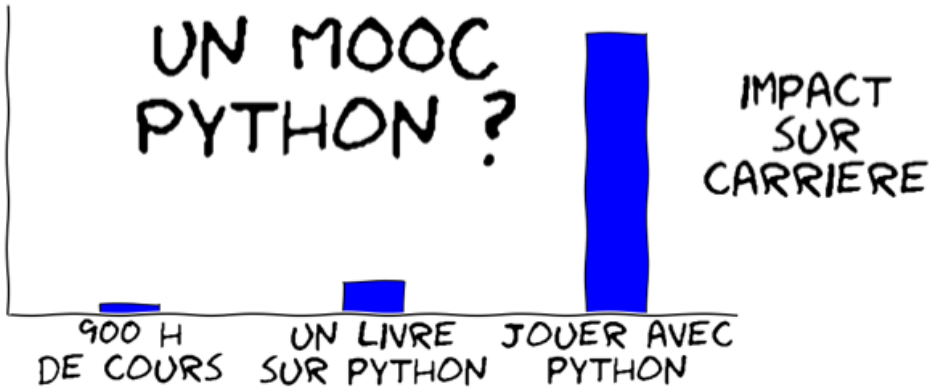
DES FONDAMENTAUX AU CONCEPTS AVANCÉS DU LANGAGE

Thierry PARMENTELAT

Arnaud LEGOUT



https ://www.fun-mooc.fr

TABLE DES MATIÈRES

SEMAINE 1

INTRODUCTION AU MOOC ET AUX OUTILS PYTHON

1.1 Attestations - modalités de délivrance

1.1.1 Cours 2017

Vous pouvez obtenir une attestation de suivi du cours. Pour cela il vous suffit de :

— répondre aux quiz **des semaines 1 à 6** ;
— obtenir une note moyenne supérieure ou égale à **60 %**.

1.1.2 Précisions

Pour être précis :

— il y a presque toujours un quiz par séquence, à partir de la semaine 1 - séquence 5 et jusqu'à la fin de la semaine 6 ; et chaque quiz contient en général de l'ordre de 3 ou 4 questions ;
— seul le tronc commun est concerné, il n'y a pas de quiz dans les modules optionnels des semaines 7 et suivantes ;
— il s'agit d'une attestation et non pas d'un certificat, nous ne délivrons pas de certificat ;
— pour plus de détails, voyez aussi cette page d'introduction.

1.2 Versions de python

Comme on l'indique dans la vidéo, la version de référence du MOOC est la version 3.6, c'est avec cette version que l'on a tourné les vidéos.

python-3.5

Si vous préférez utiliser python-3.5, la différence la plus visible pour vous apparaitra avec les *f-strings* :

```
In [ ]: age = 10

        # un exemple de f-string
        f"Jean a {age} ans"
```

Cette construction - que nous utilisons très fréquemment - n'a été introduite qu'en python-3.6, aussi si vous utilisez python-3.5 vous verrez ceci :

```
>>> age = 10
>>> f"Jean a {age} ans"
```

```
  File "<stdin>", line 1
    f"Jean a {age} ans"
                       ^
SyntaxError: invalid syntax
```

Dans ce cas vous devrez remplacer ce code avec la méthode `format` - que nous verrons en Semaine 2 lorsque nous verrons les chaines de caractères - et dans le cas présent il faudrait remplacer par ceci :

```
In [ ]: age = 10

        "Jean a {} ans".format(age)
```

Comme ces f-strings sont très présents dans le cours, il est recommandé d'utiliser au moins python-3.6.

1.3 python-3.4

La remarque vaut donc *a fortiori* pour python-3.4, qui en outre ne vous permettra pas de suivre la semaine 8 sur la programmation asynchrone, car les mots-clés `async` et `await` ont été introduits seulement dans python-3.5.

1.4 Installer la distribution standard python

1.4.1 Complément - niveau basique

Ce complément a pour but de vous donner quelques guides pour l'installation de la distribution standard python 3.

Notez bien qu'il ne s'agit ici que d'indications, il existe de nombreuses façons de procéder.

En cas de souci, commencez par chercher par vous-même sur google ou autre une solution à votre problème ; pensez également à utiliser le forum du cours.

Le point important est de **bien vérifier le numéro de version** de votre installation qui doit être **au moins 3.6**

1.5 Digression - coexistence de python2 et python3

Avant l'arrivée de la version 3 de python, les choses étaient simples, on exécutait un programme python avec une commande `python`. Depuis 2014-2015, maintenant que les deux versions de python coexistent, il est nécessaire d'adopter une convention qui permette d'installer les deux langages sous des noms qui sont non-ambigus.

C'est pourquoi actuellement, on trouve **le plus souvent** la convention suivante sous Linux et Mac OS X :

- `python3` est pour exécuter les programmes en python-3 ; du coup on trouve alors également les commandes comme `idle3` pour lancer IDLE, et par exemple `pip3` pour le gestionnaire de paquets (voir ci-dessous) ;
- `python2` est pour exécuter les programmes en python-2, avec typiquement `idle2` et `pip2` ;
- enfin selon les systèmes, la commande `python` tout court est un alias pour `python2` ou `python3`. De plus en plus souvent, par défaut `python` désigne `python3`.

à titre d'illustration, voici ce que j'obtiens sur mon mac :

```
$ python3 -V
Python 3.6.2
$ python2 -V
Python 2.7.13
$ python -V
Python 3.6.2
```

Sous Windows, vous avez un lanceur qui s'appelle `py`. Par défaut, il lance la version de python la plus récente installée, mais vous pouvez spécifier une version spécifique de la manière suivante :

```
C:\> py -2.7
```

pour lancer, par exemple, python en version 2.7. Vous trouverez toute la documentation nécessaire pour Windows sur cette page (en anglais) : https ://docs.python.org/3/using/windows.html

Pour éviter d'éventuelles confusions, nous précisons toujours `python3` dans le cours.

1.6 Installation de base

Vous utilisez Windows

La méthode recommandée sur Windows est de partir de la page https ://www.python.org/download où vous trouverez un programme d'installation qui contient tout ce dont vous aurez besoin pour suivre le cours.

Pour vérifier que vous êtes prêts, il vous faut lancer IDLE (quelque part dans le menu Démarrer) et vérifier le numéro de version.

Vous utilisez MacOS

Ici encore, la méthode recommandée est de partir de la page https ://www.python.org/download et d'utiliser le programme d'installation.

Sachez aussi, si vous utilisez déjà MacPorts (https ://www.macports.org), que vous pouvez également utiliser cet outil pour installer, par exemple python 3.6, avec la commande

```
$ sudo port install python36
```

Vous utilisez Linux

Dans ce cas il y est très probable que python-3.x soit déjà disponible sur votre machine. Pour vous en assurer, essayez de lancer la commande `python3` dans un terminal.

RHEL / Fedora Voici par exemple ce qu'on obtient depuis un terminal sur une machine installée en Fedora-20 :

```
$ python3
Python 3.6.2 (default, Jul 20 2017, 12:30:02)
[GCC 6.3.1 20161221 (Red Hat 6.3.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

Vérifiez bien le numéro de version qui doit être en 3.*x*. Si vous obtenez un message du style `python3: command not found` utilisez `dnf` (anciennement connu sous le nom de `yum`) pour installer le rpm `python3` comme ceci :

```
$ sudo dnf install python3
```

S'agissant d'`idle`, l'éditeur que nous utilisons dans le cours (optionnel si vous êtes familier avec un éditeur de texte), vérifiez sa présence comme ceci :

```
$ type idle3
idle is hashed (/usr/bin/idle3)
```

Ici encore, si la commande n'est pas disponible vous pouvez l'installer avec :

```
$ sudo yum install python3-tools
```

Debian / Ubuntu Ici encore, python-2.7 est sans doute déja disponible. Procédez comme ci-dessus, voici un exemple recueilli dans un terminal sur une machine installée en Ubuntu-14.04/trusty :

```
$ python3
Python 3.6.2 (default, Jul 20 2017, 12:30:02)
[GCC 6.3.1 20161221 (Red Hat 6.3.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

Pour installer python :

```
$ sudo apt-get install python3
```

Pour installer idle :

```
$ sudo apt-get install idle3
```

Installation de bibliothèques complémentaires

Il existe un outil très pratique pour installer des bibliothèques python, il s'appelle `pip3`, qui est documenté ici https ://pypi.python.org/pypi/pip

Sachez aussi, si par ailleurs vous utilisez un gestionnaire de package comme `rpm` sur RHEL, `apt-get` sur Debian, ou `port` sur MacOS, que de nombreux packages sont également disponibles au travers de ces outils.

1.7 Anaconda

Sachez qu'il existe beaucoup de distributions alternatives qui incluent python ; parmi elles, la plus populaire est sans aucun doute Anaconda, qui contient un grand nombre de bibliothèques de calcul scientifique, et également d'ailleurs jupyter pour travailler nativement sur des notebooks au format `.ipynb`.

Anaconda vient avec son propre gestionnaires de paquets pour l'installation de bibliothèques supplémentaires qui s'appelle `conda`.

1.8 Un peu de lecture

1.8.1 Complément - niveau basique

Le Zen de Python

Vous pouvez lire le "Zen de Python", qui résume la philosophie du langage, en important le module `this` avec ce code : (pour exécuter ce code, cliquer dans la cellule de code, et faites au clavier "Majuscule/Entrée" ou "Shift/Enter")

```
In [ ]: # le Zen de Python
        import this
```

Documentation

— On peut commencer par citer l'article de Wikipédia sur Python en français.
— La page sur le langage en français.
— La documentation originale de Python 3 - donc, en anglais - est un très bon point d'entrée lorsqu'on cherche un sujet particulier, mais (beaucoup) trop abondante pour être lue d'un seul trait. Pour chercher de la documentation sur un module particulier, le plus simple est encore d'utiliser Google - ou votre moteur de recherche favori - qui vous redirigera dans la grande majorité des cas vers la page qui va bien dans, précisément, la documentation de Python.

À titre d'exercice, cherchez la documentation du module `pathlib` en cherchant sur Google `"python module pathlib"`.

Historique et survol

— La FAQ officielle de Python (en anglais) sur les choix de conception et l'historique du langage.
— L'article de wikipedia (en anglais) sur l'historique du langage.
— Sur Wikipédia, un article (en anglais) sur la syntaxe et la sémantique de Python.

Un peu de folklore

— Le talk de Guido van Rossum à PyCon 2016.
— Sur YouTube, le sketch des Monty Python d'où provient les termes `spam`, `eggs` et autres `beans` que l'on utilise traditionnellement dans les exemples en Python plutôt que `foo` et `bar`.
— L'article Wikipédia correspondant, qui cite le langage Python.

1.8.2 Complément - niveau intermédiaire

Licence

— La licence d'utilisation est disponible ici.
— La page de la Python Software Foundation, qui est une entité légale similaire à nos associations de 1901, à but non lucratif ; elle possède les droits sur le langage.

Le processus de développement

— Comment les choix d'évolution sont proposés et discutés, au travers des PEP (Python Enhacement Proposal)
— http ://en.wikipedia.org/wiki/Python_(programming_language)#Development

— Le premier PEP décrit en détail le cycle de vie des PEPs
— http ://legacy.python.org/dev/peps/pep-0001/
— Le PEP 8, qui préconise un style de présentation (*style guide*)
— http ://legacy.python.org/dev/peps/pep-0008/
— L'index de tous les PEPs
— http ://legacy.python.org/dev/peps/

1.9 "Notebooks" Jupyter comme support de cours

Pour illustrer les vidéos du MOOC, nous avons choisi d'utiliser Jupyter pour vous rédiger les documents "mixtes" contenant du texte et du code Python, qu'on appelle des "notebooks", et dont le présent document est un exemple.

