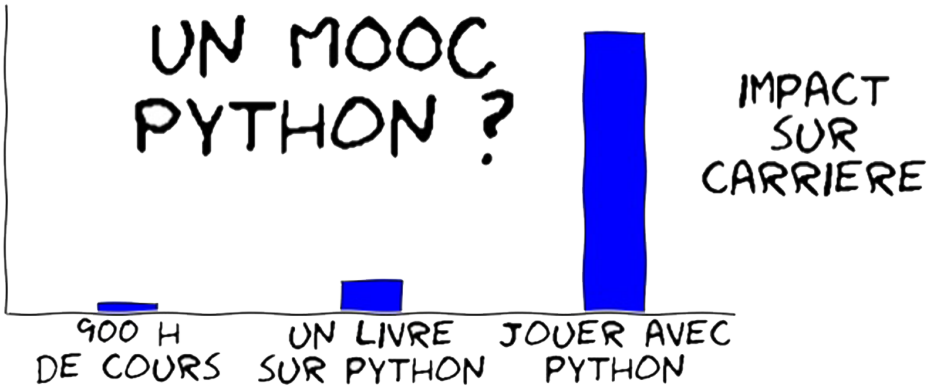
python™ 3

DES FONDAMENTAUX AU CONCEPTS AVANCÉS DU LANGAGE

SESSION 2 - 17 SEPTEMBRE 2018

Thierry PARMENTELAT

Arnaud LEGOUT



https ://www.fun-mooc.fr

Table des matières

1 Introduction au MOOC et aux outils Python **5**

2 Notions de base, premier programme en Python **7**

3 Renforcement des notions de base, références partagées **9**

3.1 Les fichiers . . . 10
3.1.1 Complément - niveau basique . . . 10
3.1.2 Complément - niveau intermédiaire . . . 11
3.1.3 Complément - niveau avancé . . . 11
3.2 Fichiers et utilitaires . . . 15
3.2.1 Complément - niveau basique . . . 15
3.2.2 Complément - niveau avancé . . . 18
3.3 Formats de fichiers : JSON et autres . . . 19
3.3.1 Compléments - niveau basique . . . 19
3.3.2 Compléments - niveau intermédiaire . . . 20
3.4 Fichiers systèmes . . . 22
3.4.1 Complément - niveau avancé . . . 22
3.5 La construction de tuples . . . 24
3.5.1 Complément - niveau intermédiaire . . . 24
3.6 Sequence unpacking . . . 28
3.6.1 Complément - niveau basique . . . 28
3.6.2 Complément - niveau intermédiaire . . . 30
3.6.3 Pour en savoir plus . . . 32
3.7 Plusieurs variables dans une boucle `for` . . . 33
3.7.1 Complément - niveau basique . . . 33
3.7.2 Complément - niveau intermédiaire . . . 33
3.8 Fichiers . . . 36
3.8.1 Exercice - niveau basique . . . 36
3.9 *Sequence unpacking* . . . 38
3.9.1 Exercice - niveau basique . . . 38
3.10 Dictionnaires . . . 39
3.10.1 Complément - niveau basique . . . 39
3.10.2 Complément - niveau intermédiaire . . . 42
3.10.3 Complément - niveau avancé . . . 45
3.11 Clés immuables . . . 48
3.11.1 Complément - niveau intermédiaire . . . 48
3.12 Gérer des enregistrements . . . 50
3.12.1 Complément - niveau intermédiaire . . . 50

3.12.2 Complément - niveau avancé . . . 52
3.13 Dictionnaires et listes . . . 53
3.13.1 Exercice - niveau basique . . . 53
3.14 Fusionner des données . . . 54
3.14.1 Exercices . . . 54
3.15 Ensembles . . . 58
3.15.1 Complément - niveau basique . . . 58
3.16 Ensembles . . . 63
3.16.1 Exercice - niveau basique . . . 63
3.16.2 Deuxième partie - niveau basique . . . 63
3.17 Exercice sur les ensembles . . . 64
3.17.1 Exercice - niveau intermédiaire . . . 64
3.18 `try ... else ... finally` . . . 66
3.18.1 Complément - niveau intermédiaire . . . 66
3.19 L'opérateur `is` . . . 69
3.19.1 Complément - niveau basique . . . 69
3.19.2 Complément - niveau intermédiaire . . . 70
3.20 Listes infinies & références circulaires . . . 73
3.20.1 Complément - niveau intermédiaire . . . 73
3.21 Les différentes copies . . . 76
3.21.1 Complément - niveau basique . . . 76
3.21.2 Complément - niveau intermédiaire . . . 77
3.22 L'instruction `del` . . . 80
3.22.1 Complément - niveau basique . . . 80
3.23 Affectation simultanée . . . 82
3.23.1 Complément - niveau basique . . . 82
3.24 Les instructions `+=` et autres revisitées . . . 83
3.24.1 Complément - niveau intermédiaire . . . 83
3.25 Classe . . . 86
3.25.1 Exercice - niveau basique . . . 86

Chapitre 1

Introduction au MOOC et aux outils Python

Chapitre 2

Notions de base, premier programme en Python

Chapitre 3

Renforcement des notions de base, références partagées

3.1 Les fichiers

3.1.1 Complément - niveau basique

Voici quelques utilisations habituelles du type fichier en Python.

Avec un *context manager*

Nous avons vu dans la vidéo les mécanismes de base sur les fichiers. Nous avons vu notamment qu'il est important de bien fermer un fichier après usage. On a vu aussi qu'il est recommandé de **toujours** utiliser l'instruction `with` et de contrôler son encodage. Il est donc recommandé de faire :

```
In [1]: # avec un `with' on garantit la fermeture du fichier
        with open("foo.txt", "w", encoding='utf-8') as sortie:
            for i in range(2):
                sortie.write(f"{i}\n")
```

Les modes d'ouverture

Les modes d'ouverture les plus utilisés sont :

— `'r'` (la chaîne contenant l'unique caractère `r`) pour ouvrir un fichier en lecture seulement ;

— `'w'` en écriture seulement ; le contenu précédent du fichier, s'il existait, est perdu ;

— `'a'` en écriture seulement ; mais pour ajouter du contenu en fin de fichier.

Voici par exemple comment on pourrait ajouter deux lignes de texte dans le fichier `foo.txt` qui contient, à ce stade du notebook, deux entiers :

```
In [2]: # on ouvre le fichier en mode 'a' comme append (= ajouter)
        with open("foo.txt", "a", encoding='utf-8') as sortie:
            for i in range(100, 102):
                sortie.write(f"{i}\n")

In [3]: # maintenant on regarde ce que contient le fichier
        with open("foo.txt", encoding='utf-8') as entree: # remarquez que sans 'mode', on ou
            for line in entree:
                # line contient déjà un retour à la ligne
                print(line, end='')
```

```
0
1
100
101
```

Il existe de nombreuses variantes au mode d'ouverture, pour par exemple :

— ouvrir le fichier en lecture *et* en écriture (mode `+`) ;

— ouvrir le fichier en mode binaire (mode `b`).

Ces variantes sont décrites dans la section sur la fonction built-in `open` dans la documentation Python.

3.1.2 Complément - niveau intermédiaire

Un fichier est un itérateur

Nous reparlerons des notions d'itérable et d'itérateur dans les semaines suivantes. Pour l'instant, on peut dire qu'un fichier - qui donc **est itérable** puisqu'on peut le lire par une boucle `for` - est aussi **son propre itérateur**. Cela implique que l'on ne peut le parcourir qu'une fois dans une boucle `for`. Pour le reparcourir, il faut le fermer et l'ouvrir de nouveau.

```
In [4]: # un fichier est son propre itérateur

In [5]: with open("foo.txt", encoding='utf-8') as entree:
            print(entree.__iter__() is entree)
```

```
True
```

Par conséquent, écrire deux boucles `for` imbriquées sur **le même objet fichier** ne **fonctionnerait pas** comme on pourrait s'y attendre.

```
In [6]: # Si l'on essaie d'écrire deux boucles imbriquées
        # sur le même objet fichier, le résultat est inattendu
        with open("foo.txt", encoding='utf-8') as entree:
            for l1 in entree:
                # on enlève les fins de ligne
                l1 = l1.strip()
                for l2 in entree:
                    # on enlève les fins de ligne
                    l2 = l2.strip()
                    print(l1, "x", l2)
```

```
0 x 1
0 x 100
0 x 101
```

3.1.3 Complément - niveau avancé

Autres méthodes

Vous pouvez également accéder à des fonctions de beaucoup plus bas niveau, notamment celle fournies directement par le système d'exploitation ; nous allons en décrire deux parmi les plus utiles.

Digression - `repr()`

Comme nous allons utiliser maintenant des outils d'assez bas niveau pour lire du texte, pour examiner ce texte nous allons utiliser la fonction `repr()`, et voici pourquoi :

```
In [7]: # construisons à la main une chaîne qui contient deux lignes
        lines = "abc" + "\n" + "def"  + "\n"

In [8]: # si on l'imprime on voit bien les retours à la ligne
        # d'ailleurs on sait qu'il n'est pas utile
        # d'ajouter un retour à la ligne à la fin
        print(lines, end="")
```

```
abc
def
```

```
In [9]: # vérifions que repr() nous permet de bien
        # voir le contenu de cette chaine
        print(repr(lines))
```

```
'abc\ndef\n'
```

Lire un contenu - bas niveau

Revenons aux fichiers; la méthode `read()` permet de lire dans le fichier un buffer d'une certaine taille :

```
In [10]: # read() retourne TOUT le contenu
         # ne pas utiliser avec de très gros fichiers bien sûr

         # une autre façon de montrer tout le contenu du fichier
         with open("foo.txt", encoding='utf-8') as entree:
             full_contents = entree.read()
             print(f"Contenu complet\n{full_contents}", end="")
```

```
Contenu complet
0
1
100
101
```

```
In [11]: # lire dans le fichier deux blocs de quatre caractères
         with open("foo.txt", encoding='utf-8') as entree:
             for bloc in range(2):
                 print(f"Bloc {bloc} >>{repr(entree.read(4))}<<")
```

```
Bloc 0 >>'0\n1\n'<<
Bloc 1 >>'100\n'<<
```

On voit donc que chaque bloc contient bien quatre caractères en comptant les sauts de ligne :

bloc #	contenu
0	un 0, un *newline*, un 1, un *newline*
1	un 1, deux 0, un *newline*

La méthode `flush`

Les entrées-sorties sur fichier sont bien souvent *bufferisées* par le système d'exploitation. Cela signifie qu'un appel à `write` ne provoque pas forcément une écriture immédiate, car pour des

raisons de performance on attend d'avoir suffisamment de matière avant d'écrire sur le disque.

Il y a des cas où ce comportement peut s'avérer gênant, et où on a besoin d'écrire immédiatement (et donc de vider le *buffer*), et c'est le propos de la méthode `flush`.

Fichiers textuels et fichiers binaires

De la même façon que le langage propose les deux types `str` et `bytes`, il est possible d'ouvrir un fichier en mode *textuel* ou en mode *binaire*.

Les fichiers que nous avons vus jusqu'ici étaient ouverts en mode *textuel* (c'est le défaut), et c'est pourquoi nous avons interagi avec eux avec des objets de type `str` :

```
In [12]: # un fichier ouvert en mode textuel nous donne des str
         with open('foo.txt', encoding='utf-8') as input:
             for line in input:
                 print("on a lu un objet de type", type(line))
```

```
on a lu un objet de type <class 'str'>
on a lu un objet de type <class 'str'>
on a lu un objet de type <class 'str'>
on a lu un objet de type <class 'str'>
```

Lorsque ce n'est pas le comportement souhaité, on peut :

— ouvrir le fichier en mode *binaire* - pour cela on ajoute le caractère `b` au mode d'ouverture ;

— et on peut alors interagir avec le fichier avec des objets de type `bytes`

Pour illustrer ce trait, nous allons : 0. créer un fichier en mode texte, et y insérer du texte en UTF-8 ; 0. relire le fichier en mode binaire, et retrouver le codage des différents caractères.

```
In [13]: # phase 1 : on écrit un fichier avec du texte en UTF-8
         # on ouvre donc le fichier en mode texte
         # en toute rigueur il faut préciser l'encodage,
         # si on ne le fait pas il sera déterminé
         # à partir de vos réglages système
         with open('strbytes', 'w', encoding='utf-8') as output:
             output.write("déjà l'été\n")
```

```
In [14]: # phase 2: on ouvre le fichier en mode binaire
         with open('strbytes', 'rb') as rawinput:
             # on lit tout le contenu
             octets = rawinput.read()
             # qui est de type bytes
             print("on a lu un objet de type", type(octets))
             # si on regarde chaque octet un par un
             for i, octet in enumerate(octets):
                 print(f"{i} → {repr(chr(octet))} [{hex(octet)}]")
```

```
on a lu un objet de type <class 'bytes'>
0 → 'd' [0x64]
1 → 'Ã' [0xc3]
```

```
2 → '©' [0xa9]
3 → 'j' [0x6a]
4 → 'Ã' [0xc3]
5 → '\xa0' [0xa0]
6 → ' ' [0x20]
7 → 'l' [0x6c]
8 → "'" [0x27]
9 → 'Ã' [0xc3]
10 → '©' [0xa9]
11 → 't' [0x74]
12 → 'Ã' [0xc3]
13 → '©' [0xa9]
14 → '\r' [0xd]
15 → '\n' [0xa]
```

Vous retrouvez ainsi le fait que l'unique caractère Unicode é a été encodé par UTF-8 sous la forme de deux octets de code hexadécimal `0xc3` et `0xa9`.

Vous pouvez également consulter ce site qui visualise l'encodage UTF-8, avec notre séquence d'entrée :

https://mothereff.in/utf-8#d%C3%A9j%C3%A0%20l%27%C3%A9t%C3%A9%0A

```
In [15]: # on peut comparer le nombre d'octets et le nombre de caractères
         with open('strbytes', encoding='utf-8') as textfile:
             print(f"en mode texte, {len(textfile.read())} caractères")
         with open('strbytes', 'rb') as binfile:
             print(f"en mode binaire, {len(binfile.read())} octets")
```

```
en mode texte, 11 caractères
en mode binaire, 16 octets
```

Ce qui correspond au fait que nos quatre caractères non-ASCII (3 x é et 1 x à) sont tous encodés par UTF-8 comme deux octets, comme vous pouvez vous en assurer ici pour é et là pour à.

Pour en savoir plus

Pour une description exhaustive vous pouvez vous reporter :

— au glossaire sur la notion de `object file`,

— et aussi et surtout au module `io` qui décrit plus en détail les fonctionnalités disponibles.

3.2 Fichiers et utilitaires

3.2.1 Complément - niveau basique

Outre les objets fichiers créés avec la fonction `open`, comme on l'a vu dans la vidéo, et qui servent à lire et écrire à un endroit précis, une application a besoin d'un minimum d'utilitaires pour **parcourir l'arborescence de répertoires et fichiers**, c'est notre propos dans ce complément.

Le module `os.path` (obsolète)

Avant la version python-3.4, la librairie standard offrait une conjonction d'outils pour ce type de fonctionnalités :

— le module `os.path`, pour faire des calculs sur les les chemins et noms de fichiers doc,
— le module `os` pour certaines fonctions complémentaires comme renommer ou détruire un fichier doc,
— et enfin le module `glob` pour la recherche de fichiers, par exemple pour trouver tous les fichiers en `*.txt` doc.

Cet ensemble un peu disparate a été remplacé par une **librairie unique** `pathlib`, qui fournit toutes ces fonctionnalités sous un interface unique et moderne, que nous **recommandons** évidemment d'utiliser pour **du nouveau code**.

