangles, on n'a pas besoin d'accéder aux longueurs des 4 côtés pour le calcul.

Enfin, les instructions `super().__init__()` sont à comprendre comme « exécuter la méthode `__init__` de la classe parente », le mot-clé `super()` étant l'équivalent de `self` pour la classe parente.

Pour créer un nouvel objet `Triangle` ou `Rectangle`, on peut alors faire :

```
t = Triangle([2, 2, 3])
r = Rectangle(2, 4)

print(t.cotes)
print(r.cotes)
```

```
[2, 2, 3]
[2, 4, 2, 4]
```

De plus, même si on ne le voit pas directement dans le code, grâce à l'héritage, la méthode `perimetre` existe pour les objets de ces classes :

```
print(t.perimetre())
print(r.perimetre())
```

```
7
12
```

Regardons de plus près ce qui se passe lorsqu'on crée un nouvel objet `Rectangle`.

```
class Rectangle(Polygone):
    def __init__(self, largeur, longueur):
        super().__init__([largeur, longueur, largeur, longueur])
```

Lors de la création d'un nouvel objet, comme pour n'importe quelle classe, la méthode `__init__` est appelée. À l'intérieur de cette méthode, deux choses sont faites :

1. `super().__init__()` signifie que l'on appelle le constructeur de la classe mère, soit `Polygone` : cela permet de définir un attribut `cotes` dont la valeur sera la liste `[largeur, longueur, largeur, longueur]` ici.

De plus, vous remarquez que la classe `Rectangle` redéfinit la méthode `perimetre`. On dit que la classe `Rectangle` **surcharge** la méthode `perimetre`.

Dans ce cas, au lieu de réutiliser la méthode `perimetre` de `Polygone`, si l'on appelle la méthode `perimetre` pour un objet de la classe `Rectangle`, c'est cette nouvelle version qui sera utilisée :

```
print(r.perimetre())
```

```
12
```

10.3.1 L'héritage multiple

Dans certains cas, on souhaitera qu'une classe hérite de deux (ou plus) classes parentes à la fois. Ce mécanisme s'appelle l'héritage multiple et il est tout à fait possible de le mettre en oeuvre en Python :

```
class Rectangle:
    pass

class Losange:
    pass

class Carre(Rectangle, Losange):
    pass
```

Dans le code ci-dessus, on définit une classe `Carre` qui hérite des classes `Rectangle` et `Losange`.

Dans le cas de l'héritage multiple, le rôle de `super()` est ambigu puisqu'on a plusieurs classes parentes. Développons un petit peu le code du dessus pour mieux voir comment tout cela se déroule :

```
class Rectangle:
    def __init__(self):
        print("Init Rectangle")
        super().__init__()

class Losange:
    def __init__(self):
        print("Init Losange")
        super().__init__()

class Carre(Rectangle, Losange):
    def __init__(self):
        print("Init Carre")
        super().__init__()

c = Carre()
```

```
Init Carre
Init Rectangle
Init Losange
```

L'affichage nous démontre que les constructeurs des deux classes parentes ont bien été appelés, dans l'ordre dans lequel elles ont été déclarées par `class Carre(Rectangle, Losange)` (donc le constructeur de `Rectangle` est appelé avant celui de `Losange`).

Il est important de noter ici que, pour que cet ordre d'appel soit effectif, il faut que chacune des classes concernées fasse appel au constructeur de sa classe parente (`super().__init__()`) dans son propre constructeur.

10.3.2 Classes abstraites

Les classes abstraites sont des classes un peu spéciales au sens elles ne sont pas faites pour être instanciées. Ces classes servent en conséquence à définir un modèle dont d'autres classes hériteront, et ce sont ces classes filles qui pourront être instanciées.

En Python, pour définir une classe abstraite, il suffit de la faire hériter de la classe `ABC` (*Abstract Base Class*) du module `abc` :

```
from abc import ABC

class FormeGeometrique(ABC):
    def __init__(self):
        super().__init__()


class Polygone(FormeGeometrique):
    def __init__(self, cotes):
        super().__init__()
        self.cotes = cotes

    def perimetre(self):
        return sum(self.cotes)

p = Polygone(cotes=[1, 1, 1, 1, 1])
print(p.perimetre())
```

```
5
```

Dans certains cas, on voudra spécifier dans la classe mère abstraite des méthodes (ou attributs calculés) à implémenter dans la ou les classes filles. Cela peut se faire en définissant ces méthodes dans la classe mère et en les décorant avec les décorateurs `@abstractmethod` (ou `@abstractproperty`, selon le cas) comme dans l'exemple suivant :

```
from abc import abstractmethod

class FormeGeometrique(ABC):
    def __init__(self):
        super().__init__()

    @abstractmethod
    def perimetre(self):
        # Le code ci-dessous importe peu puisque
        # la méthode devra être redéfinie dans les
        # classes filles
        pass


class Polygone(FormeGeometrique):
    def __init__(self, cotes):
        self.cotes = cotes

    def perimetre(self):
        return sum(self.cotes)

p = Polygone([1, 4, 4, 5, 5])
```