Nous allons dans la suite utiliser du code Python, pourtant nous n'avons pas encore abordé le langage. Pas d'inquiétude, ce code est uniquement destiné à valider le fonctionnement des notebooks, et nous n'utilisons que des choses très simples.

Avantages des notebooks

Comme vous le voyez, ce support permet un format plus lisible que des commentaires dans un fichier de code.

Nous attirons votre attention sur le fait que **les fragments de code peuvent être évalués et modifiés**. Ainsi vous pouvez facilement essayer des variantes autour du notebook original.

Notez bien également que le code Python est interprété **sur une machine distante**, ce qui vous permet de faire vos premiers pas avant même d'avoir procédé à l'installation de python sur votre propre ordinateur.

Comment utiliser les notebooks

En haut du notebook, vous avez une barre, contenant : * un titre pour le notebook, avec un numéro de version ; * une barre de menus avec les entrées `File`, `Edit`, `View`, `Insert...` ; * et une barre de boutons qui sont des raccourcis vers certains menus fréquemment utilisés. Si vous laissez votre souris au dessus d'un bouton, un petit texte apparaît, indiquant à quelle fonction correspond ce bouton.

Nous avons vu dans la vidéo qu'un notebook est constitué d'une suite de cellules, soit textuelles, soit contenant du code. Les cellules de code sont facilement reconnaissables, elles sont précédées de `In [ ]:`. La cellule qui suit celle que vous êtes en train de lire est une cellule de code.

Pour commencer, sélectionnez cette cellule de code avec votre souris, et appuyez dans la barre de boutons sur celui en forme de flèche triangulaire vers la droite (Play) :

```
In [ ]: 20 * 30
```

Comme vous le voyez, la cellule est "exécutée" (on dira plus volontiers évaluée), et on passe à la cellule suivante.

Alternativement vous pouvez simplement taper au clavier ***Shift+Enter***, ou selon les claviers ***Maj-Entrée***, pour obtenir le même effet. D'une manière générale, il est important d'apprendre et d'utiliser les raccourcis clavier, cela vous fera gagner beaucoup de temps par la suite.

La façon habituelle d'*exécuter* l'ensemble du notebook consiste à partir de la première cellule, et à taper ***Shift+Enter*** jusqu'au bout du notebook.

Lorsqu'une cellule de code a été évaluée, Jupyter ajoute sous la cellule `In` une cellule `Out` qui donne le résultat du fragment Python, soit ci-dessus 600.

Jupyter ajoute également un nombre entre les crochets pour afficher, par exemple ci-dessus, `In [1]:`. Ce nombre vous permet de retrouver l'ordre dans lequel les cellules ont été évaluées.

Vous pouvez naturellement modifier ces cellules de code pour faire des essais ; ainsi vous pouvez vous servir du modèle ci-dessous pour calculer la racine carrée de 3, ou essayer la fonction sur un nombre négatif et voir comment est signalée l'erreur.

```
In [ ]: # math.sqrt (pour square root) calcule la racine carrée
        import math
        math.sqrt(2)
```

On peut également évaluer tout le notebook en une seule fois en utilisant le menu *Cell -> Run All*.

Attention à bien évaluer les cellules dans l'ordre

Il est important que les cellules de code soient évaluées dans le bon ordre. Si vous ne respectez pas l'ordre dans lequel les cellules de code sont présentées, le résultat peut être inattendu.

En fait, évaluer un programme sous forme de notebook revient à le découper en petits fragments, et si on exécute ces fragments dans le désordre, on obtient naturellement un programme différent.

On le voit sur cet exemple :

```
In [ ]: message = "Il faut faire attention à l'ordre dans lequel on évalue les notebooks"
```

```
In [ ]: print(message)
```

Si un peu plus loin dans le notebook on fait par exemple :

```
In [ ]: # ceci a pour effet d'effacer la variable 'message'
        del message
```

qui rend le symbole `message` indéfini, alors bien sûr on ne peut plus évaluer la cellule qui fait `print` puisque la variable `message` n'est plus connue de l'interpréteur.

Réinitialiser l'interpréteur

Si vous faites trop de modifications, ou perdez le fil de ce que vous avez evalué, il peut être utile de redémarrer votre interpréteur. Le menu *Kernel* → *Restart* vous permet de faire cela, un peu à la manière de IDLE qui repart d'un interpréteur vierge lorsque vous utilisez la fonction F5.

Le menu *Kernel* → *Interrupt* peut être quant à lui utilisé si votre fragment prend trop longtemps à s'exécuter (par exemple vous avez écrit une boucle dont la logique est cassée et qui ne termine pas).

Vous travaillez sur une copie

Un des avantages principaux des notebooks est de vous permettre de modifier le code que nous avons écrit, et de voir par vous mêmes comment se comporte le code modifié.

Pour cette raison, chaque élève dispose de sa **propre copie** de chaque notebook, vous pouvez bien sûr apporter toutes les modifications que vous souhaitez à vos notebooks sans affecter les autres étudiants.

Revenir à la version du cours

Vous pouvez toujours revenir à la version "du cours" grâce au menu *File → Reset to original*.

Attention, avec cette fonction vous restaurez **tout le notebook** et donc **vous perdez vos modifications sur ce notebook**.

Télécharger au format Python

Vous pouvez télécharger un notebook au format Python sur votre ordinateur grâce au menu *File → Download as → Python*

Les cellules de texte sont préservées dans le résultat sous forme de commentaires Python.

Partager un notebook en lecture seule

Enfin, avec le menu *File → Share static version*, vous pouvez publier une version en lecture seule de votre notebook ; vous obtenez une URL que vous pouvez publier par exemple pour demander de l'aide sur le forum. Ainsi, les autres étudiants peuvent accéder en lecture seule à votre code.

Notez que lorsque vous utilisez cette fonction plusieurs fois, c'est toujours la dernière version publiée que verront vos camarades, l'URL utilisée reste toujours la même pour un étudiant et un notebook donné.

Ajouter des cellules

Vous pouvez ajouter une cellule n'importe où dans le document avec le bouton **+** de la barre de boutons.

Aussi, lorsque vous arrivez à la fin du document, une nouvelle cellule est créée chaque fois que vous évaluez la dernière cellule ; de cette façon vous disposez d'un brouillon pour vos propres essais.

À vous de jouer.

1.10 Modes d'exécution

Nous avons donc à notre disposition plusieurs façons d'exécuter un programme Python. Nous allons les étudier plus en détail :

Quoi	Avec quel outil
fichier complet	`python3 <fichier>.py`
ligne à ligne	`python3` en mode interactif ou sous `ipython3` ou avec IDLE
par fragments	dans un notebook

Pour cela nous allons voir le comportement d'un tout petit programme Python lorsqu'on l'exécute sous ces différents environnements.

On veut surtout expliquer une petite différence quant au niveau de détail de ce qui se trouve imprimé.

Essentiellement, lorsqu'on utilise l'interpréteur en mode interactif - ou sous IDLE - à chaque fois que l'on tape une ligne, le résultat est **calculé** (on dit aussi **évalué**) puis **imprimé**.

Par contre, lorsqu'on écrit tout un programme, on ne peut plus imprimer le résultat de toutes les lignes, cela produirait un flot d'impression beaucoup trop important. Par consé-

quent, si vous ne déclenchez pas une impression avec, par exemple, la fonction `print`, rien ne s'affichera.

Enfin, en ce qui concerne le notebook, le comportement est un peu hybride entre les deux, en ce sens que seul le **dernier résultat** de la cellule est imprimé.

L'interpréteur Python interactif

Le programme choisi est très simple, c'est le suivant :

```
10 * 10
20 * 20
30 * 30
```

Voici comment se comporte l'interpréteur interactif quand on lui soumet ces instructions :

```
$ python3
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 10 * 10
100
>>> 20 * 20
400
>>> 30 * 30
900
>>> exit()
$
```

Notez que pour terminer la session, il nous faut "sortir" de l'interpréteur en tapant `exit()`.

On peut aussi taper `Control-D` sous Linux ou MacOS.

Comme on le voit ici, l'interpréteur imprime **le résultat de chaque ligne**. On voit bien apparaître toutes les valeurs calculées, 100, 400, puis enfin 900.

Sous-forme de programme constitué

Voyons à présent ce que donne cette même séquence de calculs dans un programme complet. Pour cela, il nous faut tout d'abord fabriquer un fichier avec un suffixe en `.py`, en utilisant par exemple un éditeur de fichier. Le résultat doit ressembler à ceci :

```
$ cat foo.py
10 * 10
20 * 20
30 * 30
$
```

Exécutons à présent ce programme :

```
$ python3 foo.py
$
```

On constate donc que ce programme **ne fait rien !** En tous cas, selon toute apparence.

En réalité, les 3 valeurs 100, 400 et 900 sont bien calculées, mais comme aucune instruction `print` n'est présente, rien n'est imprimé et le programme se termine sans signe apparent d'avoir réellement fonctionné.

Ce comportement peut paraître un peu déroutant au début, mais comme nous l'avons mentionné c'est tout à fait délibéré. Un programme fonctionnel faisant facilement plusieurs milliers de lignes, voire beaucoup plus, il ne serait pas du tout réaliste que chaque ligne produise une impression, comme c'est le cas en mode interactif.

Dans un notebook

Voici à présent le même programme dans un notebook :

```
In [ ]: 10 * 10
        20 * 20
        30 * 30
```

Lorsqu'on exécute cette cellule (rappel : sélectionner la cellule, et utiliser le bouton en forme de flèche vers la droite, ou entrer **"Shift+Enter"** au clavier), on obtient une seule valeur dans la rubrique `Out[]`, 900, qui correspond **au résultat de la dernière ligne.**

Utiliser `print`

Ainsi, pour afficher un résultat intermédiaire, on utilise l'instruction `print`. Nous verrons cette instruction en détail dans les semaines qui viennent, mais en guise d'introduction disons seulement que c'est une fonction comme les autres en Python 3.

```
In [ ]: a = 10
        b = 20

        print(a, b)
```

On peut naturellement mélanger des objets de plusieurs types, et donc mélanger des chaînes de caratères et des nombres pour obtenir un résultat un peu plus lisible. En effet, lorsque le programme devient gros, il est important de savoir à quoi correspond une ligne dans le flot de toutes les impressions. Aussi on préfèrera quelque chose comme :

```
In [ ]: print("a =", a, "et b =", b)
```

```
In [ ]: # ou encore, équivalente mais avec un f-string
        print(f"a = {a} et b = {b}")
```

Une pratique courante consiste d'ailleurs à utiliser les commentaires pour laisser dans le code les instructions `print` qui correspondent à du debug (c'est-à-dire qui ont pu être utiles lors de la mise au point et qu'on veut pouvoir réactiver rapidement).

Utiliser `print` pour "sous-titrer" une affectation

Remarquons enfin que l'affectation à une variable ne retourne aucun résultat.
C'est à dire, en pratique, que si on écrit :

```
In [ ]: a = 100
```

même une fois l'expression évaluée par l'interpréteur, aucune ligne `Out[]` n'est ajoutée.
C'est pourquoi, il nous arrivera parfois d'écrire alors plutôt, et notamment lorsque l'expression est complexe et pour rendre explicite la valeur qui vient d'être affectée :

```
In [ ]: a = 100; print(a)
```

Notez bien que cette technique est uniquement pédagogique et n'a absolument aucun autre intérêt dans la pratique, il n'est **pas recommandé** de l'utiliser en dehors de ce contexte.

1.11 La suite de Fibonacci

1.11.1 Complément - niveau basique

Voici un premier exemple de code qui tourne.

Nous allons commencer par le faire tourner dans ce notebook. Nous verrons en fin de séance comment le faire fonctionner localement sur votre ordinateur.