Avant d'aborder `pathlib`, voici un très bref aperçu de ces trois anciens modules, pour le cas - assez probable - où vous les rencontreriez dans du code existant ; tous les noms qui suivent correspondent à des **fonctions** - par opposition à `pathlib` qui, comme nous allons le voir, offre une interface orientée objet :

— `os.path.join` ajoute '/' ou '' entre deux morceaux de chemin, selon l'OS
— `os.path.basename` trouve le nom de fichier dans un chemin
— `os.path.dirname` trouve le nom du directory dans un chemin
— `os.path.abspath` calcule un chemin absolu, c'est-à-dire à partir de la racine du filesystem

— `os.path.exists` pour savoir si un chemin existe ou pas (fichier ou répertoire)
— `os.path.isfile` (et `isdir`) pour savoir si un chemin est un fichier (et un répertoire)
— `os.path.getsize` pour obtenir la taille du fichier
— `os.path.getatime` et aussi `getmtime` et `getctime` pour obtenir les dates de création/modification d'un fichier

— `os.remove` (ou son ancien nom `os.unlink`), qui permet de supprimer un fichier
— `os.rmdir` pour supprimer un répertoire (mais qui doit être vide)
— `os.removedirs` pour supprimer tout un répertoire avec son contenu, récursivement si nécessaire
— `os.rename` pour renommer un fichier

— `glob.glob` comme dans par exemple `glob.glob("*.txt")`

Le module `pathlib`

C'est la méthode recommandée aujourd'hui pour travailler sur les fichiers et répertoires.

Orienté Objet

Comme on l'a mentionné `pathlib` offre une interface orientée objet ; mais qu'est-ce

que ça veut dire au juste ?

Ceci nous donne un prétexte pour une première application pratique des notions de module (que nous avons introduits en fin de semaine 2) et de classe (que nous allons voir en fin de semaine).

De même que le langage nous propose les types *builtin* `int` et `str`, le module `pathlib` nous expose **un type** (on dira plutôt **une classe**) qui s'appelle `Path`, que nous allons importer comme ceci :

```
In [1]: from pathlib import Path
```

Nous allons faire tourner un petit scénario qui va créer un fichier :

```
In [2]: # le nom de notre fichier jouet
        nom = 'fichier-temoin'
```

Pour commencer, nous allons vérifier si le fichier en question existe.
Pour ça nous créons un **objet** qui est une **instance** de la classe `Path`, comme ceci :

```
In [3]: # on crée un objet de la classe Path, associé au nom de fichier
        path = Path(nom)
```

Vous remarquez que c'est consistent avec par exemple :

```
In [4]: # transformer un float en int
        i = int(3.5)
```

en ce sens que le type (`int` ou `Path`) se comporte comme une usine pour créer des objets du type en question.

Quoi qu'il en soit, cet objet `path` offre un certain nombre de méthodes ; pour les voir puisque nous sommes dans un notebook, je vous invite dans la cellule suivante à utiliser l'aide en ligne en appuyant sur la touche 'Tabulation' après avoir ajouté un `.` comme si vous alliez envoyer une méthode à cet objet

```
path.[taper la touche TAB]
```

et le notebook vous montrera la liste des méthodes disponibles.

```
In [5]: # ajouter un . et utilisez la touche <Tabulation>
        path.chmod
```

```
Out[5]: <bound method Path.chmod of WindowsPath('fichier-temoin')>
```

Ainsi par exemple on peut savoir si le fichier existe avec la méthode `exists()`

```
In [6]: # au départ le fichier n'existe pas
        path.exists()
```

```
Out[6]: False
```

```
In [7]: # si j'écris dedans je le crée
        with open(nom, 'w', encoding='utf-8') as output:
            output.write('0123456789\n')
```

```
In [8]: # et maintenant il existe
        path.exists()
```

```
Out[8]: True
```

Métadonnées

Voici quelques exemples qui montrent comment accéder aux métadonnées de ce fichier :

```
In [9]: # cette méthode retourne (en un seul appel système) les métadonnées agrégées
        path.stat()
```

```
Out[9]: os.stat_result(st_mode=33206, st_ino=2251799814156457, st_dev=571766744, st_nlink=1,
```

Pour ceux que ça intéresse, l'objet retourné par cette méthode `stat` est un `namedtuple`, que l'on va voir très bientôt.

On accède aux différentes informations comme ceci :

```
In [10]: # la taille du fichier en octets est de 11
         # car il faut compter un caractère "newline" en fin de ligne
         path.stat().st_size
```

```
Out[10]: 12
```

```
In [11]: # la date de dernière modification, sous forme d'un entier
         # c'est le nombre de secondes depuis le 1er Janvier 1970
         mtime = path.stat().st_mtime
         mtime
```

```
Out[11]: 1537703765.728249
```

```
In [12]: # que je peux rendre lisible comme ceci
         # en anticipant sur le module datetime
         from datetime import datetime
         mtime_datetime = datetime.fromtimestamp(mtime)
         mtime_datetime
```

```
Out[12]: datetime.datetime(2018, 9, 23, 13, 56, 5, 728249)
```

```
In [13]: # ou encore, si je formatte pour n'obtenir que
         # l'heure et la minute
         f"{mtime_datetime:%H:%M}"
```

```
Out[13]: '13:56'
```

Détruire un fichier

```
In [14]: # je peux maintenant détruire le fichier
         path.unlink()
```

```
In [15]: # ou encore mieux, si je veux détruire
         # seulement dans le cas où il existe je peux aussi faire
         try:
             path.unlink()
         except FileNotFoundError:
             print("no need to remove")
```

```
no need to remove
```

```
In [16]: # et maintenant il n'existe plus
         path.exists()
```

```
Out[16]: False
```

```
In [17]: # je peux aussi retrouver le nom du fichier comme ceci
         # attention ce n'est pas une méthode mais un attribut
         # c'est pourquoi il n'y a pas de parenthèses
         path.name
```

```
Out[17]: 'fichier-temoin'
```

Recherche de fichiers

Maintenant je voudrais connaître la liste des fichiers de nom `*.json` dans le directory `data`.

La méthode la plus naturelle consiste à créer une instance de `Path` associée au directory lui-même :

```
In [18]: dirpath = Path('./data/')
```

Sur cet objet la méthode `glob` nous retourne un itérable qui contient ce qu'on veut :

```
In [19]: # tous les fichiers *.json dans le répertoire data/
         for json in dirpath.glob("*.json"):
             print(json)
```

Documentation complète

Voyez la documentation complète ici

3.2.2 Complément - niveau avancé

Pour ceux qui sont déjà familiers avec les classes, j'en profite pour vous faire remarquer le type de notre objet path

```
In [20]: type(path)
```

```
Out[20]: pathlib.WindowsPath
```

qui n'est pas `Path`, mais en fait une sous-classe de `Path` qui est - sur la plateforme du MOOC au moins, qui fonctionne sous linux - un objet de type `PosixPath`, qui est une sous-classe de `Path`, comme vous pouvez le voir :

```
In [21]: from pathlib import PosixPath
         issubclass(PosixPath, Path)
```

```
Out[21]: True
```

Ce qui fait que mécaniquement, path est bien une instance de `Path`

```
In [22]: isinstance(path, Path)
```

```
Out[22]: True
```

ce qui est heureux puisqu'on avait utilisé `Path()` pour construire l'objet `path` au départ :)

3.3 Formats de fichiers : JSON et autres

3.3.1 Compléments - niveau basique

Voici quelques mots sur des outils Python fournis dans la bibliothèque standard, et qui permettent de lire ou écrire des données dans des fichiers.

Le problème

Les données dans un programme Python sont stockées en mémoire (la RAM), sous une forme propice aux calculs. Par exemple un petit entier est fréquemment stocké en binaire dans un mot de 64 bits, qui est prêt à être soumis au processeur pour faire une opération arithmétique.

Ce format ne se prête pas forcément toujours à être transposé tel quel lorsqu'on doit écrire des données sur un support plus pérenne, comme un disque dur, ou encore sur un réseau pour transmission distante - ces deux supports étant à ce point de vue très voisins.

Ainsi par exemple il pourra être plus commode d'écrire notre entier sur disque, ou de le transmettre à un programme distant, sous une forme décimale qui sera plus lisible, sachant que par ailleurs toutes les machines ne codent pas un entier de la même façon.

Il convient donc de faire de la traduction dans les deux sens entre représentations d'une part en mémoire, et d'autre part sur disque ou sur réseau (à nouveau, on utilise en général les mêmes formats pour ces deux usages).

Le format JSON

Le format sans aucun doute le plus populaire à l'heure actuelle est le format JSON pour *JavaScript Object Notation*.

Sans trop nous attarder nous dirons que JSON est un encodage - en anglais marshalling - qui se prête bien à la plupart des types de base que l'on trouve dans les langages modernes comme Python, Ruby ou JavaScript.

La bibliothèque standard de Python contient le module json que nous illustrons très rapidement ici :

```
In [1]: import json

        # En partant d'une donnée construite à partir de types de base
        data = [
            # des types qui ne posent pas de problème
            [1, 2, 'a', [3.23, 4.32], {'eric': 32, 'jean': 43}],
            # un tuple
            (1, 2, 3),
        ]

        # sauver ceci dans un fichier
        with open("s1.json","w", encoding='utf-8') as json_output:
            json.dump(data, json_output)
```

```
        # et relire le résultat
        with open("s1.json", encoding='utf-8') as json_input:
            data2 = json.load(json_input)
```

Limitations de json

Certains types de base ne sont pas supportés par le format JSON (car ils ne sont pas natifs en JavaScript), c'est le cas notamment pour :

— `tuple`, qui se fait encoder comme une liste ;

— `complex`, `set` et `frozenset`, que l'on ne peut pas encoder du tout (sans étendre la bibliothèque).

C'est ce qui explique ce qui suit :

```
In [2]: # le premier élément de data est intact,
        # comme si on avait fait une *deep copy* en fait
        print("première partie de data", data[0] == data2[0])
```

```
première partie de data True
```

```
In [3]: # par contre le `tuple` se fait encoder comme une `list`
        print("deuxième partie", "entrée", type(data[1]), "sortie", type(data2[1]))
```

```
deuxième partie entrée <class 'tuple'> sortie <class 'list'>
```

Malgré ces petites limitations, ce format est de plus en plus populaire, notamment parce qu'on peut l'utiliser pour communiquer avec des applications Web écrites en JavaScript, et aussi parce qu'il est très léger, et supporté par de nombreux langages.

3.3.2 Compléments - niveau intermédiaire

Le format `csv`

Le format `csv` pour *Comma Separated Values*, originaire du monde des tableurs, peut rendre service à l'occasion, il est proposé dans le module `csv`.

Le format pickle

Le format `pickle` remplit une fonctionnalité très voisine de `JSON`, mais est spécifique à Python. C'est pourquoi, malgré des limites un peu moins sévères, son usage tend à rester plutôt marginal pour l'échange de données, on lui préfère en général le format JSON.

Par contre, pour la sauvegarde locale d'objets Python (pour, par exemple, faire des points de reprises d'un programme), il est très utile. Il est implémenté dans le module `pickle`.

Le format XML

Vous avez aussi très probablement entendu parler de XML, qui est un format assez populaire également.

Cela dit, la puissance, et donc le coût, de XML et JSON ne sont pas du tout comparables, XML étant beaucoup plus flexible mais au prix d'une complexité de mise en œuvre très supérieure.

Il existe plusieurs souches différentes de bibliothèques prenant en charge le format XML, qui sont introduites ici.

Pour en savoir plus

Voyez la page sur les formats de fichiers dans la documentation Python.

3.4 Fichiers systèmes

3.4.1 Complément - niveau avancé

Dans ce complément, nous allons voir comment un programme Python interagit avec ce qu'il est convenu d'appeler le système d'entrées-sorties standard du système d'exploitation.

Introduction

Dans un ordinateur, le système d'exploitation (Windows, Linux, macOS, etc.) comprend un noyau (*kernel*) qui est un logiciel qui a l'exclusivité pour interagir physiquement avec le matériel (processeur(s), mémoire, disque(s), périphériques, etc.) ; il offre aux programmes utilisateur (*userspace*) des abstractions pour interagir avec ce matériel.

La notion de fichier, telle qu'on l'a vue dans la vidéo, correspond à une de ces abstractions ; elle repose principalement sur les quatre opérations élémentaires suivantes :

— `open` ;

— `close` ;

— `read` ;

— `write`.

Parmi les autres conventions d'interaction entre le système (pour être précis : le *shell*) et une application, il y a les notions de :

— entrée standard (*standard input*, en abrégé `stdin`) ;

— sortie standard (*standard output*, en abrégé `stdout`) ;

— erreur standard (*standard error*, en abrégé `stderr`).

Ceci est principalement pertinent dans le contexte d'un terminal. L'idée c'est que l'on a envie de pouvoir *rediriger* les entrées-sorties d'un programme sans avoir à le modifier. De la sorte, on peut également *chaîner* des traitements à l'aide de *pipes*, sans avoir besoin de sauver les résultats intermédiaires sur disque.

Ainsi par exemple lorsque l'on écrit :

```
$ monprogramme < fichier_entree > fichier_sortie
```

Les deux fichiers en question sont ouverts par le *shell*, et passés à `monprogramme` - que celui-ci soit écrit en C, en Python ou en Java - sous la forme des fichiers `stdin` et `stdout` respectivement, et donc **déjà ouverts**.

Le module `sys`

L'interpréteur Python vous expose ces trois fichiers sous la forme d'attributs du module `sys` :

```
In [1]: import sys
        for channel in (sys.stdin, sys.stdout, sys.stderr):
            print(channel)
```

```
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='cp1252'>
<ipykernel.iostream.OutStream object at 0x041774F0>
<ipykernel.iostream.OutStream object at 0x04177CF0>
```

Dans le contexte du notebook vous pouvez constater que les deux flux de sortie sont implémentés comme des classes spécifiques à IPython. Si vous exécutez ce code localement dans votre ordinateur vous allez sans doute obtenir quelque chose comme :

```
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='UTF-8'>
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
```

On n'a pas extrêmement souvent besoin d'utiliser ces variables en règle générale, mais elles peuvent s'avérer utiles dans des contextes spécifiques.

Par exemple, l'instruction `print` écrit dans `sys.stdout` (c'est-à-dire la sortie standard). Et comme `sys.stdout` est une variable (plus exactement `stdout` est un attribut dans le module référencé par la variable `sys`) et qu'elle référence un objet fichier, on peut lui faire référencer un autre objet fichier et ainsi rediriger depuis notre programme tous les sorties, qui sinon iraient sur le terminal, vers un fichier de notre choix :

```
In [2]: # ici je fais exprès de ne pas utiliser un `with`
        # car très souvent les deux redirections apparaissent
        # dans des fonctions différentes
        import sys
        # on ouvre le fichier destination
        autre_stdout = open('ma_sortie.txt', 'w', encoding='utf-8')
        # on garde un lien vers le fichier sortie standard
        # pour le réinstaller plus tard si besoin.
        tmp = sys.stdout
        print('sur le terminal')

        # première redirection
        sys.stdout = autre_stdout
        print('dans le fichier')

        # on remet comme c'était au début
        sys.stdout = tmp
        # et alors pour être propre on n'oublie pas de fermer
        autre_stdout.close()
        print('de nouveau sur le terminal')
```

```
sur le terminal
de nouveau sur le terminal
```

```
In [3]: # et en effet, dans le fichier on a bien
        with open("ma_sortie.txt", encoding='utf-8') as check:
            print(check.read())
```

```
dans le fichier
```

3.5 La construction de tuples

3.5.1 Complément - niveau intermédiaire

Les tuples et la virgule terminale

Comme on l'a vu dans la vidéo, on peut construire un tuple à deux éléments - un couple - de quatre façons :

```
In [1]: # sans parenthèse ni virgule terminale
        couple1 = 1, 2
        # avec parenthèses
        couple2 = (1, 2)
        # avec virgule terminale
        couple3 = 1, 2,
        # avec parenthèses et virgule
        couple4 = (1, 2,)

In [2]: # toutes ces formes sont équivalentes ; par exemple
        couple1 == couple4

Out[2]: True
```

Comme on le voit :

— en réalité la **parenthèse est parfois superflue**; mais il se trouve qu'elle est **largement utilisée** pour améliorer la lisibilité des programmes, sauf dans le cas du *tuple unpacking*; nous verrons aussi plus bas qu'elle est **parfois nécessaire** selon l'endroit où le tuple apparaît dans le programme;

— la **dernière virgule est optionnelle** aussi, c'est le cas pour les tuples à au moins 2 éléments - nous verrons plus bas le cas des tuples à un seul élément.