Si, par contre, on n'implémente pas la méthode abstraite dans l'une des classes filles, cette classe ne pourra pas être

instanciée :

```
class Cercle(FormeGeometrique):
    def __init__(self, rayon):
        super().__init__()
        self.rayon = rayon

mon_cercle = Cercle()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_5599/569023471.py in <module>
      4         self.rayon = rayon
      5
----> 6 mon_cercle = Cercle()

TypeError: Can't instantiate abstract class Cercle with abstract methods perimetre
```

CHAPTER 11

TESTER SON CODE

Pour info

Il existe en Python des outils dédiés au test de programmes. Toutefois, ce chapitre ne traite pas de l'utilisation de ces outils, mais plutôt de l'intérêt des tests en général.

Dans ce document, nous avons jusqu'à présent supposé que tout se passait bien, que votre code ne retournait jamais d'erreur et qu'il ne contenait jamais de *bug*. Quel que soit votre niveau d'expertise en Python, ces deux hypothèses sont peu réalistes. Nous allons donc nous intéresser maintenant aux moyens de vérifier si votre code fait bien ce qu'on attend de lui et de mieux comprendre son comportement lorsque ce n'est pas le cas.

11.1 Les erreurs en Python

Étudions ce qu'il se passe lors de l'exécution du code suivant :

```
x = "12"
y = x + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_5657/3348748796.py in <module>
      1 x = "12"
----> 2 y = x + 2

TypeError: can only concatenate str (not "int") to str
```

Ce type de message d'erreur ne doit pas vous effrayer, il est là pour vous aider. Il vous fournit de précieuses informations :

1. l'erreur se produit à la ligne 2 de votre script Python ;

2. le problème est que Python ne peut pas convertir un objet de type `int` en chaîne de caractères (`str`) de manière implicite.

Reste à se demander pourquoi, dans le cas présent, Python voudrait transformer un entier en chaîne de caractères. Pour le comprendre, rendons-nous à la ligne 2 de notre script et décortiquons-la. Dans cette ligne (`y = x + 2`), deux opérations sont effectuées :

- la première consiste à effectuer l'opération + entre les opérandes `x` et `2` ;
- la seconde consiste à assigner le résultat de l'opération à la variable `y`.

Nous avons vu dans ce document que Python savait effectuer l'opération + avec des opérandes de types variés (nombre + nombre, liste + liste, chaîne de caractères + chaîne de caractères, et il en existe d'autres). Intéressons-nous ici au type des opérandes considérées. La variable `x` telle que définie à la ligne 1 est de type chaîne de caractères. La valeur `2` est de type entier. Il se trouve que Python n'a pas défini d'addition entre chaîne de caractères et entier et c'est pour cela que l'on obtient une erreur. Plus précisément, l'interpréteur Python nous dit : « si je pouvais convertir la valeur entière en chaîne de caractères à la volée, je pourrais faire l'opération + qui serait alors une concaténation, mais je ne me permets pas de le faire tant que vous ne l'avez pas écrit de manière explicite ».

Maintenant que nous avons compris le sens de ce *bug*, il nous reste à le corriger. Si nous souhaitons faire la somme du nombre 12 (stocké sous forme de chaîne de caractères dans la variable `x`) et de la valeur 2, nous écrivons :

```
x = "12"
y = int(x) + 2
```

et l'addition s'effectue alors correctement entre deux valeurs numériques.

11.2 Les tests unitaires

Pour pouvoir être sûr du code que vous écrivez, il faut l'avoir testé sur un ensemble d'exemples qui vous semble refléter l'ensemble des cas de figures auxquels votre programme pourra être confronté. Or, cela représente un nombre de cas de figures très important dès lors que l'on commence à écrire des programmes un tant soit peu complexes. Ainsi, il est hautement recommandé de découper son code en fonctions de tailles raisonnables et qui puissent être testées indépendamment. Les tests associés à chacune de ces fonctions sont appelés **tests unitaires**.

Tout d'abord, en mettant en place de tels tests, vous pourrez détecter rapidement un éventuel *bug* dans votre code et ainsi gagner beaucoup de temps de développement. De plus, vous pourrez également vous assurer que les modifications ultérieures de votre code ne modifient pas son comportement pour les cas testés. En effet, lorsque l'on ajoute une fonctionnalité à un programme informatique, il faut avant toute choses s'assurer que celle-ci ne cassera pas le bon fonctionnement du programme dans les cas classiques d'utilisation pour lesquels il avait été à l'origine conçu.