Le but de ce programme est de calculer la suite de Fibonacci, qui est définie comme ceci :

— $u_0 = 1$
— $u_1 = 1$
— $\forall n >= 2, u_n = u_{n-1} + u_{n-2}$

Ce qui donne pour les premières valeurs :

n	fibonacci(n)
0	1
1	1
2	2
3	3
4	5
5	8
6	13

On commence par définir la fonction `fibonacci` comme il suit. Naturellement vous n'avez pas encore tout le bagage pour lire ce code, ne vous inquiétez pas, nous allons vous expliquer tout ça dans les prochaines semaines. Le but est uniquement de vous montrer un fonctionnement de l'interpréteur Python et de IDLE.

```
In [ ]: def fibonacci(n):
            "retourne le nombre de fibonacci pour l'entier n"
            # pour les petites valeurs de n il n'y a rien à calculer
            if n <= 1:
                return 1
            # sinon on initialise f1 pour n-1 et f2 pour n-2
            f2, f1 = 1, 1
            # et on itère n-1 fois pour additionner
            for i in range(2, n + 1):
                f2, f1 = f1, f1 + f2
        #         print(i, f2, f1)
            # le résultat est dans f1
            return f1
```

Pour en faire un programme utilisable on va demander à l'utilisateur de rentrer un nombre ; il faut le convertir en entier car `input` renvoie une chaîne de caractères :

```
In [ ]: entier = int(input("Entrer un entier "))
```

On imprime le résultat :

```
In [ ]: print(f"fibonacci({entier}) = {fibonacci(entier)}")
```

Exercice

Vous pouvez donc à présent :

— exécuter le code dans ce notebook
— télécharger ce code sur votre disque comme un fichier `fibonacci_prompt.py`
— utilisez pour cela le menu *"File -> Download as -> Python"*
— et **renommez le fichier obtenu** au besoin
— l'exécuter sous IDLE
— le modifier, par exemple pour afficher les résultats intermédiaires
— on a laissé exprès une fonction `print` en commentaire que vous pouvez réactiver simplement
— l'exécuter avec l'interpréteur Python comme ceci :

```
$ python3 fibonacci_prompt.py
```

Ce code est volontairement simple et peu robuste pour ne pas l'alourdir. Par exemple, ce programme se comporte mal si vous entrez un entier négatif.

Nous allons voir tout de suite une version légèrement différente qui va vous permettre de donner la valeur d'entrée sur la ligne de commande.

1.12 La suite de Fibonacci (suite)

1.12.1 Complément - niveau intermédiaire

Nous reprenons le cas de la fonction `fibonacci` que nous avons déjà vue, mais cette fois nous voulons que l'utilisateur puisse indiquer l'entier en entrée de l'algorithme, non plus en répondant à une question, mais sur la ligne de commande, c'est-à-dire en tapant :

```
$ python3 fibonacci.py 12
```

Avertissement :
Attention, cette version-ci **ne fonctionne pas dans ce notebook**, justement car on n'a pas de moyen dans un notebook d'invoquer un programme en lui passant des arguments de cette façon. Ce notebook est rédigé pour vous permettre de vous entraîner avec la fonction de téléchargement au format Python, qu'on a vue dans la vidéo, et de faire tourner ce programme sur votre propre ordinateur.

Le module `argparse`

Cette fois nous importons le module `argparse`, c'est lui qui va nous permettre d'interpréter les arguments passés sur la ligne de commande.

```
In [ ]: from argparse import ArgumentParser
```

Puis nous répétons la fonction `fibonacci` :

```
In [ ]: def fibonacci(n):
            "retourne le nombre de fibonacci pour l'entier n"
            # pour les petites valeurs de n il n'y a rien a calculer
            if n <= 1:
                return 1
            # sinon on initialise f1 pour n-1 et f2 pour n-2
```

```
        f2, f1 = 1, 1
        # et on itère n-1 fois pour additionner
        for i in range(2, n + 1):
            f2, f1 = f1, f1 + f2
#           print(i, f2, f1)
        # le résultat est dans f1
        return f1
```

Remarque :
Certains d'entre vous auront évidemment remarqué que l'on aurait pu éviter de copier-coller la fonction `fibonacci` comme cela ; c'est à ça que servent les modules, mais nous n'en sommes pas là.

Un objet `parser`

À présent, nous utilisons le module `argparse`, pour lui dire qu'on attend exactement un argument sur la ligne de commande, et qu'il doit être un entier. Ici encore ne vous inquiétez pas si vous ne comprenez pas tout le code, l'objectif est de vous donner un morceau de code utilisable tout de suite pour jouer avec votre interpréteur Python.

```
In [ ]: # à nouveau : ceci n'est pas conçu pour être exécuté dans le notebook !
        parser = ArgumentParser()
        parser.add_argument(dest="entier", type=int,
                            help="entier d'entrée")
        input_args = parser.parse_args()
        entier = input_args.entier
```

Nous pouvons à présent afficher le résultat :

```
In [ ]: print(f"fibonacci({entier}) = {fibonacci(entier)}")
```

Vous pouvez donc à présent :

— télécharger ce code sur votre disque comme un fichier `fibonacci.py` en utilisant le menu *"File -> Download as -> Python"*

— l'exécuter avec simplement l'interpéteur Python comme ceci :
$ python3 fibonacci.py 56

1.13 La ligne *shebang*

```
#!/usr/bin/env python3
```

1.13.1 Complément - niveau avancé

Ce complément est uniquement valable pour MacOS et Linux.

Le besoin

Nous avons vu dans la vidéo que pour lancer un programme Python on fait essentiellement depuis le terminal :

```
$ python3 mon_module.py
```

Lorsqu'il s'agit d'un programme que l'on utilise fréquemment, on n'est pas forcément dans le répertoire où se trouve le programme Python, aussi dans ce cas on peut utiliser un chemin "absolu", c'est-à-dire à partir de la racine des noms de fichiers, comme par exemple :

```
$ python3 /le/chemin/jusqu/a/mon_module.py
```

Sauf que c'est assez malcommode, et cela devient vite pénible à la longue.

La solution

Sur Linux et MacOS, il existe une astuce utile pour simplifier cela. Voyons comment s'y prendre, avec par exemple le programme `fibonacci.py` que vous pouvez télécharger ici (nous verrons ce code en détail dans les deux prochains compléments). Commencez par sauver ce code sur votre ordinateur dans un fichier qui s'appelle, bien entendu, `fibonacci.py`.

On commence par éditer le tout début du fichier pour lui ajouter une **première ligne** :

```
#!/usr/bin/env python3

## La suite de Fibonacci (Suite)
...etc...
```

Cette première ligne s'appelle un Shebang dans le jargon Unix. Unix stipule que le Shebang doit être en **première position** dans le fichier.

Ensuite on rajoute au fichier, depuis le terminal, le caractère exécutable comme ceci :

```
$ pwd
/le/chemin/jusqu/a/

$ chmod +x fibonacci.py
```

À partir de là, vous pouvez utiliser le fichier `fibonacci.py` comme une commande, sans avoir à mentionner `python3`, qui sera invoqué au travers du shebang :

```
$ /le/chemin/jusqu/a/fibonacci.py 20
fibonacci(20) = 10946
```

Et donc vous pouvez aussi le déplacer dans un répertoire qui est dans votre variable `PATH`, et le rendre ainsi accessible depuis n'importe quel répertoire en faisant simplement :

```
$ export PATH=/le/chemin/jusqu/a:$PATH

$ cd /tmp
$ fibonacci.py 20
fibonacci(20) = 10946
```

Remarque : tout ceci fonctionne très bien tant que votre point d'entrée - ici `fibonacci.py` - n'utilise que des modules standards. Dans le cas où le point d'entrée vient avec au moins un module, il est également nécessaire d'installer ces modules quelque part, et d'indiquer au point d'entrée comment les trouver, nous y reviendrons dans la semaine où nous parlerons des modules.

1.14 Dessiner un carré

1.14.1 Exercice - niveau intermédiaire

Voici un tout petit programme qui dessine un carré.

Il utilise le module `turtle`, conçu précisément à des fins pédagogiques. Pour des raisons techniques, le module `turtle` n'est **pas disponible** au travers de la plateforme FUN.

Il est donc inutile d'essayer d'exécuter ce programme depuis le notebook. L'objectif de cet exercice est plutôt de vous entraîner à télécharger ce programme en utilisant le menu *"File -> Download as -> Python"*, puis à le charger dans votre IDLE pour l'exécuter sur votre machine.

Attention également à sauver le programme téléchargé **sous un autre nom** que `turtle.py`, car sinon vous allez empêcher python de trouver le module standard `turtle` ; appelez-le par exemple `turtle_basic.py`.

```
In [ ]: # on a besoin du module turtle
        import turtle
```

On commence par définir une fonction qui dessine un carré de coté `length` :

```
In [ ]: def square(length):
            "have the turtle draw a square of side <length>"
            for side in range(4):
                turtle.forward(length)
                turtle.left(90)
```

Maintenant on commence par initialiser la tortue :

```
In [ ]: turtle.reset()
```

On peut alors dessiner notre carré :

```
In [ ]: square(200)
```

Et pour finir on attend que l'utilisateur clique dans la fenêtre de la tortue, et alors on termine :

```
In [ ]: turtle.exitonclick()
```

1.14.2 Exercice - niveau avancé

Naturellement vous pouvez vous amuser à modifier ce code pour dessiner des choses un peu plus amusantes.

Dans ce cas, commencez par chercher "*module python turtle*" dans votre moteur de recherche favori, pour localiser la documentation du module `turtle`.

Vous trouverez quelques exemples pour commencer ici : * turtle_multi_squares.py pour dessiner des carrés à l'emplacement de la souris en utilisant plusieurs tortues ; * turtle_fractal.py pour dessiner une fractale simple ; * turtle_fractal_reglable.py une variation sur la fractale, plus paramétrable.

1.15 Noms de variables

1.15.1 Complément - niveau basique

Revenons sur les noms de variables autorisés ou non.
Les noms les plus simples sont constitués de lettres. Par exemple :

```
In [ ]: factoriel = 1
```

On peut utiliser aussi les majuscules, mais attention cela définit une variable différente. Ainsi :

```
In [ ]: Factoriel = 100
        factoriel == Factoriel
```

Le signe == permet de tester si deux variables ont la même valeur. Si les variables ont la même valeur, le test retournera `True`, et `False` sinon. On y reviendra bien entendu.