Conseil pour la présentation sur plusieurs lignes

En général d'ailleurs, la forme avec parenthèses et virgule terminale est plus pratique. Considérez par exemple l'initialisation suivante; on veut créer un tuple qui contient des listes (naturellement un tuple peut contenir n'importe quel objet Python), et comme c'est assez long on préfère mettre un élément du tuple par ligne :

```
In [3]: mon_tuple = ([1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9],
                    )
```

L'avantage lorsqu'on choisit cette forme (avec parenthèses, et avec virgule terminale), c'est d'abord qu'il n'est pas nécessaire de mettre un backslash à la fin de chaque ligne; parce que l'on est à l'intérieur d'une zone parenthésée, l'interpréteur Python "sait" que l'instruction n'est pas terminée et va se continuer sur la ligne suivante.

Deuxièmement, si on doit ultérieurement ajouter ou enlever un élément dans le tuple, il suffira d'enlever ou d'ajouter toute une ligne, sans avoir à s'occuper des virgules; si on avait choisi de ne pas faire figurer la virgule terminale, alors pour ajouter un élément dans le tuple après le dernier, il ne faut pas oublier d'ajouter une virgule à la ligne précédente. Cette simplicité se répercute au niveau du gestionnaire de code source, où les différences dans le

code sont plus faciles à visualiser.

Signalons enfin que ceci n'est pas propre aux tuples. La virgule terminale est également optionnelle pour les listes, ainsi d'ailleurs que pour tous les types Python où cela fait du sens, comme les dictionnaires et les ensembles que nous verrons bientôt. Et dans tous les cas où on opte pour une présentation multi-lignes, il est conseillé de faire figurer une virgule terminale.

Tuples à un élément

Pour revenir à présent sur le cas des tuples à un seul élément, c'est un cas particulier, parmi les quatre syntaxes que l'on a vues ci-dessus, on obtiendrait dans ce cas :

```
In [4]: # ATTENTION : ces deux premières formes ne construisent pas un tuple !
        simple1 = 1
        simple2 = (1)
        # celles-ci par contre construisent bien un tuple
        simple3 = 1,
        simple4 = (1,)
```

— Il est bien évident que la première forme ne crée pas de tuple ;
— et en fait la seconde non plus, car Python lit ceci comme une expression parenthésée, avec seulement un entier.

Et en fait ces deux premières formes créent un entier simple :

```
In [5]: type(simple2)

Out[5]: int
```

Les deux autres formes créent par contre toutes les deux un tuple à un élément comme on cherchait à le faire :

```
In [6]: type(simple3)

Out[6]: tuple

In [7]: simple3 == simple4

Out[7]: True
```

Pour conclure, disons donc qu'il est conseillé de **toujours mentionner une virgule terminale** lorsqu'on construit des tuples.

Parenthèse parfois obligatoire

Dans certains cas vous vous apercevrez que la parenthèse est obligatoire. Par exemple on peut écrire :

```
In [8]: x = (1,)
        (1,) == x

Out[8]: True
```

Mais si on essaie d'écrire le même test sans les parenthèses :

```
In [9]: # ceci provoque une SyntaxError
        1, == x


          File "<ipython-input-9-219921aa55c4>", line 2
        1, == x
           ^
    SyntaxError: invalid syntax
```

Python lève une erreur de syntaxe ; encore une bonne raison pour utiliser les parenthèses.

Addition de tuples

Bien que le type tuple soit immuable, il est tout à fait légal d'additionner deux tuples, et l'addition va produire un **nouveau** tuple :

```
In [10]: tuple1 = (1, 2,)
         tuple2 = (3, 4,)
         print('addition', tuple1 + tuple2)

addition (1, 2, 3, 4)
```

Ainsi on peut également utiliser l'opérateur += avec un tuple qui va créer, comme précédemment, un nouvel objet tuple :

```
In [11]: tuple1 = (1, 2,)
         tuple1 += (3, 4,)
         print('apres ajout', tuple1)

apres ajout (1, 2, 3, 4)
```

Construire des tuples élaborés

Malgré la possibilité de procéder par additions successives, la construction d'un tuple peut s'avérer fastidieuse.

Une astuce utile consiste à penser aux fonctions de conversion, pour construire un tuple à partir de - par exemple - une liste. Ainsi on peut faire par exemple ceci :

```
In [12]: # on fabrique une liste pas à pas
         liste = list(range(10))
         liste[9] = 'Inconnu'
         del liste [2:5]
         liste

Out[12]: [0, 1, 5, 6, 7, 8, 'Inconnu']

In [13]: # on convertit le résultat en tuple
         mon_tuple = tuple(liste)
         mon_tuple

Out[13]: (0, 1, 5, 6, 7, 8, 'Inconnu')
```

Digression sur les noms de fonctions prédéfinies

Remarque : Vous avez peut-être observé que nous avons choisi de ne pas appeler notre tuple simplement `tuple`. C'est une bonne pratique en général d'éviter les noms de fonctions prédéfinies par Python.

Ces variables en effet sont des variables "comme les autres". Imaginez qu'on ait en fait deux tuples à construire comme ci-dessus, voici ce qu'on obtiendrait si on n'avait pas pris cette précaution :

```
In [14]: liste = range(10)
         # ATTENTION : ceci redéfinit le symbole tuple
         tuple = tuple(liste)
         tuple

Out[14]: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

In [15]: # si bien que maintenant on ne peut plus faire ceci
         # car à ce point, tuple ne désigne plus le type tuple
         # mais l'objet qu'on vient de créer
         autre_liste = range(100)
         autre_tuple = tuple(autre_liste)


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-15-15ef2d1e6804> in <module>()
          3 # mais l'objet qu'on vient de créer
          4 autre_liste = range(100)
    ----> 5 autre_tuple = tuple(autre_liste)


        TypeError: 'tuple' object is not callable
```

Il y a une erreur parce que nous avons remplacé (ligne 2) la valeur de la variable `tuple`, qui au départ référençait le **type** tuple (ou si on préfère la fonction de conversion), par un **objet** tuple. Ainsi en ligne 5, lorsqu'on appelle à nouveau `tuple`, on essaie d'exécuter un objet qui n'est pas 'appelable' (*not callable* en anglais).

D'un autre côté, l'erreur est relativement facile à trouver dans ce cas. En cherchant toutes les occurrences de `tuple` dans notre propre code on voit assez vite le problème. De plus, je vous rappelle que votre éditeur de texte **doit** faire de la coloration syntaxique, et que toutes les fonctions built-in (dont `tuple` et `list` font partie) sont colorées spécifiquement (par exemple, en violet sous IDLE). En pratique, avec un bon éditeur de texte et un peu d'expérience, cette erreur est très rare.

3.6 Sequence unpacking

3.6.1 Complément - niveau basique

Remarque préliminaire : nous avons vainement cherché une traduction raisonnable pour ce trait du langage, connue en anglais sous le nom de *sequence unpacking* ou encore parfois *tuple unpacking*, aussi pour éviter de créer de la confusion nous avons finalement décidé de conserver le terme anglais à l'identique.

Déjà rencontré

L'affectation dans Python peut concerner plusieurs variables à la fois. En fait nous en avons déjà vu un exemple en Semaine 1, avec la fonction `fibonacci` dans laquelle il y avait ce fragment :

```
for i in range(2, n + 1):
    f2, f1 = f1, f1 + f2
```

Nous allons dans ce complément décortiquer les mécanismes derrière cette phrase qui a probablement excité votre curiosité. :)

Un exemple simple

Commençons par un exemple simple à base de tuple. Imaginons que l'on dispose d'un tuple `couple` dont on sait qu'il a deux éléments :

```
In [1]: couple = (100, 'spam')
```

On souhaite à présent extraire les deux valeurs, et les affecter à deux variables distinctes. Une solution naïve consiste bien sûr à faire simplement :

```
In [2]: gauche = couple[0]
        droite = couple[1]
        print('gauche', gauche, 'droite', droite)

gauche 100 droite spam
```

Cela fonctionne naturellement très bien, mais n'est pas très pythonique - comme on dit ;) Vous devez toujours garder en tête qu'il est rare en Python de manipuler des indices. Dès que vous voyez des indices dans votre code, vous devez vous demander si votre code est pythonique.

On préfèrera la formulation équivalente suivante :

```
In [3]: (gauche, droite) = couple
        print('gauche', gauche, 'droite', droite)

gauche 100 droite spam
```

La logique ici consiste à dire : affecter les deux variables de sorte que le tuple `(gauche, droite)` soit égal à `couple`. On voit ici la supériorité de cette notion d'unpacking sur la manipulation d'indices : vous avez maintenant des variables qui expriment la nature de l'objet manipulé, votre code devient expressif, c'est-à-dire auto-documenté.

Remarquons que les parenthèses ici sont optionnelles - comme lorsque l'on construit un tuple - et on peut tout aussi bien écrire, et c'est le cas d'usage le plus fréquent d'omission des parenthèses pour le tuple :

```
In [4]: gauche, droite = couple
        print('gauche', gauche, 'droite', droite)

gauche 100 droite spam
```

Autres types

Cette technique fonctionne aussi bien avec d'autres types. Par exemple, on peut utiliser :

— une syntaxe de liste à gauche du =;
— une liste comme expression à droite du =.

```
In [5]: # comme ceci
        liste = [1, 2, 3]
        [gauche, milieu, droit] = liste
        print('gauche', gauche, 'milieu', milieu, 'droit', droit)

gauche 1 milieu 2 droit 3
```

Et on n'est même pas obligés d'avoir le même type à gauche et à droite du signe =, comme ici :

```
In [6]: # membre droit: une liste
        liste = [1, 2, 3]
        # membre gauche : un tuple
        gauche, milieu, droit = liste
        print('gauche', gauche, 'milieu', milieu, 'droit', droit)

gauche 1 milieu 2 droit 3
```

En réalité, les seules contraintes fixées par Python sont que :

— le terme à droite du signe = soit un *itérable* (tuple, liste, string, etc.);
— le terme à gauche soit écrit comme un tuple ou une liste - notons tout de même que l'utilisation d'une liste à gauche est rare et peu pythonique;
— les deux termes aient la même longueur - en tout cas avec les concepts que l'on a vus jusqu'ici, mais voir aussi plus bas l'utilisation de `*arg` avec le *extended unpacking*.

La plupart du temps le terme de gauche est écrit comme un tuple. C'est pour cette raison que les deux termes *tuple unpacking* et *sequence unpacking* sont en vigueur.

La façon *pythonique* d'échanger deux variables

Une caractéristique intéressante de l'affectation par *sequence unpacking* est qu'elle est sûre ; on n'a pas à se préoccuper d'un éventuel ordre d'évaluation, les valeurs **à droite** de l'affectation sont **toutes** évaluées en premier, et ainsi on peut par exemple échanger deux variables comme ceci :

```
In [7]: a = 1
        b = 2
        a, b = b, a
        print('a', a, 'b', b)
```

```
a 2 b 1
```

Extended unpacking

Le *extended unpacking* a été introduit en Python 3 ; commençons par en voir un exemple :

```
In [8]: reference = [1, 2, 3, 4, 5]
        a, *b, c = reference
        print(f"a={a} b={b} c={c}")
```

```
a=1 b=[2, 3, 4] c=5
```

Comme vous le voyez, le mécanisme ici est une extension de *sequence unpacking* ; Python vous autorise à mentionner **une seule fois**, parmi les variables qui apparaissent à gauche de l'affectation, une variable **précédée de** `*`, ici `*b`.

Cette variable est interprétée comme une **liste de longueur quelconque** des éléments de `reference`. On aurait donc aussi bien pu écrire :

```
In [9]: reference = range(20)
        a, *b, c = reference
        print(f"a={a} b={b} c={c}")
```

```
a=0 b=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18] c=19
```

Ce trait peut s'avérer pratique, lorsque par exemple on s'intéresse seulement aux premiers éléments d'une structure :

```
In [10]: # si on sait que data contient prenom, nom, et un nombre inconnu d'autres informatio
         data = [ 'Jean', 'Dupont', '061234567', '12', 'rue du chemin vert', '57000', 'METZ',
         # on peut utiliser la variable _ qui véhicule l'idée que l'on ne s'y intéresse pas v
         prenom, nom, *_ = data
         print(f"prenom={prenom} nom={nom}")
```

```
prenom=Jean nom=Dupont
```

3.6.2 Complément - niveau intermédiaire

On a vu les principaux cas d'utilisation de la *sequence unpacking*, voyons à présent quelques subtilités.

Plusieurs occurrences d'une même variable

On peut utiliser **plusieurs fois** la même variable dans la partie gauche de l'affectation :

```
In [11]: # ceci en toute rigueur est légal
         # mais en pratique on évite de le faire
         entree = [1, 2, 3]
         a, a, a = entree
         print(f"a = {a}")

a = 3
```

Attention toutefois, comme on le voit ici, Python **n'impose pas** que les différentes occurrences de `a` correspondent **à des valeurs identiques** (en langage savant, on dirait que cela ne permet pas de faire de l'unification). De manière beaucoup plus pragmatique, l'interpréteur se contente de faire comme s'il faisait l'affectation plusieurs fois de gauche à droite, c'est-à-dire comme s'il faisait :

```
In [ ]: a = 1; a = 2; a = 3
```

Cette technique n'est utilisée en pratique que pour les parties de la structure dont on n'a que faire dans le contexte. Dans ces cas-là, il arrive qu'on utilise le nom de variable `_`, dont on rappelle qu'il est légal, ou tout autre nom comme `ignored` pour manifester le fait que cette partie de la structure ne sera pas utilisée, par exemple :

```
In [12]: entree = [1, 2, 3]

         _, milieu, _ = entree
         print('milieu', milieu)

         ignored, ignored, right = entree
         print('right', right)

milieu 2
right 3
```

En profondeur

Le *sequence unpacking* ne se limite pas au premier niveau dans les structures, on peut extraire des données plus profondément imbriquées dans la structure de départ; par exemple avec en entrée la liste :

```
In [13]: structure = ['abc', [(1, 2), ([3], 4)], 5]
```

Si on souhaite extraire la valeur qui se trouve à l'emplacement du `3`, on peut écrire :

```
In [14]: (a, (b, ((trois,), c)), d) = structure
         print('trois', trois)

trois 3
```

Ou encore, sans doute un peu plus lisible :

```
In [15]: (a, (b, ([trois], c)), d) = structure
         print('trois', trois)

trois 3
```

Naturellement on aurait aussi bien pu écrire ici quelque chose comme :

```
In [16]: trois = structure[1][1][0][0]
         print('trois', trois)

trois 3
```

Affaire de goût évidemment. Mais n'oublions pas une des phrases du Zen de Python *Flat is better than nested*, ce qui veut dire que ce n'est pas parce que vous pouvez faire des structures imbriquées complexes que vous devez le faire. Bien souvent, cela rend la lecture et la maintenance du code complexe, j'espère que l'exemple précédent vous en a convaincu.

Extended unpacking et profondeur

On peut naturellement ajouter de l'*extended unpacking* à n'importe quel étage d'un *unpacking* imbriqué :

```
In [17]: # un exemple très alambiqué avec plusieurs variables *extended
         tree = [1, 2, [(3, 33, 'three', 'thirty-three')], ( [4, 44, ('forty', 'forty-four')]
         *_,  ((_, *x3, _),), (*_, x4) = tree
         print(f"x3={x3}, x4={x4}")

x3=[33, 'three'], x4=('forty', 'forty-four')
```

Dans ce cas, la limitation d'avoir une seule variable de la forme `*extended` s'applique toujours, naturellement, mais à chaque niveau dans l'imbrication, comme on le voit sur cet exemple.