Prenons maintenant un exemple concret. Supposons que l'on souhaite écrire une fonction `bissextile` capable de dire si une année est bissextile ou non. En se renseignant sur le sujet, on apprend qu'une année est bissextile si :

- si elle est divisible par 4 et non divisible par 100, ou
- si elle est divisible par 400.

On en déduit un ensemble de tests adaptés :

```
print(bissextile(2004))  # True car divisible par 4 et non par 100
print(bissextile(1900))  # False car divisible par 100 et non par 400
print(bissextile(2000))  # True car divisible par 400
print(bissextile(1999))  # False car divisible ni par 4 ni par 100
```

On peut alors vérifier que le comportement de notre fonction `bissextile` est bien conforme à ce qui est attendu.

11.3 Le développement piloté par les tests

Le développement piloté par les tests (ou *Test-Driven Development*) est une technique de programmation qui consiste à rédiger les tests unitaires de votre programme avant même de rédiger le programme lui-même.

L'intérêt de cette façon de faire est qu'elle vous obligera à réfléchir aux différents cas d'utilisation d'une fonction avant de commencer à la coder. De plus, une fois ces différents cas identifiés, il est probable que la structure globale de la fonction à coder vous apparaisse plus clairement.

Si l'on reprend l'exemple de la fonction `bissextile` citée plus haut, on voit assez clairement qu'une fois que l'on a rédigé l'ensemble de tests, la fonction sera simple à coder et reprendra les différents cas considérés pour les tests:

```
def bissextile(annee):
    if annee % 4 == 0 and annee % 100 != 0:
        return True
    elif annee % 400 == 0:
        return True
    else:
        return False
```

11.4 Exercices

Exercice 11.1

En utilisant les méthodes de développement préconisées dans ce chapitre, rédigez le code et les tests d'un programme permettant de déterminer le lendemain d'une date fournie sous la forme de trois entiers (jour, mois, année).

Solution

```
def bissextile(annee):
    if annee % 4 == 0 and annee % 100 != 0:
        return True
    elif annee % 400 == 0:
        return True
    else:
        return False

def nb_jours(mois, annee):
    if mois in [1, 3, 5, 7, 8, 10, 12]:
        return 31
    elif mois in [4, 6, 9, 11]:
        return 30
    else:  # Mois de février
        if bissextile(annee):
            return 29
        else:
            return 28

def lendemain(jour, mois, annee):
    if jour < nb_jours(mois, annee):
        return jour + 1, mois, annee
```

(suite sur la page suivante)

(suite de la page précédente)

```
    elif mois < 12:  # Dernier jour du mois mais pas de l'année
        return 1, mois + 1, annee
    else:  # Dernier jour de l'année
        return 1, 1, annee + 1

# Tests de la fonction bissextile
print(bissextile(2004))  # True car divisible par 4 et non par 100
print(bissextile(1900))  # False car divisible par 100 et non par 400
print(bissextile(2000))  # True car divisible par 400
print(bissextile(1999))  # False car divisible ni par 4 ni par 100

# Tests de la fonction nb_jours
print(nb_jours(3, 2010))  # Mars : 31
print(nb_jours(4, 2010))  # Avril : 30
print(nb_jours(2, 2010))  # Février d'une année non bissextile : 28
print(nb_jours(2, 2004))  # Février d'une année bissextile : 29

# Tests de la fonction lendemain
print(lendemain(12, 2, 2010))  # 13, 2, 2010
print(lendemain(28, 2, 2010))  #  1, 3, 2010
print(lendemain(31, 12, 2010)) #  1, 1, 2011
```

Exercice 11.2

Proposez une ré-écriture de la fonction bissextile ci-dessus qui tienne en une ligne de la forme :

```
def bissextile(annee):
    return CONDITION_COMPLEXE
```

où `CONDITION_COMPLEXE` est un booléen calculé à partir de la valeur de `annee`. Assurez-vous que cette nouvelle fonction passe bien les tests énoncés ci-dessus.

Solution

```
def bissextile(annee):
    return (annee % 4 == 0 and annee % 100 != 0) or (annee % 400 == 0)

print(bissextile(2004))  # True car divisible par 4 et non par 100
print(bissextile(1900))  # False car divisible par 100 et non par 400
print(bissextile(2000))  # True car divisible par 400
print(bissextile(1999))  # False car divisible ni par 4 ni par 100
```

CHAPTER 12

CONCLUSION

Dans ce document, nous avons abordé les principes de base de la programmation en Python, tels qu'enseignés à des étudiants non informaticiens en licence à l'Université de Rennes 2.

Comme indiqué en introduction, ce document se veut évolutif. N'hésitez donc pas à faire vos remarques à son auteur dont vous trouverez le contact sur sa page web.