Conventions habituelles

En règle générale, on utilise **uniquement des minuscules** pour désigner les variables simples (ainsi d'ailleurs que pour les noms de fonctions), les majuscules sont réservées en principe pour d'autres sortes de variables, comme les noms de classe, que nous verrons ultérieurement.
Notons, qu'il s'agit uniquement d'une convention, ceci n'est pas imposé par le langage lui-même.
Pour des raisons de lisibilité, il est également possible d'utiliser le tiret bas _ dans les noms de variables. On préfèrera ainsi :

```
In [ ]: age_moyen = 75 # oui
```

plutôt que ceci (bien qu'autorisé par le langage) :

```
In [ ]: AgeMoyen = 75 # autorisé, mais non
```

On peut également utiliser des chiffres dans les noms de variables comme par exemple :

```
In [ ]: age_moyen_dept75 = 80
```

avec la restriction toutefois que le premier caractère ne peut pas être un chiffre, cette affectation est donc refusée :

```
In [ ]: 75_age_moyen = 80 # erreur de syntaxe
```

Le tiret bas comme premier caractère

Il est par contre, possible de faire commencer un nom de variable par un tiret bas comme premier caractère ; toutefois, à ce stade, nous vous déconseillons d'utiliser cette pratique qui est réservée à des conventions de nommage bien spécifiques.

```
In [ ]: _autorise_mais_deconseille = 'Voir le PEP 008'
```

Et en tous cas, il est **fortement déconseillé** d'utiliser des noms de la forme `__variable__` qui sont réservés au langage. Nous reviendrons sur ce point dans le futur, mais reagardez par exemple cette variable que nous n'avons définie nulle part mais qui pourtant existe bel et bien :

```
In [ ]: __name__  # ne définissez pas vous-même de variables de ce genre
```

Ponctuation

Dans la plage des caractères ASCII, il n'est **pas possible** d'utiliser d'autres caractères que les caractères alphanumériques et le tiret bas. Notamment le tiret haut - est interprété comme l'opération de soustraction. Attention donc à cette erreur fréquente :

```
In [ ]: age-moyen = 75  # erreur : en fait python l'interprète comme 'age - moyen = 75'
```

Caractères exotiques

En Python 3, il est maintenant aussi possible d'utiliser des caractères Unicode dans les identificateurs :

```
In [ ]: # les caractères accentués sont permis
        nom_élève = "Jules Maigret"
```

```
In [ ]: # ainsi que l'alphabet grec
        from math import cos, pi as k+kn
        θ = k+kn / 4
        cos(θ)
```

Tous les caractères Unicode ne sont pas permis - heureusement car cela serait source de confusion. Nous citons dans les références les documents qui précisent quels sont exactement les caractères autorisés.

```
In [ ]: # ce caractère n'est pas autorisé
        # il est considéré comme un signe mathématique (produit)
        ∏ = 10
```

Conseil Il est **très vivement** recommandé :

— tout d'abord de coder **en anglais** ;
— ensuite de **ne pas** définir des identificateurs avec des caractères non ASCII, dans toute la mesure du possible ;
— enfin si vous utilisez un encodage autre que utf-8, vous **devez** bien **spécifier l'encodage** utilisé dans votre fichier source ; nous y reviendrons en deuxième semaine.

Pour en savoir plus

Pour les esprits curieux, Guido van Rossum, le fondateur de Python, est le co-auteur d'un document qui décrit les conventions de codage à utiliser dans la bibliothèque standard Python. Ces règles sont plus restrictives que ce que le langage permet de faire, mais constituent une lecture intéressante si vous projetez d'écrire beaucoup de Python.

Voir dans le PEP 008 la section consacrée aux règles de nommage - (en anglais)

Voir enfin, au sujet des caractères exotiques dans les identificateurs :

— https ://www.python.org/dev/peps/pep-3131/ qui définit les caractères exotiques autorisés, et qui repose à son tour sur
— http ://www.unicode.org/reports/tr31/ (très technique !)

1.16 Les mots-clés de Python

Mots réservés

Il existe en Python certains mots spéciaux, qu'on appelle des mots-clés, ou *keywords* en anglais, qui sont réservés et **ne peuvent pas être utilisés** comme identifiants, c'est-à-dire comme un nom de variable.

C'est le cas par exemple pour l'instruction `if`, que nous verrons prochainement, qui permet bien entendu d'exécuter tel ou tel code selon le résultat d'un test.

```
In [ ]: variable = 15
        if variable <= 10:
            print("en dessous de la moyenne")
        else:
            print("au dessus")
```

À cause de la présence de cette instruction dans le langage, il n'est pas autorisé d'appeler une variable `if`.

```
In [ ]: # interdit, if est un mot-clé
        if = 1
```

Liste complète

Voici la liste complète des mots-clés :

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

Nous avons indiqué en gras les nouveautés par rapport à Python 2 (sachant que réciproquement `exec` et `print` ont perdu leur statut de mot-clé depuis Python 2, ce sont maintenant des fonctions).

Il vous faudra donc y prêter attention, surtout au début, mais avec un tout petit peu d'habitude vous saurez rapidement les éviter.

Vous remarquerez aussi que tous les bons éditeurs de texte supportant du code Python vont colorer les mots-clés différemment des variables. Par exemple, IDLE colorie les mots-clés en orange, vous pouvez donc très facilement vous rendre compte que vous allez, par erreur, en utiliser un comme nom de variable.

Cette fonctionalité de *coloration syntaxique* permet d'identifier d'un coup d'œil (grâce à un code de couleur) le rôle des différents éléments de votre code (variable, mots-clés, etc.) D'une manière générale, nous vous déconseillons fortement d'utiliser un éditeur de texte qui n'offre pas cette fonctionalité de coloration syntaxique.

Pour en savoir plus

On peut se reporter à cette page :
https ://docs.python.org/3/reference/lexical_analysis.html#keywords

1.17 Un peu de calcul sur les types

1.17.1 Complément - niveau basique

La fonction `type`

Nous avons vu dans la vidéo que chaque objet possède un type. On peut très simplement accéder au type d'un objet en appelant une fonction *built-in*, c'est-à-dire prédéfinie dans Python, qui s'appelle, eh bien oui, `type`.

On l'utilise tout simplement comme ceci :

```
In [ ]: type(1)
```

```
In [ ]: type('spam')
```

Cette fonction est assez peu utilisée par les programmeurs expérimentés, mais va nous être utile à bien comprendre le langage, notamment pour manipuler les valeurs numériques.

Types, variables et objects

On a vu également que le type est attaché **à l'objet** et non à la variable.

```
In [ ]: x = 1
        type(x)
```

```
In [ ]: # la variable x peut référencer un objet de n'importe quel type

        x = [1, 2, 3]
        type(x)
```

1.17.2 Complément - niveau avancé

La fonction `isinstance`

Une autre fonction prédéfinie, voisine de `type` mais plus utile dans la pratique, est la fonction `isinstance` qui permet de savoir si un objet est d'un type donné. Par exemple :

```
In [ ]: isinstance(23, int)
```

À la vue de ce seul exemple, on pourrait penser que `isinstance` est presque identique à `type` ; en réalité elle est un peu plus élaborée, notamment pour la programmation objet et l'héritage, nous aurons l'occasion d'y revenir.

On remarque ici en passant que la variable `int` est connue de Python alors que nous ne l'avons pas définie. Il s'agit d'une variable prédéfinie, qui désigne le type des entiers, que nous étudierons très bientôt.

Pour conclure sur `isinstance`, cette fonction est utile en pratique précisément parce que Python est à typage dynamique. Aussi il est souvent utile de s'assurer qu'une variable passée à une fonction est du (ou des) type(s) attendu(s), puisque contrairement à un langage typé statiquement comme C++, on n'a aucune garantie de ce genre à l'exécution. À nouveau, nous aurons l'occasion de revenir sur ce point.

1.18 Gestion de la mémoire

1.18.1 Complément - niveau basique

L'objet de ce complément est de vous montrer qu'avec Python vous n'avez pas à vous préoccuper de la mémoire. Pour expliquer la notion de gestion de la mémoire, il nous faut donner un certain nombre de détails sur d'autres langages comme C et C++. Si vous souhaitez suivre ce cours à un niveau basique vous pouvez ignorer ce complément et seulement retenir que Python se charge de tout pour vous :)

1.18.2 Complément - niveau intermédiaire

Langages de bas niveau

Dans un langage traditionnel de bas niveau comme C ou C++, le programmeur est en charge de l'allocation - et donc de la libération - de la mémoire.

Ce qui signifie que, sauf pour les valeurs stockées dans la pile, le programmeur est amené : * à réclamer de la mémoire au système d'exploitation en appelant explicitement `malloc` (C) ou `new` (C++) ; * et réciproquement à rendre cette mémoire au système d'exploitation lorsqu'elle n'est plus utilisée, en appelant `free` (C) ou `delete` (C++).

Avec ce genre de langage, la gestion de la mémoire est un aspect important de la programmation. Ce modèle offre une grande flexibilité, mais au prix d'un coût élevé en termes de vitesse de développement.

En effet, il est assez facile d'oublier de libérer la mémoire après usage, ce qui peut conduire à épuiser les ressources disponibles. À l'inverse, utiliser une zone mémoire non allouée peut conduire à des bugs très difficiles à localiser et à des problèmes de sécurité majeurs. Notons qu'une grande partie des attaques en informatiques reposent sur l'exploitation d'erreurs de gestion de la mémoire.

Langages de haut niveau

Pour toutes ces raisons, avec un langage de plus haut niveau comme Python, le programmeur est libéré de cet aspect de la programmation.

Pour anticiper un peu sur le cours des semaines suivantes, voici ce que vous pouvez garder en tête s'agissant de la gestion mémoire en Python : * vous créez vos objets au fur et à mesure de vos besoins ;

— vous n'avez pas besoin de les libérer explicitement, le "*Garbage Collector*" de Python va s'en charger pour recycler la mémoire lorsque c'est possible ;

— Python a tendance à être assez gourmand en mémoire, comparé à un langage de bas niveau, car tout est objet et chaque objet est assorti de *méta-informations* qui occupent une place non négligeable. Par exemple, chaque objet possède au minimum :

— une référence vers son type - c'est le prix du typage dynamique ;

— un compteur de références - le nombre d'autres valeurs (variables ou objets) qui pointent vers l'objet, cette information est notamment utilisée, précisément, par le *Garbage Collector* pour déterminer si la mémoire utilisée par un objet peut être libérée ou non.

— un certain nombre de types prédéfinis et non mutables sont implémentés en Python comme des *singletons*, c'est-à-dire qu'un seul objet est créé et partagé, c'est le cas par exemple pour les petits entiers et les chaînes de caractères, on en reparlera ;

— lorsqu'on implémente une classe, il est possible de lui conférer cette caractéristique de singleton, de manière à optimiser la mémoire nécessaire pour exécuter un programme.

1.19 Typages statique et dynamique

1.19.1 Complément - niveau intermédiaire

Parmi les langages typés, on distingue les langages à typage statique et ceux à typage dynamique. Ce notebook tente d'éclaircir ces notions pour ceux qui n'y sont pas familiers.

Typage statique

À une extrémité du spectre, on trouve les langages compilés, dits à typage statique, comme par exemple C ou C++.

En C on écrira, par exemple, une version simpliste de la fonction factoriel comme ceci :

```
int factoriel(int n) {
    int result = 1;
    for (int loop = 1; loop <= n; loop++)
        result *= loop;
    return result;
}
```

Comme vous pouvez le voir - ou le deviner - toutes les **variables** utilisées ici (comme par exemple `n`, `result` et `loop`) sont typées :

- on doit appeler `factoriel` avec un argument `n` qui doit être un entier (`int` est le nom du type entier) ;
- les variables internes `result` et `loop` sont de type entier ;
- `factoriel` retourne une valeur de type entier.

Ces informations de type ont essentiellement trois fonctions :

- en premier lieu, elles sont nécessaires au compilateur. En C si le programmeur ne précisait pas que `result` est de type entier, le compilateur n'aurait pas suffisamment d'éléments pour générer le code assembleur correspondant ;
- en contrepartie, le programmeur a un contrôle très fin de l'usage qu'il fait de la mémoire et du matériel. Il peut choisir d'utiliser un entier sur 32 ou 64 bits, signé ou pas, ou construire avec `struct` et `union` un arrangement de ses données ;
- enfin, et surtout, ces informations de type permettent de faire un contrôle *a priori* de la validité du programme, par exemple, si à un autre endroit dans le code on trouve :

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    /* le premier argument de la ligne de commande est argv[1] */
    char *input = argv[1];
    /* calculer son factoriel et afficher le résultat */
    printf("Factoriel (%s) = %d\n", input, factoriel(input));
    /*                                          ^^^^^
     * ici on appelle factoriel avec une entrée de type 'chaîne de caracteres' */
}
```

alors le compilateur va remarquer qu'on essaie d'appeler `factoriel` avec comme argument `input` qui, pour faire simple, est une chaîne de caractères et comme `factoriel` s'attend à recevoir un entier, ce programme n'a aucune chance de compiler.

On parle alors de **typage statique**, en ce sens que chaque **variable** a exactement un type qui est défini par le programmeur une bonne fois pour toutes.

C'est ce qu'on appelle le **contrôle de type**, ou *type-checking* en anglais. Si on ignore le point sur le contrôle fin de la mémoire, qui n'est pas crucial à notre sujet, ce modèle de contrôle de type présente :

— l'**inconvénient** de demander davantage au programmeur (je fais abstraction, à ce stade et pour simplifier, de langages à inférence de types comme ML et Haskell) ;
— et l'**avantage** de permettre un contrôle étendu, et surtout précoce (avant même de l'exécuter), de la bonne correction du programme.

Cela étant dit, le typage statique en C n'empêche pas le programmeur débutant d'essayer d'écrire dans la mémoire à partir d'un pointeur `NULL` - et le programme de s'interrompre brutalement. Il faut être conscient des limites du typage statique.

Typage dynamique

À l'autre bout du spectre, on trouve des langages comme, eh bien, Python.
Pour comprendre cette notion de typage dynamique, regardons la fonction suivante `somme`.