3.6.3 Pour en savoir plus

— Le PEP (en anglais) qui introduit le *extended unpacking*.

3.7 Plusieurs variables dans une boucle `for`

3.7.1 Complément - niveau basique

Nous avons vu précédemment (séquence 'Les tuples', complément 'Sequence unpacking') la possibilité d'affecter plusieurs variables à partir d'un seul objet, comme ceci :

```
In [1]: item = (1, 2)
        a, b = item
        print(f"a={a} b={b}")

a=1 b=2
```

D'une façon analogue, il est possible de faire une boucle `for` qui itère sur **une seule** liste mais qui *agit* sur **plusieurs variables**, comme ceci :

```
In [2]: entrees = [(1, 2), (3, 4), (5, 6)]
        for a, b in entrees:
            print(f"a={a} b={b}")

a=1 b=2
a=3 b=4
a=5 b=6
```

À chaque itération, on trouve dans `entree` un tuple (d'abord `(1, 2)`, puis à l'itération suivante `(3, 4)`, etc.); à ce stade les variables `a` et `b` vont être affectées à, respectivement, le premier et le deuxième élément du tuple, exactement comme dans le *sequence unpacking*. Cette mécanique est massivement utilisée en Python.

3.7.2 Complément - niveau intermédiaire

La fonction `zip`

Voici un exemple très simple qui utilise la technique que l'on vient de voir.

Imaginons qu'on dispose de deux listes de longueurs égales, dont on sait que les entrées correspondent une à une, comme par exemple :

```
In [3]: villes = ["Paris", "Nice", "Lyon"]
        populations = [2*10**6, 4*10**5, 10**6]
```

Afin d'écrire facilement un code qui "associe" les deux listes entre elles, Python fournit une fonction *built-in* baptisée `zip`; voyons ce qu'elle peut nous apporter sur cet exemple :

```
In [4]: list(zip(villes, populations))

Out[4]: [('Paris', 2000000), ('Nice', 400000), ('Lyon', 1000000)]
```

On le voit, on obtient en retour une liste composée de tuples. On peut à présent écrire une boucle `for` comme ceci :

```
In [5]: for ville, population in zip(villes, populations):
            print(population, "habitants à", ville)
```

```
2000000 habitants à Paris
400000 habitants à Nice
1000000 habitants à Lyon
```

Qui est, nous semble-t-il, beaucoup plus lisible que ce que l'on serait amené à écrire avec des langages plus traditionnels.

Tout ceci se généralise naturellement à plus de deux variables :

```
In [6]: for i, j, k in zip(range(3), range(100, 103), range(200, 203)):
            print(f"i={i} j={j} k={k}")
```

```
i=0 j=100 k=200
i=1 j=101 k=201
i=2 j=102 k=202
```

Remarque : lorsqu'on passe à `zip` des listes de tailles différentes, le résultat est tronqué, c'est l'entrée **de plus petite taille** qui détermine la fin du parcours.

```
In [7]: # on n'itère que deux fois
        # car le premier argument de zip est de taille 2
        for units, tens in zip([1, 2], [10, 20, 30, 40]):
            print(units, tens)
```

```
1 10
2 20
```

La fonction `enumerate`

Une autre fonction très utile permet d'itérer sur une liste avec l'indice dans la liste, il s'agit de `enumerate` :

```
In [8]: for i, ville in enumerate(villes):
            print(i, ville)
```

```
0 Paris
1 Nice
2 Lyon
```

Cette forme est **plus simple** et **plus lisible** que les formes suivantes qui sont équivalentes, mais qui ne sont pas pythoniques :

```
In [9]: for i in range(len(villes)):
            print(i, villes[i])
```

```
0 Paris
1 Nice
2 Lyon
```

```
In [10]: for i, ville in zip(range(len(villes)), villes):
             print(i, ville)

0 Paris
1 Nice
2 Lyon
```

3.8 Fichiers

3.8.1 Exercice - niveau basique

Calcul du nombre de lignes, de mots et de caractères

```
In [ ]: # chargement de l'exercice
        from corrections.exo_comptage import exo_comptage
```

On se propose d'écrire une *moulinette* qui annote un fichier avec des nombres de lignes, de mots et de caractères.

Le but de l'exercice est d'écrire une fonction `comptage` :

— qui prenne en argument un nom de fichier d'entrée (on suppose qu'il existe) et un nom de fichier de sortie (on suppose qu'on a le droit de l'écrire) ;

— le fichier d'entrée est supposé encodé en UTF-8 ;

— le fichier d'entrée est laissé intact ;

— pour chaque ligne en entrée, le fichier de sortie comporte une ligne qui donne le numéro de ligne, le nombre de mots (**séparés par des espaces**), le nombre de caractères (y compris la fin de ligne), et la ligne d'origine.

```
In [ ]: # un exemple de ce qui est attendu
        exo_comptage.example()
```

```
In [ ]: # votre code
        def comptage(in_filename, out_filename):
            "votre code"
```

N'oubliez pas de vérifier que vous ajoutez bien les **fins de ligne**, car la vérification automatique est pointilleuse (elle utilise l'opérateur ==), et rejettera votre code si vous ne produisez pas une sortie rigoureusement similaire à ce qui est attendu.

```
In [ ]: # pour vérifier votre code
        # voyez aussi un peu plus bas, une cellule d'aide au debugging

        exo_comptage.correction(comptage)
```

La méthode `debug` applique votre fonction au premier fichier d'entrée, et affiche le résultat comme dans l'exemple ci-dessus :

```
In [ ]: # debugging
        exo_comptage.debug(comptage)
```

Accès aux fichiers d'exemples

Vous pouvez télécharger les fichiers d'exemples :

— Romeo and Juliet

— Lorem Ipsum

— "Une charogne" en UTF-8

Pour les courageux, je vous donne également "Une charogne" en ISO-8859-15, qui contient le même texte que "Une charogne", mais encodé en Latin-9, connu aussi sous le nom ISO-8859-15.

Ce dernier fichier n'est pas à prendre en compte dans la version basique de l'exercice, mais vous pourrez vous rendre compte par vous-mêmes, au cas où cela ne serait pas clair encore pour vous, qu'il n'est pas facile d'écrire une fonction `comptage` qui devine l'encodage, c'est-à-dire qui fonctionne correctement avec des entrées indifféremment en Unicode ou Latin, sans que cet encodage soit passé en paramètre à `comptage`.

C'est d'ailleurs le propos de la bibliothèque `chardet` qui s'efforce de déterminer l'encodage de fichiers d'entrée, sur la base de modèles statistiques.

3.9 *Sequence unpacking*

3.9.1 Exercice - niveau basique

```
In [ ]: # chargeons l'exercice
        from corrections.exo_surgery import exo_surgery
```

Cet exercice consiste à écrire une fonction `surgery`, qui prend en argument une liste, et qui retourne la **même** liste **modifiée** comme suit :

— si la liste est de taille 0 ou 1, elle n'est pas modifiée ;

— si la liste est de taille paire, on intervertit les deux premiers éléments de la liste ;

— si elle est de taille impaire, on intervertit les deux derniers éléments.

```
In [ ]: # voici quelques exemples de ce qui est attendu
        exo_surgery.example()
```

```
In [ ]: # écrivez votre code
        def surgery(liste):
            "<votre_code>"
```

```
In [ ]: # pour le vérifier, évaluez cette cellule
        exo_surgery.correction(surgery)
```

3.10 Dictionnaires

3.10.1 Complément - niveau basique

Ce document résume les opérations courantes disponibles sur le type `dict`. On rappelle que le type `dict` est un type **mutable**.

Création en extension

On l'a vu, la méthode la plus directe pour créer un dictionnaire est en extension comme ceci :

```
In [1]: annuaire = {'marc': 35, 'alice': 30, 'eric': 38}
        print(annuaire)
```

```
{'marc': 35, 'alice': 30, 'eric': 38}
```

Création - la fonction dict

Comme pour les fonctions `int` ou `list`, la fonction `dict` est une fonction de construction de dictionnaire - on dit un constructeur. On a vu aussi dans la vidéo qu'on peut utiliser ce constructeur à base d'une liste de tuples (`clé`, `valeur`)

```
In [2]: # le paramètre de la fonction dict est
        # une liste de couples (clé, valeur)
        annuaire = dict([('marc', 35), ('alice', 30), ('eric', 38)])
        print(annuaire)
```

```
{'marc': 35, 'alice': 30, 'eric': 38}
```

Remarquons qu'on peut aussi utiliser cette autre forme d'appel à `dict` pour un résultat équivalent :

```
In [3]: annuaire = dict(marc=35, alice=30, eric=38)
        print(annuaire)
```

```
{'marc': 35, 'alice': 30, 'eric': 38}
```

Remarquez ci-dessus l'absence de quotes autour des clés comme `marc`. Il s'agit d'un cas particulier de passage d'arguments que nous expliciterons plus longuement en fin de semaine 4.

Accès atomique

Pour accéder à la valeur associée à une clé, on utilise la notation à base de crochets `[]` :

```
In [4]: print('la valeur pour marc est', annuaire['marc'])
```

```
la valeur pour marc est 35
```

Cette forme d'accès ne fonctionne que si la clé est effectivement présente dans le dictionnaire. Dans le cas contraire, une exception `KeyError` est levée. Aussi si vous n'êtes pas sûr que la clé soit présente, vous pouvez utiliser la méthode `get` qui accepte une valeur par défaut :

```
In [5]: print('valeur pour marc', annuaire.get('marc', 0))
        print('valeur pour inconnu', annuaire.get('inconnu', 0))

valeur pour marc 35
valeur pour inconnu 0
```

Le dictionnaire est un type **mutable**, et donc on peut **modifier la valeur** associée à une clé :

```
In [6]: annuaire['eric'] = 39
        print(annuaire)

{'marc': 35, 'alice': 30, 'eric': 39}
```

Ou encore, exactement de la même façon, **ajouter une entrée** :

```
In [7]: annuaire['bob'] = 42
        print(annuaire)

{'marc': 35, 'alice': 30, 'eric': 39, 'bob': 42}
```

Enfin pour **détruire une entrée**, on peut utiliser l'instruction `del` comme ceci :

```
In [8]: # pour supprimer la clé 'marc' et donc sa valeur aussi
        del annuaire['marc']
        print(annuaire)

{'alice': 30, 'eric': 39, 'bob': 42}
```

Pour savoir si une clé est présente ou non, il est conseillé d'utiliser l'opérateur d'appartenance `in` comme ceci :

```
In [9]: # forme recommandée
        print('john' in annuaire)

False
```

Parcourir toutes les entrées

La méthode la plus fréquente pour parcourir tout un dictionnaire est à base de la méthode `items` ; voici par exemple comment on pourrait afficher le contenu :

```
In [10]: for nom, age in annuaire.items():
             print(f"{nom}, age {age}")
```

```
alice, age 30
eric, age 39
bob, age 42
```

On remarque d'abord que les entrées sont listées dans le désordre, plus précisément, il n'y a pas de notion d'ordre dans un dictionnaire ; ceci est dû à l'action de la fonction de hachage, que nous avons vue dans la vidéo précédente.

On peut obtenir séparément la liste des clés et des valeurs avec :

```
In [11]: for clé in annuaire.keys():
             print(clé)

alice
eric
bob
```

```
In [12]: for valeur in annuaire.values():
             print(valeur)

30
39
42
```

La fonction `len`

On peut comme d'habitude obtenir la taille d'un dictionnaire avec la fonction `len` :

```
In [13]: print(f"{len(annuaire)} entrées dans annuaire")

3 entrées dans annuaire
```

Pour en savoir plus sur le type `dict`

Pour une liste exhaustive reportez-vous à la page de la documentation Python ici :

https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

3.10.2 Complément - niveau intermédiaire

La méthode `update`

On peut également modifier un dictionnaire avec le contenu d'un autre dictionnaire avec la méthode `update` :

```
In [14]: print(f"avant: {list(annuaire.items())}")

avant: [('alice', 30), ('eric', 39), ('bob', 42)]
```

```
In [15]: annuaire.update({'jean':25, 'eric':70})
         list(annuaire.items())

Out[15]: [('alice', 30), ('eric', 70), ('bob', 42), ('jean', 25)]
```

`collections.OrderedDict` : dictionnaire et ordre d'insertion

Attention : un dictionnaire est **non ordonné !** Il ne se souvient pas de l'ordre dans lequel les éléments ont été insérés. C'était particulièrement visible dans les versions de Python jusque 3.5 :

```
In [ ]: %%python2

        # cette cellule utilise python-2.7 pour illustrer le fait
        # que les dictionnaires ne sont pas ordonnés

        d = {'c' : 3, 'b' : 1, 'a' : 2}
        for k, v in d.items():
            print k, v
```

En réalité, et depuis la version 3.6 de Python, il se trouve qu'**incidemment** l'implémentation CPython (la plus répandue donc) a été modifiée, et maintenant on peut avoir l'**impression** que les dictionnaires sont ordonnés :

```
In [17]: d = {'c' : 3, 'b' : 1, 'a' : 2}
         for k, v in d.items():
             print(k, v)

c 3
b 1
a 2
```

Il faut insister sur le fait qu'il s'agit d'un **détail d'implémentation**, et que vous ne devez pas écrire du code qui suppose que les dictionnaires sont ordonnés.

Si vous avez besoin de dictionnaires qui sont **garantis** ordonnés, voyez dans le module `collections` la classe `OrderedDict`, qui est une personnalisation (une sous-classe) du type `dict`, qui cette fois possède cette bonne propriété :

```
In [18]: from collections import OrderedDict
         d = OrderedDict()
         for i in ['a', 7, 3, 'x']:
             d[i] = i
         for k, v in d.items():
             print('OrderedDict', k, v)
```

```
OrderedDict a a
OrderedDict 7 7
OrderedDict 3 3
OrderedDict x x
```

`collections.defaultdict` : initialisation automatique

Imaginons que vous devez gérer un dictionnaire dont les valeurs sont des listes, et que votre programme ajoute des valeurs au fur et à mesure dans ces listes.

Avec un dictionnaire de base, cela peut vous amener à écrire un code qui ressemble à ceci :

```
In [19]: # imaginons qu'on a lu dans un fichier des couples (x, y)
         tuples = [
             (1, 2),
             (2, 1),
             (1, 3),
             (2, 4),
         ]
```

```
In [20]: # et on veut construire un dictionnaire
         # x -> [liste de tous les y connectés à x]
         resultat = {}

         for x, y in tuples:
             if x not in resultat:
                 resultat[x] = []
             resultat[x].append(y)

         for key, value in resultat.items():
             print(key, value)
```

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

Cela fonctionne, mais n'est pas très élégant. Pour simplifier ce type de traitement, vous pouvez utiliser `defaultdict`, une sous-classe de `dict` dans le module `collections` :

```
In [21]: from collections import defaultdict

         # on indique que les valeurs doivent être créées à la volée
         # en utilisant la fonction list
         resultat = defaultdict(list)
```

```
        # du coup plus besoin de vérifier la présence de la clé
        for x, y in tuples:
            resultat[x].append(y)

        for key, value in resultat.items():
            print(key, value)
```

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

Cela fonctionne aussi avec le type `int`, lorsque vous voulez par exemple compter des occurrences :

```
In [22]: compteurs = defaultdict(int)

         phrase = "une phrase dans laquelle on veut compter les caractères"

         for c in phrase:
             compteurs[c] += 1

         sorted(compteurs.items())
```

```
Out[22]: [(' ', 8),
          ('a', 5),
          ('c', 3),
          ('d', 1),
          ('e', 8),
          ('h', 1),
          ('l', 4),
          ('m', 1),
          ('n', 3),
          ('o', 2),
          ('p', 2),
          ('q', 1),
          ('r', 4),
          ('s', 4),
          ('t', 3),
          ('u', 3),
          ('v', 1),
          ('è', 1)]
```

Signalons enfin une fonctionnalité un peu analogue, quoiqu'un peut moins élégante à mon humble avis, mais qui est présente avec les dictionnaires `dict` standard. Il s'agit de la méthode `setdefault` qui permet, en un seul appel, de retourner la valeur associée à une clé et de créer cette clé au besoin, c'est-à-dire si elle n'est pas encore présente :

```
In [23]: # avant
         annuaire
```

```
Out[23]: {'alice': 30, 'bob': 42, 'eric': 70, 'jean': 25}
```

```
In [24]: # ceci sera sans effet car eric est déjà présent
         annuaire.setdefault('eric', 50)
```

```
Out[24]: 70

In [25]: # par contre ceci va insérer une entrée dans le dictionnaire
         annuaire.setdefault('inconnu', 50)

Out[25]: 50

In [26]: # comme on le voit
         annuaire

Out[26]: {'alice': 30, 'bob': 42, 'eric': 70, 'inconnu': 50, 'jean': 25}
```

Notez bien que `setdefault` peut éventuellement créer une entrée mais ne **modifie jamais** la valeur associée à une clé déjà présente dans le dictionnaire, comme le nom le suggère d'ailleurs.