```
In [ ]: def somme(*largs):
            "retourne la somme de tous ses arguments"
            if not largs:
                return 0
            result = largs[0]
            for i in range(1, len(largs)):
                result += largs[i]
            return result
```

Naturellement, vous n'êtes pas à ce stade en mesure de comprendre le fonctionnement intime de la fonction. Mais vous pouvez tout de même l'utiliser :

```
In [ ]: somme(12, 14, 300)
```

```
In [ ]: liste1 = ['a', 'b', 'c']
        liste2 = [0, 20, 30]
        liste3 = ['spam', 'eggs']
        somme(liste1, liste2, liste3)
```

Vous pouvez donc constater que `somme` peut fonctionner avec des objets de types différents. En fait, telle qu'elle est écrite, elle va fonctionner s'il est possible de faire + entre ses arguments. Ainsi, par exemple, on pourrait même faire :

```
In [ ]: # Python sait faire + entre deux chaînes de caractères
        somme('abc', 'def')
```

Mais par contre on ne pourrait pas faire

```
In [ ]: # ceci va déclencher une exception à l'exécution
        somme(12, [1, 2, 3])
```

Il est utile de remarquer que le typage de Python, qui existe bel et bien comme on le verra, est qualifié de dynamique parce que le type est attaché **à un objet** et non à la variable qui le référence. On aura bien entendu l'occasion d'approfondir tout ça dans le cours.
En Python, on fait souvent référence au typage sous l'appellation *duck typing*, de manière imagée :

If it looks like a duck and quacks like a duck, it's a duck.

On voit qu'on se trouve dans une situation très différente de celle du programmeur C/C++, en ce sens que :

— à l'écriture du programme, il n'y aucun des surcoûts qu'on trouve avec C ou C++ en terme de définition de type ;
— aucun contrôle de type n'est effecué *a priori* par le langage au moment de la définition de la fonction `somme` ;
— par contre au moment de l'éxécution, s'il s'avère qu'on tente de faire une somme entre deux types qui ne peuvent pas être additionnés, comme ci-dessus avec un entier et une liste, le programme ne pourra pas se dérouler correctement.

Il y a deux points de vue vis-à-vis de la question du typage.

Les gens habitués au *typage statique* se plaignent du typage dynamique en disant qu'on peut écrire des programmes faux et qu'on s'en rend compte trop tard - à l'exécution.

À l'inverse les gens habitués au *typage dynamique* font valoir que le typage statique est très partiel, par exemple, en C si on essaie d'écrire dans un pointeur `NULL`, le système d'exploitation ne le permet pas et le programme sort tout aussi brutalement.

Bref, selon le point de vue, le typage dynamique est vécu comme un inconvénient (pas assez de bonnes propriétés détectées par le langage) ou comme un avantage (pas besoin de passer du temps à déclarer le type des variables, ni à faire des conversions pour satisfaire le compilateur). Vous remarquerez cependant qu'à l'usage, en terme de vitesse de développement, les inconvénients du typage dynamique sont très largement compensés par ses avantages.

Type hints

Signalons enfin que depuis python-3.5, il est **possible** d'ajouter des annotations de type, pour expliciter les suppositions qui sont faites par le programmeur pour le bon foncionnement du code.

Nous aurons là encore l'occasion de détailler ce point dans le cours, signalons simplement que ces annotations sont totalement optionnelles, et que même lorsqu'elles sont présentes elles ne sont pas utilisées à l'exécution par l'interpréteur. L'idée est plutôt de permettre à des outils externes, comme par exemple `mypy`, d'effectuer des contrôles plus poussés concernant la correction du programme.

1.20 Utiliser Python comme une calculette

Lorsque vous démarrez l'interprète Python, vous disposez en fait d'une calculette, par exemple, vous pouvez taper :

```
In [ ]: 20 * 60
```

Les règles de **priorité** entre les opérateurs sont habituelles, les produits et divisions sont évalués en premier, ensuite les sommes et soustractions :

```
In [ ]: 2 * 30 + 10 * 5
```

De manière générale, il est recommandé de bien parenthéser ses expressions. De plus, les parenthèses facilitent la lecture d'expressions complexes.

Par exemple, il vaut mieux écrire ce qui suit, qui est équivalent mais plus lisible :

```
In [ ]: (2 * 30) + (10 * 5)
```

Attention, en Python la division / est une division naturelle :

```
In [ ]: 48 / 5
```

Rappelez-vous des opérateurs suivants qui sont très pratiques :

code	opération
//	quotient
%	modulo
**	puissance

```
In [ ]: # calculer un quotient
        48 // 5
```

```
In [ ]: # modulo (le reste de la division par)
        48 % 5
```

```
In [ ]: # puissance
        2 ** 10
```

Vous pouvez facilement faire aussi des calculs sur les complexes. Souvenez-vous seulement que la constante complexe que nous notons `i` en français se note `j` en Python, ce choix a été fait par le BDFL - alias Guido van Rossum - pour des raisons de lisibilité :

```
In [ ]: # multiplication de deux nombres complexes
        (2 + 3j) * 2.5j
```

Aussi, pour entrer ce nombre complexe `j`, il faut toujours le faire précéder d'un nombre, donc ne pas entrer simplement `j` (qui serait compris comme un nom de variable, nous allons voir ça tout de suite) mais plutôt `1j` ou encore `1.j`, comme ceci :

```
In [ ]: 1j * 1.j
```

Utiliser des variables

Il peut être utile de stocker un résultat qui sera utilisé plus tard, ou de définir une valeur constante. Pour cela on utilise tout simplement une affectation comme ceci :

```
In [ ]: # pour définir une variable il suffit de lui assigner une valeur
        largeur = 5
```

```
In [ ]: # une fois la variable définie, on peut l'utiliser, ici comme un nombre
        largeur * 20
```

```
In [ ]: # après quoi bien sûr la variable reste inchangée
        largeur * 10
```

Pour les symboles mathématiques, on peut utiliser la même technique :

```
In [ ]: # pour définir un réel, on utilise le point au lieu d'une virgule en français
        pi = 3.14159
        2 * pi * 10
```

Pour les valeurs spéciales comme π, on peut utiliser les valeurs prédéfinies par la bibliothèque mathématique de Python. En anticipant un peu sur la notion d'importation que nous approfondirons plus tard, on peut écrire :

```
In [ ]: from math import e, pi
```

Et ainsi imprimer les racines troisièmes de l'unité par la formule :
$r_n = e^{2i\pi\frac{n}{3}}$, pour $n \in \{0,1,2\}$

```
In [ ]: n = 0
        print("n=", n, "racine = ", e**((2.j*pi*n)/3))
        n = 1
        print("n=", n, "racine = ", e**((2.j*pi*n)/3))
        n = 2
        print("n=", n, "racine = ", e**((2.j*pi*n)/3))
```

Remarque : bien entendu il sera possible de faire ceci plus simplement lorsque nous aurons vu les boucles `for`.

Les types

Ce qui change par rapport à une calculatrice standard est le fait que les valeurs sont typées. Pour illustrer les trois types de nombres que nous avons vus jusqu'ici :

```
In [ ]: # le type entier s'appelle 'int'
        type(3)

In [ ]: # le type flottant s'appelle 'float'
        type(3.5)

In [ ]: # le type complexe s'appelle 'complex'
        type(1j)
```

Chaînes de caractères

On a également rapidement besoin de chaînes de caractères, on les étudiera bientôt en détails, mais en guise d'avant-goût :

```
In [ ]: chaine = "Bonjour le monde !"
        print(chaine)
```

Conversions

Il est parfois nécessaire de convertir une donnée d'un type dans un autre. Par exemple on peut demander à l'utilisateur d'entrer une valeur au clavier grâce à la fonction `input`, comme ceci :

```
In [ ]: reponse = input("quel est votre âge ? ")

In [ ]: # vous avez entré la chaîne suivante
        print(reponse)

In [ ]: # ici reponse est une variable, et son contenu est de type chaîne de caractères
        type(reponse)
```

Maintenant je veux faire des calculs sur votre âge, par exemple le multiplier par 2. Si je m'y prends naïvement, ça donne ceci :

```
In [ ]: # multiplier une chaîne de caractères par deux ne fait pas ce que l'on veut,
        # nous verrons plus tard que ça fait une concaténation
        2 * reponse
```

C'est pourquoi il me faut ici d'abord **convertir** la (valeur de la) variable `reponse` en un entier, que je peux ensuite doubler (assurez-vous d'avoir bien entré ci-dessus une valeur qui correspond à un nombre entier)

```
In [ ]: # reponse est une chaine
        # je la convertis en entier en appelant la fonction int()
        age = int(reponse)
        type(age)
```

```
In [ ]: # que je peux maintenant multiplier par 2
        2 * age
```

Ou si on préfère, en une seule fois :

```
In [ ]: print("le double de votre age est", 2*int(reponse))
```

Conversions - suite

De manière plus générale, pour convertir un objet en un entier, un flottant, ou une chaîne de caractères, on peut simplement appeler une fonction *built-in* qui porte le même nom que le type cible :

Type	Fonction
Entier	`int`
Flottant	`float`
Complexe	`complex`
Chaîne	`str`

Ainsi dans l'exemple précédent, `int(reponse)` représente la conversion de `reponse` en entier.

On a illustré cette même technique dans les exemples suivants :

```
In [ ]: # dans l'autre sens, si j'ai un entier
        a = 2345
```

```
In [ ]: # je peux facilement le traduire en chaîne de caractères
        str(2345)
```

```
In [ ]: # ou en complexe
        complex(2345)
```

Nous verrons plus tard que ceci se généralise à tous les types de Python, pour convertir un object `x` en un type `bidule`, on appelle `bidule(x)`. On y reviendra, bien entendu.

Grands nombres

Comme les entiers sont de précision illimitée, on peut améliorer leur lisibilité en insérant des caractères _ qui sont simplement ignorés à l'exécution.