3.10.3 Complément - niveau avancé

Pour bien appréhender les dictionnaires, il nous faut souligner certaines particularités, à propos de la valeur de retour des méthodes comme `items()`, `keys()` et `values()`.

Ce sont des objets itérables

Les méthodes `items()`, `keys()` et `values()` ne retournent pas des listes (comme c'était le cas en Python 2), mais des **objets itérables** :

```
In [27]: d = {'a' : 1, 'b' : 2}
         keys = d.keys()
         keys

Out[27]: dict_keys(['a', 'b'])
```

Comme ce sont des itérables, on peut naturellement faire un `for` avec, on l'a vu :

```
In [28]: for key in keys:
             print(key)

a
b
```

Et un test d'appartenance avec `in` :

```
In [29]: print('a' in keys)

True


In [30]: print('x' in keys)

False
```

Mais ce ne sont pas des listes

```
In [31]: isinstance(keys, list)
```

```
Out[31]: False
```

Ce qui signifie qu'on n'a **pas alloué de mémoire** pour stocker toutes les clés, mais seulement un objet qui ne prend pas de place, ni de temps à construire :

```
In [32]: # construisons un dictionnaire
         # pour anticiper un peu sur la compréhension de dictionnaire

         big_dict = {k : k**2 for k in range(1_000_000)}
```

```
In [33]: %%timeit -n 10000
         # créer un objet vue est très rapide
         big_keys = big_dict.keys()
```

```
193 ns ± 51 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
In [34]: # on répète ici car timeit travaille dans un espace qui lui est propre
         # et donc on n'a pas défini big_keys pour notre interpréteur
         big_keys = big_dict.keys()
```

```
In [35]: %%timeit -n 20
         # si on devait vraiment construire la liste ce serait beaucoup plus long
         big_lkeys = list(big_keys)
```

```
20.1 ms ± 391 µs per loop (mean ± std. dev. of 7 runs, 20 loops each)
```

En fait ce sont des *vues* Une autre propriété un peu inattendue de ces objets, c'est que **ce sont des vues**; ce qu'on veut dire par là (pour ceux qui connaissent, cela fait fait référence à la notion de vue dans les bases de données) c'est que la vue *voit* les changements fait sur l'objet dictionnaire *même après sa création* :

```
In [36]: d = {'a' : 1, 'b' : 2}
         keys = d.keys()
```

```
In [37]: # sans surprise, il y a deux clés dans keys
         for k in keys:
             print(k)
```

```
a
b
```

```
In [38]: # mais si maintenant j'ajoute un objet au dictionnaire
         d['c'] = 3
         # alors on va 'voir' cette nouvelle clé à partir
         # de l'objet keys qui pourtant est inchangé
         for k in keys:
             print(k)
```

```
a
b
c
```

Reportez vous à la section sur les vues de dictionnaires pour plus de détails.

Python 2 Ceci est naturellement en fort contraste avec tout ce qui se passait en Python 2, où l'on avait des méthodes distinctes, par exemple `keys()`, `iterkeys()` et `viewkeys()`, selon le type d'objets que l'on souhaitait construire.

3.11 Clés immuables

3.11.1 Complément - niveau intermédiaire

Nous avons vu comment manipuler un dictionnaire, il nous reste à voir un peu plus en détail les contraintes qui sont mises par le langage sur ce qui peut servir de clé dans un dictionnaire. On parle dans ce complément spécifiquement des clefs construites à partir des types `built-in`. Le cas de vos propres classes utilisées comme clefs de dictionnaires n'est pas abordé dans ce complément.

Une clé doit être immuable

Si vous vous souvenez de la vidéo sur les tables de hash, la mécanique interne du dictionnaire repose sur le calcul, à partir de chaque clé, d'une fonction de hachage.

C'est-à-dire que, pour simplifier, on localise la présence d'une clé en calculant d'abord $f(cl) = hash$

puis on poursuit la recherche en utilisant *hash* comme indice dans le tableau contenant les couples (clé, valeur).

On le rappelle, c'est cette astuce qui permet de réaliser les opérations sur les dictionnaires en temps constant - c'est-à-dire indépendamment du nombre d'éléments.

Cependant, pour que ce mécanisme fonctionne, il est indispensable que **la valeur de la clé reste inchangée** pendant la durée de vie du dictionnaire. Sinon, bien entendu, on pourrait avoir le scénario suivant :

— on range un tuple `(clef, valeur)` à un premier indice $f(clef) = hash_1$;

— on modifie la valeur de $clef$ qui devient $clef'$;

— on recherche notre valeur à l'indice $f(clef') = hash_2 \neq hash_1$.

et donc, avec ces hypothèses, on n'a plus la garantie de bon fonctionnement de la logique.

Une clé doit être globalement immuable

Nous avons depuis le début du cours longuement insisté sur le caractère mutable ou immuable des différents types prédéfinis de Python. Vous devez donc à présent avoir au moins en partie ce tableau en tête :

Type	Mutable ?
`int, float`	immuable
`complex,bool`	immuable
`str`	immuable
`list`	mutable
`dict`	mutable
`set`	mutable
`frozenset`	immuable

Le point important ici, est qu'il **ne suffit pas**, pour une clé, d'être **de type immuable**.

On peut le voir sur un exemple très simple ; donnons-nous donc un dictionnaire :

```
In [1]: d = {}
```

Et commençons avec un objet de type immuable, un tuple d'entiers :

```
In [2]: bonne_cle = (1, 2)
```

Cet objet est non seulement **de type immuable**, mais tous ses composants et sous-composants sont **immuables**, on peut donc l'utiliser comme clé dans le dictionnaire :

```
In [3]: d[bonne_cle] = "pas de probleme ici"
        print(d)
```

```
{(1, 2): 'pas de probleme ici'}
```

Si à présent on essaie d'utiliser comme clé un tuple qui contient une liste :

```
In [4]: mauvaise_cle = (1, [1, 2])
```

Il se trouve que cette clé, **bien que de type immuable**, peut être **indirectement modifiée** puisque :

```
In [5]: mauvaise_cle[1].append(3)
        print(mauvaise_cle)
```

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

Et c'est pourquoi on ne peut pas utiliser cet objet comme clé dans le dictionnaire :

```
In [6]: # provoque une exception
        d[mauvaise_cle] = 'on ne peut pas faire ceci'
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-6-08a3625a4eed> in <module>()
          1 # provoque une exception
    ----> 2 d[mauvaise_cle] = 'on ne peut pas faire ceci'


        TypeError: unhashable type: 'list'
```

Pour conclure, il faut retenir qu'un objet n'est éligible pour être utilisé comme clé que s'il est **composé de types immuables de haut en bas** de la structure de données.

La raison d'être principale du type `tuple`, que nous avons vu la semaine passée, et du type `frozenset`, que nous verrons très prochainement, est précisément de construire de tels objets globalement immuables.

Épilogue

Tout ceci est valable pour les types *built-in*. Nous verrons que pour les types définis par l'utilisateur - les classes donc - que nous effleurons à la fin de cette semaine et que nous étudions plus en profondeur en semaine 6, c'est un autre mécanisme qui est utilisé pour calculer la clé de hachage d'une instance de classe.

3.12 Gérer des enregistrements

3.12.1 Complément - niveau intermédiaire

Implémenter un enregistrement comme un dictionnaire

Il nous faut faire le lien entre dictionnaire Python et la notion d'enregistrement, c'est-à-dire une donnée composite qui contient plusieurs champs. (À cette notion correspond, selon les langages, ce qu'on appelle un `struct` ou un `record`.)

Imaginons qu'on veuille manipuler un ensemble de données concernant des personnes; chaque personne est supposée avoir un nom, un âge et une adresse mail.

Il est possible, et assez fréquent, d'utiliser le dictionnaire comme support pour modéliser ces données comme ceci :

```
In [1]: personnes = [
            {'nom': 'Pierre',  'age': 25, 'email': 'pierre@example.com'},
            {'nom': 'Paul',    'age': 18, 'email': 'paul@example.com'},
            {'nom': 'Jacques', 'age': 52, 'email': 'jacques@example.com'},
        ]
```

Bon, très bien, nous avons nos données, il est facile de les utiliser.

Par exemple, pour l'anniversaire de Pierre on fera :

```
In [2]: personnes[0]['age'] += 1
```

Ce qui nous donne :

```
In [3]: for personne in personnes:
            print(10*"=")
            for info, valeur in personne.items():
                print(f"{info} -> {valeur}")
```

```
==========
nom -> Pierre
age -> 26
email -> pierre@example.com
==========
nom -> Paul
age -> 18
email -> paul@example.com
==========
nom -> Jacques
age -> 52
email -> jacques@example.com
```

Un dictionnaire pour indexer les enregistrements

Cela dit, il est bien clair que cette façon de faire n'est pas très pratique; pour marquer l'anniversaire de Pierre on ne sait bien entendu pas que son enregistrement est le premier dans la liste. C'est pourquoi il est plus adapté, pour modéliser ces informations, d'utiliser non

pas une liste, mais à nouveau… un dictionnaire.

Si on imagine qu'on a commencé par lire ces données séquentiellement dans un fichier, et qu'on a calculé l'objet `personnes` comme la liste qu'on a vue ci-dessus, alors il est possible de construire un index de ces dictionnaires, (un dictionnaire de dictionnaires, donc).

C'est-à-dire, en anticipant un peu sur la construction de dictionnaires par compréhension :

```
In [4]: # on crée un index permettant de retrouver rapidement
        # une personne dans la liste
        index_par_nom = {personne['nom']: personne for personne in personnes}
        index_par_nom

Out[4]: {'Jacques': {'age': 52, 'email': 'jacques@example.com', 'nom': 'Jacques'},
         'Paul': {'age': 18, 'email': 'paul@example.com', 'nom': 'Paul'},
         'Pierre': {'age': 26, 'email': 'pierre@example.com', 'nom': 'Pierre'}}

In [5]: # du coup pour accéder à l'enregistrement pour Pierre
        index_par_nom['Pierre']

Out[5]: {'age': 26, 'email': 'pierre@example.com', 'nom': 'Pierre'}
```

Attardons-nous un tout petit peu ; nous avons construit un dictionnaire par compréhension, en créant autant d'entrées que de personnes. Nous aborderons en détail la notion de compréhension de sets et de dictionnaires en semaine 5, donc si cette notation vous paraît étrange pour le moment, pas d'inquiétude.

Le résultat est donc un dictionnaire qu'on peut afficher comme ceci :

```
In [6]: for nom, record in index_par_nom.items():
            print(f"Nom : {nom} -> enregistrement : {record}")

Nom : Pierre -> enregistrement : {'nom': 'Pierre', 'age': 26, 'email': 'pierre@example.com'}
Nom : Paul -> enregistrement : {'nom': 'Paul', 'age': 18, 'email': 'paul@example.com'}
Nom : Jacques -> enregistrement : {'nom': 'Jacques', 'age': 52, 'email': 'jacques@example.com
```

Dans cet exemple, le premier niveau de dictionnaire permet de trouver rapidement un objet à partir d'un nom ; dans le second niveau au contraire on utilise le dictionnaire pour implémenter un enregistrement, à la façon d'un `struct` en C.

Techniques similaires

Notons enfin qu'il existe aussi, en Python, un autre mécanisme qui peut être utilisé pour gérer ce genre d'objets composites, ce sont les classes que nous verrons en semaine 6, et qui permettent de définir de nouveaux `types` plutôt que, comme nous l'avons fait ici, d'utiliser un type prédéfini. Dans ce sens, l'utilisation d'une classe permet davantage de souplesse, au prix de davantage d'effort.

3.12.2 Complément - niveau avancé

La même idée, mais avec une classe `Personne`

Je vais donner ici une implémentation du code ci-dessus, qui utilise une classe pour modéliser les personnes. Naturellement je n'entre pas dans les détails, que l'on verra en semaine 6, mais j'espère vous donner un aperçu des classes dans un usage réaliste, et vous montrer les avantages de cette approche.

Pour commencer je définis la classe `Personne`, qui va me servir à modéliser chaque personne :

```
In [7]: class Personne:

            # le constructeur - vous ignorez le paramètre self,
            # on pourra construire une personne à partir de
            # 3 paramètres
            def __init__(self, nom, age, email):
                self.nom = nom
                self.age = age
                self.email = email

            # je définis cette méthode pour avoir
            # quelque chose de lisible quand je print()
            def __repr__(self):
                return f"{self.nom} ({self.age} ans) sur {self.email}"
```

Pour construire ma liste de personnes, je fais alors :

```
In [8]: personnes2 = [
            Personne('Pierre',  25, 'pierre@example.com'),
            Personne('Paul',    18, 'paul@example.com'),
            Personne('Jacques', 52, 'jacques@example.com'),
        ]
```

Si je regarde un élément de la liste j'obtiens :

```
In [9]: personnes2[0]
```

```
Out[9]: Pierre (25 ans) sur pierre@example.com
```

Je peux indexer tout ceci comme tout à l'heure, si j'ai besoin d'un accès rapide :

```
In [10]: # je dois utiliser cette fois personne.nom et non plus personne['nom']
         index2 = {personne.nom : personne for personne in personnes2}
```

Le principe ici est exactement identique à ce qu'on a fait avec le dictionnaire de dictionnaires, mais on a construit un dictionnaire d'instances.

Et de cette façon :

```
In [11]: print(index2['Pierre'])
```

```
Pierre (25 ans) sur pierre@example.com
```

Rendez-vous en semaine 6 pour approfondir la notion de classes et d'instances.

3.13 Dictionnaires et listes

3.13.1 Exercice - niveau basique

```
In [ ]: from corrections.exo_graph_dict import exo_graph_dict
```

On veut implémenter un petit modèle de graphes. Comme on a les données dans des fichiers, on veut analyser des fichiers d'entrée qui ressemblent à ceci :

```
In [ ]: !cat data/graph1.txt
```

qui signifierait :

— un graphe à 3 sommets *s1*, *s2* et *s3* ;

— et 4 arêtes

  — une entre *s1* et *s2* de longueur 10 ;

  — une entre *s2* et *s3* de longueur 12 ;

  — etc...

On vous demande d'écrire une fonction qui lit un tel fichier texte, et construit (et retourne) un dictionnaire Python qui représente ce graphe.