```
In [ ]: tres_grand_nombre = 23_456_789_012_345

        tres_grand_nombre

In [ ]: # ça marche aussi avec les flottants
        123_456.789_012
```

Entiers et bases

Les calculettes scientifiques permettent habituellement d'entrer les entiers dans d'autres bases que la base 10.

En Python, on peut aussi entrer un entier sous forme binaire comme ceci :

```
In [ ]: deux_cents = 0b11001000
        print(deux_cents)
```

Ou encore sous forme octale (en base 8) comme ceci :

```
In [ ]: deux_cents = 0o310
        print(deux_cents)
```

Ou enfin encore en hexadecimal (base 16) comme ceci :

```
In [ ]: deux_cents = 0xc8
        print(deux_cents)
```

Pour d'autres bases, on peut utiliser la fonction de conversion `int` en lui passant un argument supplémentaire :

```
In [ ]: deux_cents = int('3020', 4)
        print(deux_cents)
```

Fonctions mathématiques

Python fournit naturellement un ensemble très complet d'opérateurs mathématiques pour les fonctions exponentielles, trigonométriques et autres, mais leur utilisation ne nous est pas encore accessible à ce stade et nous les verrons ultérieurement.

1.21 Affectations et Opérations (à la +=)

1.21.1 Complément - niveau intermédiaire

Il existe en Python toute une famille d'opérateurs dérivés de l'affectation qui permettent de faire en une fois une opération et une affectation. En voici quelques exemples.

Incrémentation

On peut facilement augmenter la valeur d'une variable numérique comme ceci :

```
In [ ]: entier = 10

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

Comme on le devine peut-être, ceci est équivalent à :

```
In [ ]: entier = 10

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

Autres opérateurs courants

Cette forme, qui combine opération sur une variable et réaffectation du résultat à la même variable, est disponible avec tous les opérateurs courants :

```
In [ ]: entier -= 4
        print('après décrément', entier)
        entier *= 2
        print('après doublement', entier)
        entier /= 2
        print('mis à moitié', entier)
```

Types non numériques

En réalité cette construction est disponible sur tous les types qui supportent l'opérateur en question. Par exemple, les listes (que nous verrons bientôt) peuvent être additionnées entre elles :

```
In [ ]: liste = [0, 3, 5]
        print('liste', liste)

        liste += ['a', 'b']
        print('après ajout', liste)
```

Beaucoup de types supportent l'opérateur +, qui est sans doute de loin celui qui est le plus utilisé avec cette construction.

Opérateurs plus abscons

Signalons enfin que l'on trouve aussi cette construction avec d'autres opérateurs moins fréquents, par exemple :

```
In [ ]: entier = 2
        print('entier:', entier)
        entier **= 10
        print('à la puissance dix:', entier)
        entier %= 5
        print('modulo 5:', entier)
        entier <<= 2
        print('double décalage gauche:', entier)
```

1.22 Notions sur la précision des calculs flottants

1.22.1 Complément - niveau avancé

Le problème

Comme pour les entiers, les calculs sur les flottants sont, naturellement, réalisés par le processeur. Cependant contrairement au cas des entiers où les calculs sont toujours exacts, les flottants posent un problème de précision. Cela n'est pas propre au langage Python, mais est dû à la technique de codage des nombres flottants sous forme binaire.

Voyons tout d'abord comment se matérialise le problème :

```
In [ ]: 0.2 + 0.4
```

Il faut retenir que lorsqu'on écrit un nombre flottant sous forme décimale, la valeur utilisée en mémoire pour représenter ce nombre, parce que cette valeur est codée en binaire, ne représente **pas toujours exactement** le nombre entré.

```
In [ ]: # du coup cette expression est fausse, à cause de l'erreur d'arrondi
        0.3 - 0.1 == 0.2
```

Aussi, comme on le voit, les différentes erreurs d'arrondi qui se produisent à chaque étape du calcul s'accumulent et produisent un résultat parfois surprenant. De nouveau, ce problème n'est pas spécifique à Python, il existe pour tous les langages et il est bien connu des numériciens.

Dans une grande majorité des cas, ces erreurs d'arrondi ne sont pas pénalisantes. Il faut toutefois en être conscient car cela peut expliquer des comportements curieux.

Une solution : penser en termes de nombres rationnels

Tout d'abord si votre problème se pose bien en termes de nombres rationnels, il est alors tout à fait possible de le résoudre avec exactitude.

Alors qu'il n'est pas possible d'écrire exactement $3/10$ en base 2, ni d'ailleurs $1/3$ en base 10, on peut représenter **exactement** ces nombres dès lors qu'on les considère comme des fractions et qu'on les encode avec deux nombres entiers.

Python fournit en standard le module `fractions` qui permet de résoudre le problème. Voici comment on pourrait l'utiliser pour vérifier, cette fois avec succès, que $0.3 - 0.1$ vaut bien 0.2. Ce code anticipe sur l'utilisation des modules et des classes en Python, ici nous créons des objets de type `Fraction` :

```
In [ ]: # on importe le module fractions, qui lui-même définit le symbole Fraction
        from fractions import Fraction

        # et cette fois, les calculs sont exacts, et l'expression retourne bien True
        Fraction(3, 10) - Fraction(1, 10) == Fraction(2, 10)
```

Ou encore d'ailleurs, équivalent et plus lisible :

```
In [ ]: Fraction('0.3') - Fraction('0.1') == Fraction('2/10')
```

Une autre solution : le module decimal

Si par contre vous ne manipulez pas des nombres rationnels et que du coup la représentation sous forme de fractions ne peut pas convenir dans votre cas, signalons l'existence du module standard `decimal` qui offre des fonctionnalités très voisines du type `float`, tout en éliminant la plupart des inconvénients, au prix naturellement d'une consommation mémoire supérieure.

Pour reprendre l'exemple de départ, mais en utilisant le module decimal, on écrirait alors :

```
In [ ]: from decimal import Decimal

        Decimal('0.3') - Decimal('0.1') == Decimal('0.2')
```

Pour aller plus loin

Tous ces documents sont en anglais :

— un tutoriel sur les nombres flottants ;
— la documentation sur la classe Fraction ;
— la documentation sur la classe Decimal.

1.23 Opérations *bit à bit* (*bitwise*)

1.23.1 Compléments - niveau avancé

Les compléments ci-dessous expliquent des fonctions évoluées sur les entiers. Les débutants en programmation peuvent sans souci sauter cette partie en cas de difficultés.

Opérations logiques : ET `&`, OU `|` et OU exclusif `^`

Il est possible aussi de faire des opérations *bit à bit* sur les nombres entiers. Le plus simple est de penser à l'écriture du nombre en base 2.

Considérons par exemple deux entiers constants dans cet exercice

```
In [ ]: x49 = 49
        y81 = 81
```

Ce qui nous donne comme décomposition binaire :

$x49 = 49 = 32+16+1 \rightarrow (0,1,1,0,0,0,1)$
$y81 = 81 = 64+16+1 \rightarrow (1,0,1,0,0,0,1)$

Pour comprendre comment passer de $32+16+1$ à $(0,1,1,0,0,0,1)$ il suffit d'observer que :
$32+16+1 = \mathbf{0}*2^6 + \mathbf{1}*2^5 + \mathbf{1}*2^4 + \mathbf{0}*2^3 + \mathbf{0}*2^2 + \mathbf{0}*2^1 + \mathbf{1}*2^0$

Et logique : opérateur `&` L'opération logique `&` va faire un *et* logique bit à bit entre les opérandes, ainsi

```
In [ ]: x49 & y81
```

Et en effet :

$x49 \rightarrow (0,1,1,0,0,0,1)$
$y81 \rightarrow (1,0,1,0,0,0,1)$
$x49 \,\&\, y81 \rightarrow (0,0,1,0,0,0,1) \rightarrow 16+1 \rightarrow 17$

Ou logique : opérateur | De même, l'opérateur logique | fait simplement un *ou* logique, comme ceci :

```
In [ ]: x49 | y81
```

On s'y retrouve parce que :

$$\begin{array}{rcl} x49 & \to & (0,1,1,0,0,0,1) \\ y81 & \to & (1,0,1,0,0,0,1) \\ x49 \mid y81 & \to & (1,1,1,0,0,0,1) \to 64+32+16+1 \to 113 \end{array}$$

Ou exclusif : opérateur ^ Enfin, on peut également faire la même opération à base de *ou exclusif* avec l'opérateur ^ :

```
In [ ]: x49 ^ y81
```

Je vous laisse le soin de décortiquer le calcul à titre d'exercice (le ou exclusif de deux bits est vrai si et seulement si exactement une des deux entrées est vraie).

Décalages

Un décalage *à gauche* de, par exemple, 4 positions, revient à décaler tout le champ de bits de 4 cases à gauche (les 4 nouveaux bits insérés sont toujours des 0). C'est donc équivalent à une multiplication par $2^4 = 16$:

```
In [ ]: x49 << 4
```

$$\begin{array}{rcl} x49 & \to & (0,1,1,0,0,0,1) \\ x49 << 4 & \to & (0,1,1,0,0,0,1,0,0,0,0) \to 512+256+16 \to 784 \end{array}$$

De la même façon le décalage à droite de n revient à une division par 2^n (plus précisément, le quotient de la division) :

```
In [ ]: x49 >> 4
```

$$\begin{array}{rcl} x49 & \to & (0,1,1,0,0,0,1) \\ x49 >> 4 & \to & (0,0,0,0,0,1,1) \to 2+1 \to 3 \end{array}$$

Une astuce

On peut utiliser la fonction *built-in* `bin` pour calculer la représentation binaire d'un entier, attention la valeur de retour est une chaîne de caractères de type `str` :

```
In [ ]: bin(x49)
```

Dans l'autre sens, on peut aussi entrer un entier directement en base 2 comme ceci, ici comme on le voit `x49bis` est bien un entier :

```
In [ ]: x49bis = 0b110001
        x49bis == x49
```

Pour en savoir plus

Section de la documentation Python.

1.24 Estimer le plus petit (grand) flottant

1.24.1 Exercice - niveau basique

Le plus petit flottant

En corollaire de la discussion sur la précision des flottants, il faut savoir que le système de codage en mémoire impose aussi une limite. Les réels très petits, ou très grands, ne peuvent plus être représentés de cette manière.

C'est notamment très gênant si vous implémentez un logiciel probabiliste, comme des graphes de Markov, où les probabilités d'occurrence de séquences très longues tendent très rapidement vers des valeurs extrêmement petites.

Le but de cet exercice est d'estimer la valeur du plus petit flottant qui peut être représenté comme un flottant. Pour vous aider, voici deux valeurs :

```
In [ ]: 10**-320
```

```
In [ ]: 10**-330
```

Comme on le voit, 10^{-320} est correctement imprimé, alors que 10^{-330} est, de manière erronée, rapporté comme étant nul.

Notes :

— À ce stade du cours, pour estimer le plus petit flottant, procédez simplement par approximations successives.

— Sans utiliser de boucle, la précision que vous pourrez obtenir n'est que fonction de votre patience, ne dépassez pas 4 à 5 itérations successives :)

— Il est par contre pertinent d'utiliser une approche rationnelle pour déterminer l'itération suivante (par opposition à une approche "au petit bonheur"). Pour ceux qui ne connaissent pas, nous vous recommandons de vous documenter sur l'algorithme de **dichotomie**.

```
In [ ]: 10**-325
```

Voici quelques cellules de code vides, vous pouvez en créer d'autres si nécessaire, le plus simple étant de taper `Alt+Enter`, ou d'utiliser le menu *"Insert -> Insert Cell Below"*

```
In [ ]: # vos essais successifs ici
```

```
In [ ]: .24*10**-323
```

Le plus grand flottant

La même limitation s'applique sur les grands nombres. Toutefois cela est un peu moins évident, car comme toujours il faut faire attention aux types :

```
In [ ]: 10**450
```

Ce qui passe très bien car j'ai utilisé un `int` pour l'exposant, dans ce premier cas Python calcule le résultat comme un `int`, qui est un type qui n'a pas de limitation de précision (Python utilise intelligemment autant de bits que nécessaire pour ce genre de calculs).

Par contre, si j'essaie de faire le même calcul avec un exposant flottant, Python essaie cette fois de faire son calcul avec un flottant, et là on obtient une erreur :

```
In [ ]: 10**450.0
```

On peut d'ailleurs remarquer que le comportement ici n'est pas extrêmement cohérent, car avec les petits nombres Python nous a silencieusement transformé 10^{-330} en 0, alors que pour les grands nombres, il lève une exception (nous verrons les exceptions plus tard, mais vous pouvez dès maintenant remarquer que le comportement est différent dans les deux cas).

Quoi qu'il en soit, la limite pour les grands nombres se situe entre les deux valeurs 10^{300} et 10^{310}. On vous demande à nouveau d'estimer comme ci-dessus une valeur approchée du plus grand nombre qu'il soit possible de représenter comme un flottant.

```
In [ ]: 10**300.
```

```
In [ ]: 10**310.
```

```
In [ ]: # vos essais successifs ici
```

1.24.2 Complément - niveau avancé

En fait, on peut accéder à ces valeurs minimales et maximales pour les flottants comme ceci

```
In [ ]: import sys
        print(sys.float_info)
```

Et notamment, comme expliqué ici.

```
In [ ]: print("Flottant minimum", sys.float_info.min)
        print("Flottant maximum", sys.float_info.max)
```

Sauf que vous devez avoir trouvé un maximum voisin de cette valeur, mais le minimum observé expérimentalement ne correspond pas bien à cette valeur.

Pour ceux que cela intéresse, l'explication à cette apparente contradiction réside dans l'utilisation de nombres dénormaux.

SEMAINE 2

NOTIONS DE BASE POUR ÉCRIRE SON PREMIER PROGRAMME EN PYTHON

2.1 Les outils de base sur les chaînes de caractères (`str`)

2.1.1 Complément - niveau intermédiaire

Lire la documentation

Même après des années de pratique, il est difficile de se souvenir de toutes les méthodes travaillant sur les chaînes de caractères. Aussi il est toujours utile de recourir à la documentation embarquée

```
In [ ]: help(str)
```

Nous allons tenter ici de citer les méthodes les plus utilisées. Nous n'avons le temps que de les utiliser de manière très simple, mais bien souvent il est possible de passer en argument des options permettant de ne travailler que sur une sous-chaîne, ou sur la première ou dernière occurrence d'une sous-chaîne. Nous vous renvoyons à la documentation pour obtenir toutes les précisions utiles.

Découpage - assemblage : `split` et `join`

Les méthodes `split` et `join` permettent de découper une chaîne selon un séparateur pour obtenir une liste, et à l'inverse de reconstruire une chaîne à partir d'une liste.

`split` permet donc de découper :

```
In [ ]: 'abc=:=def=:=ghi=:=jkl'.split('=:=')
```

Et à l'inverse :

```
In [ ]: "=:=".join(['abc', 'def', 'ghi', 'jkl'])
```

Attention toutefois si le séparateur est un terminateur, la liste résultat contient alors une dernière chaîne vide. En pratique, on utilisera la méthode `strip`, que nous allons voir ci-dessous, avant la méthode `split` pour éviter ce problème.

```
In [ ]: 'abc;def;ghi;jkl;'.split(';')
```

Qui s'inverse correctement cependant :

```
In [ ]: ";".join(['abc', 'def', 'ghi', 'jkl', ''])
```

Remplacement : `replace`

`replace` est très pratique pour remplacer une sous-chaîne par une autre, avec une limite éventuelle sur le nombre de remplacements :

```
In [ ]: "abcdefabcdefabcdef".replace("abc", "zoo")
```

```
In [ ]: "abcdefabcdefabcdef".replace("abc", "zoo", 2)
```

Plusieurs appels à `replace` peuvent être chaînés comme ceci :

```
In [ ]: "les [x] qui disent [y]".replace("[x]", "chevaliers").replace("[y]", "Ni")
```

Nettoyage : `strip`

On pourrait par exemple utiliser `replace` pour enlever les espaces dans une chaîne, ce qui peut être utile pour "nettoyer" comme ceci :

```
In [ ]: " abc:def:ghi ".replace(" ", "")
```

Toutefois bien souvent on préfère utiliser `strip` qui ne s'occupe que du début et de la fin de la chaîne, et gère aussi les tabulations et autres retour à la ligne :

```
In [ ]: " \tune chaîne avec des trucs qui dépassent \n".strip()
```

On peut appliquer `strip` avant `split` pour éviter le problème du dernier élément vide :

```
In [ ]: 'abc;def;ghi;jkl;'.strip(';').split(';')
```

Rechercher une sous-chaîne

Plusieurs outils permettent de chercher une sous-chaîne. Il existe `find` qui renvoie le plus petit index où on trouve la sous-chaîne :

```
In [ ]: # l'indice du début de la première occurrence
        "abcdefcdefghefghijk".find("def")
```

```
In [ ]: # ou -1 si la chaîne n'est pas présente
        "abcdefcdefghefghijk".find("zoo")
```

`rfind` fonctionne comme `find` mais en partant de la fin de la chaîne :

```
In [ ]: # en partant de la fin
        "abcdefcdefghefghijk".rfind("fgh")
```

```
In [ ]: # notez que le résultat correspond
        # tout de même toujours au début de la chaîne
        "abcdefcdefghefghijk"[13]
```

La méthode `index` se comporte comme `find`, mais en cas d'absence elle lève une **exception** (nous verrons ce concept plus tard) plutôt que de renvoyer `-1` :

```
In [ ]: "abcdefcdefghefghijk".index("def")
```

```
In [ ]: try:
            "abcdefcdefghefghijk".index("zoo")
        except Exception as e:
            print("OOPS", type(e), e)
```

Mais le plus simple pour chercher si une sous-chaîne est dans une autre chaîne est d'utiliser l'instruction `in` sur laquelle nous reviendrons lorsque nous parlerons des séquences :

```
In [ ]: "def" in "abcdefcdefghefghijk"
```

La méthode `count` compte le nombre d'occurrences d'une sous-chaîne :

```
In [ ]: "abcdefcdefghefghijk".count("ef")
```

Signalons enfin les méthodes de commodité suivantes :

```
In [ ]: "abcdefcdefghefghijk".startswith("abcd")
```

```
In [ ]: "abcdefcdefghefghijk".endswith("ghijk")
```

S'agissant des deux dernières, remarquons que :
`chaine.startswith(sous_chaine)` $\Longleftrightarrow$ `chaine.find(sous_chaine) == 0`
`chaine.endswith(sous_chaine)` $\Longleftrightarrow$ `chaine.rfind(sous_chaine) == (len(chaine) - len(sous_chaine))`
On remarque ici la supériorité en terme d'expressivité des méthodes pythoniques `startswith` et `endswith`.

Changement de casse

Voici pour conclure quelques méthodes utiles qui parlent d'elles-mêmes :

```
In [ ]: "monty PYTHON".upper()
```

```
In [ ]: "monty PYTHON".lower()
```

```
In [ ]: "monty PYTHON".swapcase()
```

```
In [ ]: "monty PYTHON".capitalize()
```

```
In [ ]: "monty PYTHON".title()
```

Pour en savoir plus

Tous ces outils sont documentés en détail ici (en anglais).

2.2 Formatage de chaînes de caractères

2.2.1 Complément - niveau basique

On désigne par formatage les outils qui permettent d'obtenir une présentation fine des résultats, que ce soit pour améliorer la lisibilité lorsqu'on s'adresse à des humains, ou pour respecter la syntaxe d'un outil auquel on veut passer les données pour un traitement ultérieur.

La fonction `print`

Nous avons jusqu'à maintenant presque toujours utilisé la fonction `print` pour afficher nos résultats. Comme on l'a vu, celle-ci réalise un formatage sommaire : elle insère un espace entre les valeurs qui lui sont passées.

```
In [ ]: print(1, 'a', 12 + 4j)
```

La seule subtilité notable concernant `print` est que, par défaut, elle ajoute un saut de ligne à la fin. Pour éviter ce comportement, on peut passer à la fonction un argument `end`, qui sera inséré *au lieu* du saut de ligne. Ainsi par exemple :

```
In [ ]: # une première ligne
        print("une", "seule", "ligne")

In [ ]: # une deuxième ligne en deux appels à print
        print("une", "autre", end=' ')
        print("ligne")
```

Il faut remarquer aussi que `print` est capable d'imprimer **n'importe quel objet**. Nous l'avons déjà fait avec les listes et les tuples, voici par exemple un module :

```
In [ ]: # on peut imprimer par exemple un objet 'module'
        import math

        print('le module math est', math)
```

En anticipant un peu, voici comment `print` présente les instances de classe (ne vous inquiétez pas, nous apprendrons dans une semaine ultérieure ce que sont les classes et les instances).

```
In [ ]: # pour définir la classe Personne
        class Personne:
            pass

        # et pour créer une instance de cette classe
        personne = Personne()

In [ ]: # voila comment s'affiche une instance de classe
        print(personne)
```

On rencontre assez vite les limites de `print` :

— d'une part, il peut être nécessaire de formater une chaîne de caractères sans nécessairement vouloir l'imprimer, ou en tous cas pas immédiatement ;
— d'autre part, les espaces ajoutés peuvent être plus néfastes qu'utiles ;
— enfin, on peut avoir besoin de préciser un nombre de chiffres significatifs, ou de choisir comment présenter un date.

C'est pourquoi il est plus courant de **formatter** les chaines - c'est à dire de calculer des chaines en mémoire, sans nécessairement les imprimer de suite, et c'est ce que nous allons étudier dans ce complément.

Les *f-strings*

Depuis la version 3.6 de Python, on peut utiliser les f-strings, le premier mécanisme de formatage que nous étudierons. C'est le mécanisme de formatage le plus simple et le plus agréable à utiliser.

Je vous recommande tout de même de lire les sections à propos de `format` et de `%`, qui sont encore massivement utilisées dans le code existant (surtout `%` d'ailleurs, bien que essentiellement obsolète).

Mais définissons d'abord quelques données à afficher :

```
In [ ]: # donnons-nous quelques variables
        prenom, nom, age = 'Jean', 'Dupont', 35

In [ ]: # mon premier f-string
        f"{prenom} {nom} a {age} ans"
```

Vous remarquez d'abord que le string commence par `f"`, c'est bien sûr pour cela qu'on l'apelle un *f-string*.

On peut bien entendu ajouter le `f` devant toutes les formes de strings, qu'ils commencent par `'` ou `"` ou `'''` ou `"""`.

Ensuite vous remarquez que les zones délimitées entre `{}` sont remplacées. La logique d'un *f-string*, c'est tout simplement de considérer l'intérieur d'un `{}` comme du code Python (une expression pour être précis), de l'évaluer, et d'utiliser le résultat pour remplir le `{}`.

Ça veut dire, en clair, que je peux faire des calculs à l'intérieur des `{}`.