Dans cet exercice on choisit :

— de modéliser le graphe comme un dictionnaire indexé sur les (noms de) sommets ;

— et chaque valeur est une liste de tuples de la forme (*suivant*, *longueur*), dans l'ordre d'apparition dans le fichier d'entrée.

```
In [ ]: # voici ce qu'on obtiendrait par exemple avec les données ci-dessus
        exo_graph_dict.example()
```

Notes

— Vous remarquerez que l'exemple ci-dessus retourne un dictionnaire standard ; une solution qui utiliserait `defaultdict` est acceptable également ;

— Notez bien également que dans le résultat, la longueur d'un arc est attendue comme un `int`.

```
In [ ]: # n'oubliez pas d'importer si nécessaire

        # à vous de jouer
        def graph_dict(filename):
            "votre code"
```

```
In [ ]: exo_graph_dict.correction(graph_dict)
```

3.14 Fusionner des données

3.14.1 Exercices

Cet exercice vient en deux versions, une de niveau basique et une de niveau intermédiaire.

La version basique est une application de la technique d'indexation que l'on a vue dans le complément "Gérer des enregistrements". On peut très bien faire les deux versions dans l'ordre, une fois qu'on a fait la version basique on est en principe un peu plus avancé pour aborder la version intermédiaire.

Contexte

Nous allons commencer à utiliser des données un peu plus réalistes. Il s'agit de données obtenues auprès de MarineTraffic - et légèrement simplifiées pour les besoins de l'exercice. Ce site expose les coordonnées géographiques de bateaux observées en mer au travers d'un réseau de collecte de type *crowdsourcing*.

De manière à optimiser le volume de données à transférer, l'API de MarineTraffic offre deux modes pour obtenir les données :

— **mode étendu** : chaque mesure (bateau x position x temps) est accompagnée de tous les détails du bateau (`id`, nom, pays de rattachement, etc.) ;

— **mode abrégé** : chaque mesure est uniquement attachée à l'`id` du bateau.

En effet, chaque bateau possède un identifiant unique qui est un entier, que l'on note `id`.

Chargement des données

Commençons par charger les données de l'exercice :

```
In [ ]: from corrections.exo_marine_dict import extended, abbreviated
```

Format des données

Le format de ces données est relativement simple, il s'agit dans les deux cas d'une liste d'entrées - une par bateau.

Chaque entrée à son tour est une liste qui contient :

```
mode étendu: [id, latitude, longitude, date_heure, nom_bateau, code_pays, ...]
mode abrégé: [id, latitude, longitude, date_heure]
```

sachant que les entrées après le code pays dans le format étendu ne nous intéressent pas pour cet exercice.

```
In [ ]: # une entrée étendue est une liste qui ressemble à ceci
        sample_extended_entry = extended[3]
        print(sample_extended_entry)
```

```
In [ ]: # une entrée abrégée ressemble à ceci
        sample_abbreviated_entry = abbreviated[0]
        print(sample_abbreviated_entry)
```

On précise également que les deux listes `extended` et `abbreviated` :

— possèdent exactement **le même nombre** d'entrées ;
— et correspondent **aux mêmes bateaux** ;
— mais naturellement **à des moments différents** ;
— et **pas forcément dans le même ordre**.

Exercice - niveau basique

```
In [ ]: # chargement de l'exercice
        from corrections.exo_marine_dict import exo_index
```

But de l'exercice On vous demande d'écrire une fonction `index` qui calcule, à partir de la liste des données étendues, un dictionnaire qui est :

— indexé par l'`id` de chaque bateau ;
— et qui a pour valeur la liste qui décrit le bateau correspondant.

De manière plus imagée, si :

```
extended = [ bateau1, bateau2, ... ]
```

Et si :

```
bateau1 = [ id1, latitude, ... ]
```

On doit obtenir comme résultat de `index` un dictionnaire :

```
{
    id1 -> [ id_bateau1, latitude, ... ],
    id2 ...
}
```

Bref, on veut pouvoir retrouver les différents éléments de la liste `extended` par accès direct, en ne faisant qu'une seule recherche dans l'index.

```
In [ ]: # le résultat attendu
        result_index = exo_index.resultat(extended)

        # on en profite pour illustrer le module pprint
        from pprint import pprint

        # à quoi ressemble le résultat pour un bateau au hasard
        for key, value in result_index.items():
            print("==== clé")
            pprint(key)
            print("==== valeur")
            pprint(value)
            break
```

Remarquez ci-dessus l'utilisation d'un utilitaire parfois pratique : le module `pprint` pour pretty-printer.

Votre code

```
In [ ]: def index(extended):
            "<votre_code>"
```

Validation

```
In [ ]: exo_index.correction(index, abbreviated)
```

Vous remarquerez d'ailleurs que la seule chose que l'on utilise dans cet exercice, c'est que l'id des bateaux arrive en première position (dans la liste qui matérialise le bateau), aussi votre code doit marcher à l'identique avec les bateaux étendus :

```
In [ ]: exo_index.correction(index, extended)
```

Exercice - niveau intermédiaire

```
In [ ]: # chargement de l'exercice
        from corrections.exo_marine_dict import exo_merge
```

But de l'exercice On vous demande d'écrire une fonction `merge` qui fasse une consolidation des données, de façon à obtenir en sortie un dictionnaire :

```
id -> [nom_bateau, code_pays, position_etendu, position_abrege]
```

dans lequel les deux objets `position` sont tous les deux des tuples de la forme :

```
(latitude, longitude, date_heure)
```

Voici par exemple un couple clé-valeur dans le résultat attendu :

```
In [ ]: # le résultat attendu
        result_merge = exo_merge.resultat(extended, abbreviated)

        # à quoi ressemble le résultat pour un bateau au hasard
        from pprint import pprint
        for key_value in result_merge.items():
            pprint(key_value)
            break
```

Votre code

```
In [ ]: def merge(extended, abbreviated):
            "votre code"
```

Validation

```
In [ ]: exo_merge.correction(merge, extended, abbreviated)
```

Les fichiers de données complets

Signalons enfin pour ceux qui sont intéressés que les données chargées dans cet exercice sont disponibles au format JSON - qui est précisément celui exposé par marinetraffic.

Nous avons beaucoup simplifié les données d'entrée pour vous permettre une mise au point plus facile. Si vous voulez vous amuser à charger des données un peu plus significatives, sachez que :

— vous avez accès aux fichiers de données plus complets :
  — `data/marine-e1-ext.json`
  — `data/marine-e1-abb.json`
— pour charger ces fichiers, qui sont donc au format JSON, la connaissance intime de ce format n'est pas nécessaire, on peut tout simplement utiliser le module `json`. Voici le code utilisé dans l'exercice pour charger ces JSON en mémoire ; il utilise des notions que nous verrons dans les semaines à venir :

```
In [ ]: # load data from files
        import json

        with open("data/marine-e1-ext.json", encoding="utf-8") as feed:
            extended_full = json.load(feed)

        with open("data/marine-e1-abb.json", encoding="utf-8") as feed:
            abbreviated_full = json.load(feed)
```

Une fois que vous avez un code qui fonctionne vous pouvez le lancer sur ces données plus copieuses en faisant :

```
In [ ]: exo_merge.correction(merge, extended_full, abbreviated_full)
```

3.15 Ensembles

3.15.1 Complément - niveau basique

Ce document résume les opérations courantes disponibles sur le type `set`. On rappelle que le type `set` est un type **mutable**.

Création en extension

On crée un ensemble avec les accolades, comme les dictionnaires, mais sans utiliser le caractère `:`, et cela donne par exemple :

```
In [1]: heteroclite = {'marc', 12, 'pierre', (1, 2, 3), 'pierre'}
        print(heteroclite)

{'pierre', 12, 'marc', (1, 2, 3)}
```

Création - la fonction `set`

Il devrait être clair à ce stade que, le nom du type étant `set`, la fonction `set` est un constructeur d'ensemble. On aurait donc aussi bien pu faire :

```
In [2]: heteroclite2 = set(['marc', 12, 'pierre', (1, 2, 3), 'pierre'])
        print(heteroclite2)

{'pierre', 12, 'marc', (1, 2, 3)}
```

Créer un ensemble vide

Il faut remarquer que l'on ne peut pas créer un ensemble vide en extension. En effet :

```
In [3]: type({})

Out[3]: dict
```

Ceci est lié à des raisons historiques, les ensembles n'ayant fait leur apparition que tardivement dans le langage en tant que citoyen de première classe.

Pour créer un ensemble vide, la pratique la plus courante est celle-ci :

```
In [4]: ensemble_vide = set()
        print(type(ensemble_vide))

<class 'set'>
```

Ou également, moins élégant mais que l'on trouve parfois dans du vieux code :

```
In [5]: autre_ensemble_vide = set([])
        print(type(autre_ensemble_vide))

<class 'set'>
```

Un élément dans un ensemble doit être globalement immuable

On a vu précédemment que les clés dans un dictionnaire doivent être globalement immuables. Pour exactement les mêmes raisons, les éléments d'un ensemble doivent aussi être globalement immuables :

```
# on ne peut pas insérer un tuple qui contient une liste
>>> ensemble = {(1, 2, [3, 4])}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Le type `set` étant lui-même mutable, on ne peut pas créer un ensemble d'ensembles :

```
>>> ensemble = {{1, 2}}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

Et c'est une des raisons d'être du type `frozenset`.

Création - la fonction `frozenset`

Un `frozenset` est un ensemble qu'on ne peut pas modifier, et qui donc peut servir de clé dans un dictionnaire, ou être inclus dans un autre ensemble (mutable ou pas).

Il n'existe pas de raccourci syntaxique comme les `{}` pour créer un ensemble immuable, qui doit être créé avec la fonction `frozenset`. Toutes les opérations documentées dans ce notebook, et qui n'ont pas besoin de modifier l'ensemble, sont disponibles sur un `frozenset`.

Parmi les fonctions exclues sur un `frozenset`, on peut citer : `update`, `pop`, `clear`, `remove` ou `discard`.

Opérations simples

```
In [6]: # pour rappel
        heteroclite

Out[6]: {(1, 2, 3), 12, 'marc', 'pierre'}
```

Test d'appartenance

```
In [7]: (1, 2, 3) in heteroclite

Out[7]: True
```

Cardinal

```
In [8]: len(heteroclite)

Out[8]: 4
```

Manipulations

```
In [9]: ensemble = {1, 2, 1}
        ensemble

Out[9]: {1, 2}

In [10]: # pour nettoyer
         ensemble.clear()
         ensemble

Out[10]: set()

In [11]: # ajouter un element
         ensemble.add(1)
         ensemble

Out[11]: {1}

In [12]: # ajouter tous les elements d'un autre *ensemble*
         ensemble.update({2, (1, 2, 3), (1, 3, 5)})
         ensemble

Out[12]: {(1, 2, 3), (1, 3, 5), 1, 2}

In [13]: # enlever un element avec discard
         ensemble.discard((1, 3, 5))
         ensemble

Out[13]: {(1, 2, 3), 1, 2}

In [14]: # discard fonctionne même si l'élément n'est pas présent
         ensemble.discard('foo')
         ensemble

Out[14]: {(1, 2, 3), 1, 2}

In [15]: # enlever un élément avec remove
         ensemble.remove((1, 2, 3))
         ensemble

Out[15]: {1, 2}

In [16]: # contrairement à discard, l'élément doit être présent,
         # sinon il y a une exception
         try:
             ensemble.remove('foo')
         except KeyError as e:
             print("remove a levé l'exception", e)

remove a levé l'exception 'foo'
```

La capture d'exception avec `try` et `except` sert à capturer une erreur d'exécution du programme (que l'on appelle exception) pour continuer le programme. Le but de cet exemple est simplement de montrer (d'une manière plus élégante que de voir simplement le programme planter avec une exception non capturée) que l'expression `ensemble.remove('foo')` génère une exception. Si ce concept vous paraît obscur, pas d'inquiétude, nous l'aborderons cette semaine et nous y reviendrons en détail en semaine 6.

```
In [17]: # pop() ressemble à la méthode éponyme sur les listes
         # sauf qu'il n'y a pas d'ordre dans un ensemble
         while ensemble:
             element = ensemble.pop()
             print("element", element)
         print("et bien sûr maintenant l'ensemble est vide", ensemble)

element 1
element 2
et bien sûr maintenant l'ensemble est vide set()
```

Opérations classiques sur les ensembles

Donnons-nous deux ensembles simples :

```
In [18]: A2 = set([0, 2, 4, 6])
         print('A2', A2)
         A3 = set([0, 6, 3])
         print('A3', A3)

A2 {0, 2, 4, 6}
A3 {0, 3, 6}
```

N'oubliez pas que les ensembles, comme les dictionnaires, ne sont **pas ordonnés**.

Remarques :

— les notations des opérateurs sur les ensembles rappellent les opérateurs "bit-à-bit" sur les entiers ;

— ces opérateurs sont également disponibles sous la forme de méthodes.

Union

```
In [19]: A2 | A3

Out[19]: {0, 2, 3, 4, 6}
```

Intersection

```
In [20]: A2 & A3

Out[20]: {0, 6}
```

Différence

```
In [21]: A2 - A3
```

```
Out[21]: {2, 4}
```

```
In [22]: A3 - A2
```

```
Out[22]: {3}
```

Différence symétrique

On rappelle que $A \Delta B = (A - B) \cup (B - A)$

```
In [23]: A2 ^ A3
```

```
Out[23]: {2, 3, 4}
```

Comparaisons

Ici encore on se donne deux ensembles :

```
In [24]: superset = {0, 1, 2, 3}
         print('superset', superset)
         subset =  {1, 3}
         print('subset', subset)
```

```
superset {0, 1, 2, 3}
subset {1, 3}
```

Égalité

```
In [25]: heteroclite == heteroclite2
```

```
Out[25]: True
```

Inclusion

```
In [26]: subset <= superset
```

```
Out[26]: True
```

```
In [27]: subset < superset
```

```
Out[27]: True
```

```
In [28]: heteroclite < heteroclite2
```

```
Out[28]: False
```

Ensembles disjoints

```
In [29]: heteroclite.isdisjoint(A3)
```

```
Out[29]: True
```

Pour en savoir plus

Reportez vous à la section sur les ensembles dans la documentation Python.

3.16 Ensembles

3.16.1 Exercice - niveau basique

```
In [ ]: # charger l'exercice
        from corrections.exo_read_set import exo_read_set
```

On se propose d'écrire une fonction `read_set` qui construit un ensemble à partir du contenu d'un fichier. Voici par exemple un fichier d'entrée :

```
In [ ]: !cat data/setref1.txt
```

`read_set` va prendre en argument un nom de fichier (vous pouvez supposer qu'il existe), enlever les espaces éventuelles au début et à la fin de chaque ligne, et construire un ensemble de toutes les lignes ; par exemple :

```
In [ ]: exo_read_set.example()
```

```
In [ ]: # écrivez votre code ici
        def read_set(filename):
            "votre code"
```

```
In [ ]: # vérifiez votre code ici
        exo_read_set.correction(read_set)
```

3.16.2 Deuxième partie - niveau basique

```
In [ ]: # la définition de l'exercice
        from corrections.exo_read_set import exo_search_in_set
```

Ceci étant acquis, on veut écrire une deuxième fonction `search_in_set` qui prend en argument deux fichiers :

— `filename_reference` est le nom d'un fichier contenant des mots de référence ;
— `filename` est le nom d'un fichier contenant des mots, dont on veut savoir s'ils sont ou non dans les références.

Pour cela `search_in_set` doit retourner une liste, contenant pour chaque ligne du fichier `filename` un tuple avec :

— la ligne (sans les espaces de début et de fin, ni la fin de ligne) ;
— un booléen qui indique si ce mot est présent dans les références ou pas.

Par exemple :

```
In [ ]: !cat data/setref1.txt
```

```
In [ ]: !cat data/setsample1.txt
```

```
In [ ]: exo_search_in_set.example()
```

```
In [ ]: # à vous
        def search_in_set(filename_reference, filename):
            "votre code"
```

```
In [ ]: # vérifiez
        exo_search_in_set.correction(search_in_set)
```

3.17 Exercice sur les ensembles

3.17.1 Exercice - niveau intermédiaire

```
In [ ]: # chargement de l'exercice
        from corrections.exo_marine_set import exo_diff
```

Les données

Nous reprenons le même genre de données marines en provenance de MarineTraffic que nous avons vues dans l'exercice précédent.

```
In [ ]: from corrections.exo_marine_set import abbreviated, extended
```

Rappels sur les formats

```
étendu: [id, latitude, longitude, date_heure, nom_bateau, code_pays...]
abrégé: [id, latitude, longitude, date_heure]
```

```
In [ ]: print(extended[0])
```

```
In [ ]: print(abbreviated[0])
```

But de l'exercice

```
In [ ]: # chargement de l'exercice
        from corrections.exo_marine_set import exo_diff
```

Notez bien une différence importante avec l'exercice précédent : cette fois **il n'y a plus correspondance** entre les bateaux rapportés dans les données étendues et abrégées.

Le but de l'exercice est précisément d'étudier la différence, et pour cela on vous demande d'écrire une fonction

```
diff(extended, abbreviated)
```

qui retourne un tuple à trois éléments :

— l'ensemble (`set`) des **noms** des bateaux présents dans `extended` mais pas dans `abbreviated`;

— l'ensemble des **noms** des bateaux présents dans `extended` et dans `abbreviated`;

— l'ensemble des **id** des bateaux présents dans `abbreviated` mais pas dans `extended` (par construction, les données ne nous permettent pas d'obtenir les noms de ces bateaux).

```
In [ ]: # le résultat attendu
        result = exo_diff.resultat(extended, abbreviated)

        # combien de bateaux sont concernés
        def show_result(extended, abbreviated, result):
            """
            Affiche divers décomptes sur les arguments
            en entrée et en sortie de diff
            """
            print(10*'-', "Les entrées")
```

```
            print(f"Dans extended: {len(extended)} entrées")
            print(f"Dans abbreviated: {len(abbreviated)} entrées")
            print(10*'-', "Le résultat du diff")
            extended_only, both, abbreviated_only = result
            print(f"Dans extended mais pas dans abbreviated {len(extended_only)}")
            print(f"Dans les deux {len(both)}")
            print(f"Dans abbreviated mais pas dans extended {len(abbreviated_only)}")

        show_result(extended, abbreviated, result)
```

Votre code

```
In [ ]: def diff(extended, abbreviated):
            "<votre_code>"
```

Validation

```
In [ ]: exo_diff.correction(diff, extended, abbreviated)
```

Des fichiers de données plus réalistes

Comme pour l'exercice précédent, les données fournies ici sont très simplistes ; vous pouvez, si vous le voulez, essayer votre code avec des données (un peu) plus réalistes en chargeant des fichiers de données plus complets :

— data/marine-e2-ext.json

— data/marine-e2-abb.json

Ce qui donnerait en Python :

```
In [ ]: # load data from files
        import json

        with open("data/marine-e2-ext.json", encoding="utf-8") as feed:
            extended_full = json.load(feed)

        with open("data/marine-e2-abb.json", encoding="utf-8") as feed:
            abbreviated_full = json.load(feed)
```

```
In [ ]: # le résultat de votre fonction sur des données plus vastes
        # attention, show_result fait des hypothèses sur le type de votre résultat
        # aussi si vous essayez d'exécuter ceci avec comme fonction diff
        # la version vide qui est dans le notebook original
        # cela peut provoquer une exception
        diff_full = diff(extended_full, abbreviated_full)
        show_result(extended_full, abbreviated_full, diff_full)
```

Je signale enfin à propos de ces données plus complètes que :

— on a supprimé les entrées correspondant à des bateaux différents mais de même nom ; cette situation peut arriver dans la réalité (c'est pourquoi d'ailleurs les bateaux ont un *id*) mais ici ce n'est pas le cas ;

— il se peut par contre qu'un même bateau fasse l'objet de plusieurs mesures dans `extended` et/ou dans `abbreviated`.

3.18 `try` ... `else` ... `finally`

3.18.1 Complément - niveau intermédiaire

L'instruction `try` est généralement assortie d'une une ou plusieurs clauses `except`, comme on l'a vu dans la vidéo.

Sachez que l'on peut aussi utiliser - après toutes les clauses `except` :

— une clause `else`, qui va être exécutée si aucune exception n'est attrapée ;

— t/ou une clause `finally` qui sera alors exécutée quoi qu'il arrive.

Voyons cela sur des exemples.

`finally`

C'est sans doute `finally` qui est la plus utile de ces deux clauses, car elle permet de faire un nettoyage **dans tous les cas de figure** - de ce point de vue, cela rappelle un peu les *context managers*.

Et par exemple, comme avec les *context managers*, une fonction peut faire des choses même après un `return`.

```
In [1]: # une fonction qui fait des choses après un return
        def return_with_finally(number):
            try:
                return 1/number
            except ZeroDivisionError as e:
                print(f"OOPS, {type(e)}, {e}")
                return("zero-divide")
            finally:
                print("on passe ici même si on a vu un return")

In [2]: # sans exception
        return_with_finally(1)
```

```
on passe ici même si on a vu un return
```

```
Out[2]: 1.0
```

```
In [3]: # avec exception
        return_with_finally(0)
```

```
OOPS, <class 'ZeroDivisionError'>, division by zero
on passe ici même si on a vu un return
```

```
Out[3]: 'zero-divide'
```

`else`

La logique ici est assez similaire, sauf que le code du `else` n'est exécuté que dans le cas où aucune exception n'est attrapée.

En première approximation, on pourrait penser que c'est équivalent de mettre du code dans la clause `else` ou à la fin de la clause `try`. En fait il y a une différence subtile :

> *The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.*

Dit autrement, si le code dans la clause `else` lève une exception, celle-ci ne **sera pas attrapée** par le `try` courant, et sera donc propagée.

Voici un exemple rapide, en pratique on rencontre assez peu souvent une clause `else` dans un `try` :

```
In [4]: # pour montrer la clause else dans un usage banal
        def function_with_else(number):
            try:
                x = 1/number
            except ZeroDivisionError as e:
                print(f"OOPS, {type(e)}, {e}")
            else:
                print("on passe ici seulement avec un nombre non nul")
            return 'something else'
```

```
In [5]: # sans exception
        function_with_else(1)
```

```
on passe ici seulement avec un nombre non nul
```

```
Out[5]: 'something else'
```

```
In [6]: # avec exception
        function_with_else(0)
```

```
OOPS, <class 'ZeroDivisionError'>, division by zero
```

```
Out[6]: 'something else'
```

Remarquez que `else` ne présente pas cette particularité de "traverser" le `return`, que l'on a vue avec `finally` :

```
In [7]: # la clause else ne traverse pas les return
        def return_with_else(number):
            try:
                return 1/number
            except ZeroDivisionError as e:
                print(f"OOPS, {type(e)}, {e}")
                return("zero-divide")
            else:
                print("on ne passe jamais ici à cause des return")
```

```
In [8]: # sans exception
        return_with_else(1)

Out[8]: 1.0

In [9]: # avec exception
        return_with_else(0)

OOPS, <class 'ZeroDivisionError'>, division by zero


Out[9]: 'zero-divide'
```

Pour en savoir plus

Voyez le tutorial sur les exceptions dans la documentation officielle.

3.19 L'opérateur `is`

3.19.1 Complément - niveau basique

```
In [ ]: %load_ext ipythontutor
```

Les opérateurs `is` et `==`

— nous avons déjà parlé de l'opérateur `==` qui **compare la valeur** de deux objets ;
— python fournit aussi un opérateur `is` qui permet de savoir si deux valeurs correspondent **au même objet** en mémoire.

Nous allons illustrer la différence entre ces deux opérateurs.

Scénario 1

```
In [1]: # deux listes identiques
        a = [1, 2]
        b = [1, 2]

        # les deux objets se ressemblent
        print('==', a == b)
```

```
== True
```

```
In [2]: # mais ce ne sont pas les mêmes objets
        print('is', a is b)
```

```
is False
```

Scénario 2

```
In [3]: # par contre ici il n'y a qu'une liste
        a = [1, 2]

        # et les deux variables
        # référencent le même objet
        b = a

        # non seulement les deux expressions se ressemblent
        print('==', a == b)
```

```
== True
```

```
In [4]: # mais elles désignent le même objet
        print('is', a is b)
```

```
is True
```

La même chose sous pythontutor

Scénario 1

```
In [ ]: %%ipythontutor curInstr=2
        a = [1, 2]
        b = [1, 2]
```

Scénario 2

```
In [ ]: %%ipythontutor curInstr=1
        # équivalent à la forme ci-dessus
        # a = [1, 2]
        # b = a
        a = b = [1, 2]
```

Utilisez `is` plutôt que == lorsque c'est possible

La pratique usuelle est d'utiliser `is` lorsqu'on compare avec un objet qui est un singleton, comme typiquement `None`.

Par exemple on préfèrera écrire :

```
In [5]: undef = None

        if undef is None:
            print('indéfini')
```

```
indéfini
```

Plutôt que :

```
In [6]: if undef == None:
            print('indéfini')
```

```
indéfini
```

Qui se comporte de la même manière (à nouveau, parce qu'on compare avec `None`), mais est légèrement moins lisible, et franchement moins pythonique. :)

Notez aussi et surtout que `is` est **plus efficace** que `==`. En effet `is` peut être évalué en temps constant, puisqu'il s'agit essentiellement de comparer les deux adresses. Alors que pour `==` il peut s'agir de parcourir toute une structure de données possiblement très complexe.

3.19.2 Complément - niveau intermédiaire

La fonction `id`

Pour bien comprendre le fonctionnement de `is` nous allons voir la fonction `id` qui retourne un identificateur unique pour chaque objet ; un modèle mental acceptable est celui d'adresse mémoire.

```
In [7]: id(True)
```

```
Out[7]: 1924796304
```

Comme vous vous en doutez, l'opérateur `is` peut être décrit formellement à partir de `id` comme ceci :

`(a is b)` $\Longleftrightarrow$ `(id(a) == id(b))`

Certains types de base sont des singletons

Un singleton est un objet qui n'existe qu'en un seul exemplaire dans la mémoire. Un usage classique des singletons en Python est de minimiser le nombre d'objets immuables en mémoire. Voyons ce que cela nous donne avec des entiers :

```
In [8]: a = 3
        b = 3
        print('a', id(a), 'b', id(b))
```

```
a 1924978512 b 1924978512
```

Tiens, c'est curieux, nous avons ici deux objets, que l'on pourrait penser différents, mais en fait ce sont les mêmes ; `a` et `b` désignent **le même objet** python, et on a :

```
In [9]: a is b
```

```
Out[9]: True
```

Il se trouve que, dans le cas des petits entiers, python réalise une optimisation de l'utilisation de la mémoire. Quel que soit le nombre de variables dont la valeur est `3`, un seul objet correspondant à l'entier `3` est alloué et créé, pour éviter d'engorger la mémoire. On dit que l'entier `3` est implémenté comme un singleton ; nous reverrons ceci en exercice.

On trouve cette optimisation avec quelques autres objets python, comme par exemple :

```
In [10]: a = ""
         b = ""
         a is b
```

```
Out[10]: True
```

Ou encore, plus surprenant :

```
In [11]: a = "foo"
         b = "foo"
         a is b
```

```
Out[11]: True
```

Conclusion cette optimisation ne touche aucun type mutable (heureusement) ; pour les types immuables, il n'est pas extrêmement important de savoir en détail quels objets sont implémentés de la sorte.

Ce qui est par contre extrêmement important est de comprendre la différence entre `is` et `==`, et de les utiliser à bon escient au risque d'écrire du code fragile.

Pour en savoir plus

Aux étudiants de niveau avancé, nous recommandons la lecture de la section “Objects, values and types” dans la documentation Python :

https://docs.python.org/3/reference/datamodel.html#objects-values-and-types

qui aborde également la notion de “garbage collection”, que nous n’aurons pas le temps d’approfondir dans ce MOOC.

3.20 Listes infinies & références circulaires

3.20.1 Complément - niveau intermédiaire

```
In [ ]: %load_ext ipythontutor
```

Nous allons maintenant construire un objet un peu abscons. Cet exemple précis n'a aucune utilité pratique, mais permet de bien comprendre la logique du langage.

Construisons une liste à un seul élément, peu importe quoi :

```
In [1]: infini_1 = [None]
```

À présent nous allons remplacer le premier et seul élément de la liste par… la liste elle-même :

```
In [2]: infini_1[0] = infini_1
        print(infini_1)
```

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

Pour essayer de décrire l'objet liste ainsi obtenu, on pourrait dire qu'il s'agit d'une liste de taille 1 et de profondeur infinie, une sorte de fil infini en quelque sorte.

Naturellement, l'objet obtenu est difficile à imprimer de manière convaincante. Pour faire en sorte que cet objet soit tout de même imprimable, et éviter une boucle infinie, python utilise l'ellipse `...` pour indiquer ce qu'on appelle une référence circulaire. Si on n'y prenait pas garde en effet, il faudrait écrire `[[[[ etc. ]]]]` avec une infinité de crochets.

Voici la même séquence exécutée sous http://pythontutor.com ; il s'agit d'un site très utile pour comprendre comment python implémente les objets, les références et les partages.

Cliquez sur le bouton `Forward` pour avancer dans l'exécution de la séquence. À la fin de la séquence vous verrez - ce n'est pas forcément clair - la seule cellule de la liste à se référencer elle-même :

```
In [ ]: %%ipythontutor height=230
        infini_1 = [None]
        infini_1[0] = infini_1
```

Toutes les fonctions de python ne sont pas aussi intelligentes que `print`. Bien qu'on puisse comparer cette liste avec elle-même :

```
In [3]: infini_1 == infini_1
```

```
Out[3]: True
```

il n'en est pas de même si on la compare avec un objet analogue mais pas identique :

```
In [4]: infini_2 = [0]
        infini_2[0] = infini_2
        print(infini_2)
        infini_1 == infini_2
```

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

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)

<ipython-input-4-3c044c24fccb> in <module>()
  2 infini_2[0] = infini_2
  3 print(infini_2)
----> 4 infini_1 == infini_2


RecursionError: maximum recursion depth exceeded in comparison
```

Généralisation aux références circulaires

On obtient un phénomène équivalent dès lors qu'un élément contenu dans un objet fait référence à l'objet lui-même. Voici par exemple comment on peut construire un dictionnaire qui contient une référence circulaire :

```
In [5]: collection_de_points = [
            {'x': 10,'y': 20},
            {'x': 30,'y': 50},
            # imaginez plein de points
        ]

        # on rajoute dans chaque dictionnaire une clé 'points'
        # qui référence la collection complète
        for point in collection_de_points:
            point['points'] = collection_de_points

        # la structure possède maintenant des références circulaires
        print(collection_de_points)
```

```
[{'x': 10, 'y': 20, 'points': [...]}, {'x': 30, 'y': 50, 'points': [...]}]
```

On voit à nouveau réapparaître les ellipses, qui indiquent que pour chaque point, le nouveau champ `points` est un objet qui a déjà été imprimé.

Cette technique est cette fois très utile et très utilisée dans la pratique, dès lors qu'on a besoin de naviguer de manière arbitraire dans une structure de données compliquée. Dans cet exemple, pas très réaliste naturellement, on pourrait à présent accéder depuis un point à tous les autres points de la collection dont il fait partie.

À nouveau il peut être intéressant de voir le comportement de cet exemple avec http://pythontutor.com pour bien comprendre ce qui se passe, si cela ne vous semble pas clair à première vue :

In [ ]:
```
%%ipythontutor curInstr=7
points = [
    {'x': 10,'y': 20},
    {'x': 30,'y': 50},
]

for point in points:
    point['points'] = points
```

3.21 Les différentes copies

```
In [ ]: %load_ext ipythontutor
```

3.21.1 Complément - niveau basique

Deux types de copie

Pour résumer les deux grands types de copie que l'on a vus dans la vidéo :

— La *shallow copy* - de l'anglais *shallow* qui signifie superficiel ;

— La *deep copy* - de *deep* qui signifie profond.

Le module `copy`

Pour réaliser une copie, la méthode la plus simple, en ceci qu'elle fonctionne avec tous les types de manière identique, consiste à utiliser le module standard `copy`, et notamment :

— `copy.copy` pour une copie superficielle ;

— `copy.deepcopy` pour une copie en profondeur.

```
In [1]: import copy
        #help(copy.copy)
        #help(copy.deepcopy)
```

Un exemple

Nous allons voir le résultat des deux formes de copie sur un même sujet de départ.

La copie superficielle / *shallow* copie / `copy.copy`

N'oubliez pas de cliquer le bouton `Forward` dans la fenêtre pythontutor :

```
In [ ]: %%ipythontutor height=410 curInstr=6
        import copy
        # On se donne un objet de départ
        source = [
            [1, 2, 3],  # une liste
            {1, 2, 3},  # un ensemble
            (1, 2, 3),  # un tuple
            '123',       # un string
            123,         # un entier
        ]
        # une copie simple renvoie ceci
        shallow_copy = copy.copy(source)
```

Vous remarquez que :

— la source et la copie partagent tous leurs (sous-)éléments, et notamment la liste `source[0]` et l'ensemble `source[1]` ;

— ainsi, après cette copie, on peut modifier l'un de ces deux objets (la liste ou l'ensemble), et ainsi modifier la source **et** la copie.

On rappelle aussi que, la source étant une liste, on aurait pu aussi bien faire la copie superficielle avec

```
shallow2 = source[:]
```

La copie profonde / *deep* copie / `copy.deepcopy` Sur le même objet de départ, voici ce que fait la copie profonde :

```
In [ ]: %%ipythontutor height=410 curInstr=6
        import copy
        # On se donne un objet de départ
        source = [
            [1, 2, 3],  # une liste
            {1, 2, 3},  # un ensemble
            (1, 2, 3),  # un tuple
            '123',       # un string
            123,         # un entier
        ]
        # une copie profonde renvoie ceci
        deep_copy = copy.deepcopy(source)
```

Ici, il faut remarquer que :

— les deux objets mutables accessibles via `source`, c'est-à-dire **la liste** `source[0]` et **l'ensemble** `source[1]`, ont été tous deux dupliqués ;

— **le tuple** correspondant à `source[2]` n'est **pas dupliqué**, mais comme il n'est **pas mutable** on ne peut pas modifier la copie au travers de la source ;

— de manière générale, on a la bonne propriété que la source et sa copie ne partagent rien qui soit modifiable ;

— et donc on ne peut pas modifier l'un au travers de l'autre.

On retrouve donc à nouveau l'optimisation qui est mise en place dans python pour implémenter les types immuables comme des singletons lorsque c'est possible. Cela a été vu en détail dans le complément consacré à l'opérateur `is`.

3.21.2 Complément - niveau intermédiaire

```
In [2]: # on répète car le code précédent a seulement été exposé à pythontutor
        import copy
        source = [
            [1, 2, 3],  # une liste
            {1, 2, 3},  # un ensemble
            (1, 2, 3),  # un tuple
            '123',       # un string
            123,         # un entier
        ]
        shallow_copy = copy.copy(source)
        deep_copy = copy.deepcopy(source)
```

Objets *égaux* au sens logique

Bien sûr ces trois objets se ressemblent si on fait une comparaison *logique* avec `==` :

```
In [3]: print('source == shallow_copy:', source == shallow_copy)
        print('source == deep_copy:', source == deep_copy)
```

```
source == shallow_copy: True
source == deep_copy: True
```

Inspectons les objets de premier niveau

Mais par contre si on compare **l'identité** des objets de premier niveau, on voit que `source` et `shallow_copy` partagent leurs objets :

```
In [4]: # voir la cellule ci-dessous si ceci vous parait peu clair
        for i, (source_item, copy_item) in enumerate(zip(source, shallow_copy)):
            compare = source_item is copy_item
            print(f"source[{i}] is shallow_copy[{i}] -> {compare}")
```

```
source[0] is shallow_copy[0] -> True
source[1] is shallow_copy[1] -> True
source[2] is shallow_copy[2] -> True
source[3] is shallow_copy[3] -> True
source[4] is shallow_copy[4] -> True
```

```
In [5]: # rappel au sujet de zip et enumerate
        # la cellule ci-dessous est essentiellement équivalente à
        for i in range(len(source)):
            compare = source[i] is shallow_copy[i]
            print(f"source[{i}] is shallow_copy[{i}] -> {compare}")
```

```
source[0] is shallow_copy[0] -> True
source[1] is shallow_copy[1] -> True
source[2] is shallow_copy[2] -> True
source[3] is shallow_copy[3] -> True
source[4] is shallow_copy[4] -> True
```

Alors que naturellement ce **n'est pas le cas** avec la copie en profondeur :

```
In [6]: for i, (source_item, deep_item) in enumerate(zip(source, deep_copy)):
            compare = source_item is deep_item
            print(f"source[{i}] is deep_copy[{i}] -> {compare}")
```

```
source[0] is deep_copy[0] -> False
source[1] is deep_copy[1] -> False
source[2] is deep_copy[2] -> True
source[3] is deep_copy[3] -> True
source[4] is deep_copy[4] -> True
```

On retrouve ici ce qu'on avait déjà remarqué sous pythontutor, à savoir que les trois derniers objets - immuables - n'ont pas été dupliqués comme on aurait pu s'y attendre.

On modifie la source

Il doit être clair à présent que, précisément parce que `deep_copy` est une copie en profondeur, on peut modifier `source` sans impacter du tout `deep_copy`.

S'agissant de `shallow_copy`, par contre, seuls les éléments de premier niveau ont été copiés. Aussi si on fait une modification par exemple **à l'intérieur** de la liste qui est le premier fils de `source`, cela sera **répercuté** dans `shallow_copy` :

```
In [7]: print("avant, source      ", source)
        print("avant, shallow_copy", shallow_copy)
        source[0].append(4)
        print("après, source      ", source)
        print("après, shallow_copy", shallow_copy)

avant, source       [[1, 2, 3], {1, 2, 3}, (1, 2, 3), '123', 123]
avant, shallow_copy [[1, 2, 3], {1, 2, 3}, (1, 2, 3), '123', 123]
après, source       [[1, 2, 3, 4], {1, 2, 3}, (1, 2, 3), '123', 123]
après, shallow_copy [[1, 2, 3, 4], {1, 2, 3}, (1, 2, 3), '123', 123]
```

Si par contre on remplace complètement un élément de premier niveau dans la source, cela ne sera pas répercuté dans la copie superficielle :

```
In [8]: print("avant, source      ", source)
        print("avant, shallow_copy", shallow_copy)
        source[0] = 'remplacement'
        print("après, source      ", source)
        print("après, shallow_copy", shallow_copy)

avant, source       [[1, 2, 3, 4], {1, 2, 3}, (1, 2, 3), '123', 123]
avant, shallow_copy [[1, 2, 3, 4], {1, 2, 3}, (1, 2, 3), '123', 123]
après, source       ['remplacement', {1, 2, 3}, (1, 2, 3), '123', 123]
après, shallow_copy [[1, 2, 3, 4], {1, 2, 3}, (1, 2, 3), '123', 123]
```

Copie et circularité

Le module `copy` est capable de copier - même en profondeur - des objets contenant des références circulaires.

```
In [9]: l = [None]
        l[0] = l
        l

Out[9]: [[...]]

In [10]: copy.copy(l)

Out[10]: [[[...]]]

In [11]: copy.deepcopy(l)

Out[11]: [[...]]
```

Pour en savoir plus

On peut se reporter à la section sur le module `copy` dans la documentation Python.

3.22 L'instruction `del`

3.22.1 Complément - niveau basique

Voici un récapitulatif sur l'instruction `del` selon le contexte dans lequel elle est utilisée.

Sur une variable

On peut annuler la définition d'une variable, avec `del`.

Pour l'illustrer, nous utilisons un bloc `try ... except ...` pour attraper le cas échéant l'exception `NameError`, qui est produite lorsqu'on référence une variable qui n'est pas définie.

```
In [1]: # la variable a n'est pas définie
        try:
            print('a=', a)
        except NameError as e:
            print("a n'est pas définie")
```

```
a n'est pas définie
```

```
In [2]: # on la définit
        a = 10

        # aucun souci ici, l'exception n'est pas levée
        try:
            print('a=', a)
        except NameError as e:
            print("a n'est pas définie")
```

```
a= 10
```

```
In [3]: # maintenant on peut effacer la variable
        del a

        # c'est comme si on ne l'avait pas définie
        # dans la cellule précédente
        try:
            print('a=', a)
        except NameError as e:
            print("a n'est pas définie")
```

```
a n'est pas définie
```

Sur une liste

On peut enlever d'une liste les éléments qui correspondent à une *slice* :

```
In [4]: # on se donne une liste
        l = list(range(12))
        print(l)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

```
In [5]: # on considère une slice dans cette liste
        print('slice=', l[2:10:3])

        # voyons ce que ça donne si on efface cette slice
        del l[2:10:3]
        print("après del", l)
```

```
slice= [2, 5, 8]
après del [0, 1, 3, 4, 6, 7, 9, 10, 11]
```

Sur un dictionnaire

Avec `del` on peut enlever une clé, et donc la valeur correspondante, d'un dictionnaire :

```
In [6]: # partons d'un dictionaire simple
        d = dict(foo='bar', spam='eggs', a='b')
        print(d)
```

```
{'foo': 'bar', 'spam': 'eggs', 'a': 'b'}
```

```
In [7]: # on peut enlever une clé avec del
        del d['a']
        print(d)
```

```
{'foo': 'bar', 'spam': 'eggs'}
```

On peut passer plusieurs arguments à `del`

```
In [8]: # Voyons où en sont nos données
        print('l', l)
        print('d', d)
```

```
l [0, 1, 3, 4, 6, 7, 9, 10, 11]
d {'foo': 'bar', 'spam': 'eggs'}
```

```
In [9]: # on peut invoquer 'del' avec plusieurs expressions
        # séparées par une virgule
        del l[3:], d['spam']

        print('l', l)
        print('d', d)
```

```
l [0, 1, 3]
d {'foo': 'bar'}
```

Pour en savoir plus

La page sur l'instruction `del` dans la documentation Python.

3.23 Affectation simultanée

3.23.1 Complément - niveau basique

Nous avons déjà parlé de l'affectation par *sequence unpacking* (en Semaine 3, séquence "Les tuples"), qui consiste à affecter à plusieurs variables des "morceaux" d'un objet, comme dans :

```
In [1]: x, y = ['spam', 'egg']
```

Dans ce complément nous allons voir une autre forme de l'affectation, qui consiste à affecter **le même objet** à plusieurs variables. Commençons par un exemple simple :

```
In [2]: a = b = 1
        print('a', a, 'b', b)
```

```
a 1 b 1
```

La raison pour laquelle nous abordons cette construction maintenant est qu'elle a une forte relation avec les références partagées ; pour bien le voir, nous allons utiliser une valeur mutable comme valeur à affecter :

```
In [3]: # on affecte a et b au même objet liste vide
        a = b = []
```

Dès lors nous sommes dans le cas typique d'une référence partagée ; une modification de `a` va se répercuter sur `b` puisque ces deux variables désignent **le même objet** :

```
In [4]: a.append(1)
        print('a', a, 'b', b)
```

```
a [1] b [1]
```

Ceci est à mettre en contraste avec plusieurs affectations séparées :

```
In [5]: # si on utilise deux affectations différentes
        a = []
        b = []

        # alors on peut changer a sans changer b
        a.append(1)
        print('a', a, 'b', b)
```

```
a [1] b []
```

On voit que dans ce cas chaque affectation crée une liste vide différente, et les deux variables ne partagent plus de donnée.

D'une manière générale, utiliser l'affectation simultanée vers un objet mutable crée mécaniquement des **références partagées**, aussi vérifiez bien dans ce cas que c'est votre intention.

3.24 Les instructions += et autres revisitées

3.24.1 Complément - niveau intermédiaire

Nous avons vu en première semaine (Séquence "Les types numériques") une première introduction à l'instruction += et ses dérivées comme *=, **=, etc.

Ces constructions ont une définition à géométrie variable

En C quand on utilise += (ou encore ++) on modifie la mémoire en place - historiquement, cet opérateur permettait au programmeur d'aider à l'optimisation du code pour utiliser les instructions assembleur idoines.

Ces constructions en Python s'inspirent clairement de C, aussi dans l'esprit ces constructions devraient fonctionner en **modifiant** l'objet référencé par la variable.

Mais les types numériques en Python ne sont **pas mutables**, alors que les listes le sont. Du coup le comportement de += est **différent** selon qu'on l'utilise sur un nombre ou sur une liste, ou plus généralement selon qu'on l'invoque sur un type mutable ou non. Voyons cela sur des exemples très simples :

```
In [1]: # Premier exemple avec un entier

        # on commence avec une référence partagée
        a = b = 3
        a is b
```

```
Out[1]: True
```

```
In [2]: # on utilise += sur une des deux variables
        a += 1

        # ceci n'a pas modifié b
        # c'est normal, l'entier n'est pas mutable

        print(a)
        print(b)
        print(a is b)
```

```
4
3
False
```

```
In [3]: # Deuxième exemple, cette fois avec une liste

        # la même référence partagée
        a = b = []
        a is b
```

```
Out[3]: True
```

```
In [4]: # pareil, on fait += sur une des variables
        a += [1]

        # cette fois on a modifié a et b
        # car += a pu modifier la liste en place
        print(a)
        print(b)
        print(a is b)
```

```
[1]
[1]
True
```

Vous voyez donc que la sémantique de += (c'est bien entendu le cas pour toutes les autres formes d'instructions qui combinent l'affectation avec un opérateur) **est différente** suivant que l'objet référencé par le terme de gauche est **mutable ou immuable**.

Pour cette raison, c'est là une opinion personnelle, cette famille d'instructions n'est pas le trait le plus réussi dans le langage, et je ne recommande pas de l'utiliser.

Précision sur la définition de +=

Nous avions dit en première semaine, et en première approximation, que :

```
x += y
```

était équivalent à :

```
x = x + y
```

Au vu de ce qui précède, on voit que ce n'est **pas tout à fait exact**, puisque :

```
In [5]: # si on fait x += y sur une liste
        # on fait un effet de bord sur la liste
        # comme on vient de le voir

        a = []
        print("avant", id(a))
        a += [1]
        print("après", id(a))
```

```
avant 82877176
après 82877176
```

```
In [6]: # alors que si on fait x = x + y sur une liste
        # on crée un nouvel objet liste

        a = []
        print("avant", id(a))
        a = a + [1]
        print("après", id(a))
```

```
avant 82981752
après 82877496
```

Vous voyez donc que vis-à-vis des références partagées, ces deux façons de faire mènent à un résultat différent.

3.25 Classe

3.25.1 Exercice - niveau basique

```
In [ ]: # charger l'exercice
        from corrections.cls_fifo import exo_fifo
```

On veut implémenter une classe pour manipuler une queue d'événements. La logique de cette classe est que :

— on la crée sans argument ;

— on peut toujours ajouter un élément avec la méthode `incoming` ;

— et tant que la queue contient des éléments on peut appeler la méthode `outgoing`, qui retourne et enlève un élément dans la queue.

Cette classe s'appelle `Fifo` pour *First In, First Out*, c'est-à-dire que les éléments retournés par `outgoing` le sont dans le même ordre où ils ont été ajoutés.

La méthode `outgoing` retourne `None` lorsqu'on l'appelle sur une pile vide.

```
In [ ]: # voici un exemple de scénario
        exo_fifo.example()
```

```
In [ ]: # vous pouvez définir votre classe ici
        class Fifo:
            def __init__(self):
                "votre code"
            def incoming(self, value):
                "votre code"
            def outgoing(self):
                "votre code"
```

```
In [ ]: # et la vérifier ici
        exo_fifo.correction(Fifo)
```