```
In [ ]: # toutes les expressions sont autorisées à l'intérieur d'un {}
        f"dans 10 ans {prenom} aura {age + 10} ans"

In [ ]: # on peut donc aussi mettre des appels de fonction
        notes = [12, 15, 19]
        f"nous avons pour l'instant {len(notes)} notes"
```

Nous allons en rester là pour la partie en niveau basique. Il nous reste à étudier comment chaque `{}` est formatté (par exemple comment choisir le nombre de chiffres significatifs sur un flottant), voyez plus bas pour plus de détails sur ce point.

Comme vous le voyez, les *f-strings* fournissent une méthode très simple et expressive pour formatter des données dans des chaines de caractère. Redisons-le pour être bien clair : un *f-string* **ne réalise pas d'impression**, il faut donc le passer à `print` si l'impression est souhaitée.

La méthode `format`

Avant l'introduction des *f-strings*, la technique recommandée pour faire du formattage était d'utiliser la méthode `format` qui est définie sur les objets `str` et qui s'utilise comme ceci :

```
In [ ]: "{} {} a {} ans".format(prenom, nom, age)
```

Dans cet exemple le plus simple, les données sont affichées en lieu et place des `{}`, dans l'ordre où elles sont fournies.

Cela convient bien lorsqu'on a peu de données. Si par la suite on veut changer l'ordre par exemple des nom et prénom, on peut bien sûr échanger l'ordre des arguments passés à format, ou encore utiliser la **liaison par position**, comme ceci :

```
In [ ]: "{1} {0} a {2} ans".format(prenom, nom, age)
```

Dans la pratique toutefois, cette forme est assez peu utile, on lui préfère souvent la **liaison par nom** qui se présente comme ceci :

```
In [ ]: "{le_prenom} {le_nom} a {l_age} ans".format(le_nom=nom, le_prenom=prenom, l_age=age)
```

Dans ce premier exemple de liaison par nom, nous avons délibérément utilisé des noms différents pour les données externes et pour les noms apparaissant dans le format, pour bien illustrer comment la liaison est résolue, mais on peut aussi bien faire tout simplement :

```
In [ ]: "{prenom} {nom} a {age} ans".format(nom=nom, prenom=prenom, age=age)
```

Voici qui conclut notre courte introduction à la méthode `format`.

2.2.2 Complément - niveau intermédiaire

La toute première version du formatage : l'opérateur %

`format` a été en fait introduite assez tard dans Python, pour remplacer la technique que nous allons présenter maintenant.

Étant donné le volume de code qui a été écrit avec l'opérateur %, il nous a semblé important d'introduire brièvement cette construction ici. Vous ne devez cependant pas utiliser cet opérateur dans du code moderne, la manière pythonique de formatter les chaînes de caractères est le f-string.

Le principe de l'opérateur % est le suivant. On élabore comme ci-dessus un "format" c'est-à-dire le patron de ce qui doit être rendu, auquel on passe des arguments pour "remplir" les trous. Voyons les exemples de tout à l'heure rendus avec l'opérateur % :

```
In [ ]: # l'ancienne façon de formatter les chaînes avec %
        # est souvent moins lisible
        "%s %s a %s ans" % (prenom, nom, age)
```

On pouvait également avec cet opérateur recourir à un mécanisme de liaison par nommage, en passant par un dictionnaire. Pour anticiper un tout petit peu sur cette notion que nous verrons très bientôt, voici comment

```
In [ ]: variables = {'le_nom': nom, 'le_prenom': prenom, 'l_age': age}
        "%(le_nom)s, %(le_prenom)s, %(l_age)s ans" % variables
```

2.2.3 Complément - niveau avancé

De retour aux *f-strings* et à la fonction `format`, il arrive qu'on ait besoin de spécifier plus finement la façon dont une valeur doit être affichée.

Précision des arrondis

C'est typiquement le cas avec les valeurs flottantes pour lesquelles la précision de l'affichage vient au détriment de la lisibilité. Voici deux formes équivalentes pour obtenir une valeur de pi arrondie :

```
In [ ]: from math import pi

In [ ]: # un f-string
        f"pi avec seulement 2 chiffres apres la virgule {pi:.2f}"
```

```
In [ ]: # avec format() et liaison par nom
        "pi avec seulement 2 chiffres apres la virgule {flottant:.2f}".format(flottant=pi)
```

Dans ces deux exemples, la partie à l'intérieur des `{}` et à droite du `:` s'appelle le format, ici `.2f` ; vous remarquez que c'est le même pour les *f-strings* et pour `format`, et c'est toujours le cas. C'est pourquoi on ne verra plus à partir d'ici que des exemples avec les *f-strings*.

`0` en début de nombre

Pour forcer un petit entier à s'afficher sur 4 caractères, avec des `0` ajoutés au début si nécessaire :

```
In [ ]: x = 15

        f"{x:04d}"
```

Ici on utilise le format `d` (toutes ces lettres `d`, `f`, `g` viennent des formats ancestraux de la libc comme `printf`). Ici avec `04d` on précise qu'on veut une sortie sur 4 caractères et qu'il faut remplir à gauche si nécessaire avec des `0`.

Largeur fixe

Dans certains cas, on a besoin d'afficher des données en colonnes de largeurs fixes, on utilise pour cela les formats `<` `^` et `>` pour afficher à gauche, au centre, ou à droite d'une zone de largeur fixe :

```
In [ ]: # les données à afficher
        comptes = [
         ('Apollin', 'Dupont', 127),
         ('Myrtille', 'Lamartine', 25432),
         ('Prune', 'Soc', 827465),
        ]

        for prenom, nom, solde in comptes:
            print(f"{prenom:<10} -- {nom:^12} -- {solde:>8} €")
```

Voir aussi

Nous vous invitons à vous reporter à la documentation de `format` pour plus de détails sur les formats disponibles, et notamment aux nombreux exemples qui y figurent.

2.3 Obtenir une réponse de l'utilisateur

2.3.1 Complément - niveau basique

Occasionnellement, il peut être utile de poser une question à l'utilisateur.

La fonction `input`

C'est le propos de la fonction `input`. Par exemple :

```
In [ ]: nom_ville = input("Entrez le nom de la ville : ")
        print(f"nom_ville={nom_ville}")
```

Attention à bien vérifier/convertir

Notez bien que `input` renvoie **toujours une chaîne de caractère** (`str`). C'est assez évident, mais il est très facile de l'oublier et de passer cette chaîne directement à une fonction qui s'attend à recevoir, par exemple, un nombre entier, auquel cas les choses se passent mal :

```
>>> input("nombre de lignes ? ") + 3
nombre de lignes ? 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

Dans ce cas il faut appeler la fonction `int` pour convertir le résultat en un entier :

```
In [ ]: int(input("Nombre de lignes ? ")) + 3
```

Limitations

Cette fonction peut être utile pour vos premiers pas en Python.

En pratique toutefois, on utilise assez peu cette fonction, car les applications "réelles" viennent avec leur propre interface utilisateur, souvent graphique, et disposent donc d'autres moyens que celui-ci pour interagir avec l'utilisateur.

Les applications destinées à fonctionner dans un terminal, quant à elles, reçoivent traditionnellement leurs données de la ligne de commande. C'est le propos du module `argparse` que nous avons déjà rencontré en première semaine.

2.4 Expressions régulières et le module `re`

2.4.1 Complément - niveau intermédiaire

Une expression régulière est un objet mathématique permettant de décrire un ensemble de textes qui possèdent des propriétés communes. Par exemple, s'il vous arrive d'utiliser un terminal, et que vous tapez :

```
$ dir *.txt
```

(ou `ls *.txt` sur Linux ou macOS), vous utilisez l'expression régulière `*.txt` qui désigne tous les fichiers dont le nom se termine par `.txt`. On dit que l'expression régulière *filtre* toutes les chaînes qui se terminent par `.txt` (l'expression anglaise consacrée est le *pattern matching*).

Le langage Perl a été le premier à populariser l'utilisation des expressions régulières en les supportant nativement dans le langage, et non au travers d'une bibliothèque.

En Python, les expressions régulières sont disponibles de manière plus traditionnelle, via le module `re` de la bibliothèque standard, que nous allons voir maintenant.

Dans la commande ci-dessus, `*.txt` est une expression régulière très simple. Le module `re` fournit le moyen de construire des expressions régulières très élaborées et plus puissantes que ce que supporte le terminal. C'est pourquoi la syntaxe des regexps de `re` est un peu différente. Par exemple, pour filtrer la même famille de chaînes que `*.txt` avec le module `re`, il nous faudra écrire l'expression régulière sous une forme légèrement différente.

Le propos de ce complément est de vous donner une première introduction au module `re`.

```
In [ ]: import re
```

Je vous conseille d'avoir sous la main la documentation du module `re` pendant que vous lisez ce complément.

Un exemple simple

`findall` On se donne deux exemples de chaînes :

```
In [ ]: sentences = ['Lacus a donec, vitae gravida proin sociis.',
                     'Neque ipsum! rhoncus cras quam.']
```

On peut **chercher tous** les mots se terminant par `a` ou `m` dans une chaîne avec `findall` :

```
In [ ]: for sentence in sentences:
            print(f"---- dans >{sentence}<")
            print(re.findall(r"\w*[am]\W", sentence))
```

Ce code permet de chercher toutes (`findall`) les occurrences de l'expression régulière, qui ici est définie par le *raw-string* :

```
r"\w*[am]\W"
```

Nous verrons tout à l'heure comment fabriquer des expressions régulières plus en détail, mais pour démystifier au moins celle-ci, on a mis bout à bout les morceaux suivants :

— `\w*` : on veut trouver une sous-chaîne qui commence par un nombre quelconque, y compris nul (`*`) de caractères alphanumériques (`\w`) ;
— `[am]` : immédiatement après, il nous faut trouver un caractère `a` ou `m` ;
— `\W` : et enfin, il nous faut un caractère qui ne soit **pas** alphanumérique. Ceci est important puisqu'on cherche les mots qui **se terminent** par un `a` ou un `m`, si on ne le mettait pas on obtiendrait ceci :

```
In [ ]: # le \W final est important
        # voici ce qu'on obtient si on l'omet
        for sentence in sentences:
            print(f"---- dans >{sentence}<")
            print(re.findall(r"\w*[am]", sentence))
```

`split` Une autre forme simple d'utilisation des regexps est `re.split`, qui fournit une fonctionnalité voisine de `str.split`, mais ou les séparateurs sont exprimés comme une expression régulière :

```
In [ ]: for sentence in sentences:
            print(f"---- dans >{sentence}<")
            print(re.split(r"\W+", sentence))
            print()
```

Ici, l'expression régulière qui bien sûr décrit le séparateur, est simplement `\W+` c'est-à-dire toute suite d'au moins un caractère non alphanumérique.

Nous avons donc là un moyen simple, et plus puissant que `str.split`, de couper un texte en mots.

`sub` Une troisième méthode utilitaire est `re.sub` qui permet de remplacer les occurrences d'une *regexp*, comme par exemple :

```
In [ ]: for sentence in sentences:
            print(f"---- dans >{sentence}<")
            print(re.sub(r"(\w+)", r"X\1Y", sentence))
            print()
```

Ici, l'expression régulière (le premier argument) contient un **groupe** : on a utilisé des parenthèses autour du `\w+`. Le second argument est la chaîne de remplacement, dans laquelle on a fait **référence au groupe** en écrivant `\1`, qui veut dire tout simplement "le premier groupe".

Donc au final, l'effet de cet appel est d'entourer toutes les suites de caractères alphanumériques par `X` et `Y`.

Pourquoi un *raw-string* ? En guise de digression, il n'y a aucune obligation à utiliser un *raw-string*, d'ailleurs on rappelle qu'il n'y a pas de différence de nature entre un *raw-string* et une chaîne usuelle :

```
In [ ]: raw = r'abc'
        regular = 'abc'
        # comme on a pris une 'petite' chaîne ce sont les mêmes objets
        print(f"both compared with is → {raw is regular}")
        # et donc a fortiori
        print(f"both compared with == → {raw == regular}")
```

Il se trouve que le *backslash* \ à l'intérieur des expressions régulières est d'un usage assez courant - on l'a vu déjà plusieurs fois. C'est pourquoi on **utilise fréquemment un *raw-string*** pour décrire une expression régulière, et en général à chaque fois qu'elle comporte un *backslash*. On rappelle que le raw-string désactive l'interprétation des \ à l'intérieur de la chaîne, par exemple, `\t` est interprété comme un caractère de tabulation. Sans raw-string, il faut doubler tous les \ pour qu'il n'y ait pas d'interprétation.

Un deuxième exemple

Nous allons maintenant voir comment on peut d'abord vérifier si une chaîne est conforme au critère défini par l'expression régulière, mais aussi *extraire* les morceaux de la chaîne qui correspondent aux différentes parties de l'expression.

Pour cela, supposons qu'on s'intéresse aux chaînes qui comportent 5 parties, une suite de chiffres, une suite de lettres, des chiffres à nouveau, des lettres et enfin de nouveau des chiffres.

Pour cela on considère ces trois chaînes en entrée :

```
In [ ]: samples = ['890hj000nnm890',    # cette entrée convient
                  '123abc456def789',    # celle-ci aussi
                  '8090abababab879',    # celle-ci non
                 ]
```

`match` Pour commencer, voyons que l'on peut facilement **vérifier si une chaîne vérifie** ou non le critère :

```
In [ ]: regexp1 = "[0-9]+[A-Za-z]+[0-9]+[A-Za-z]+[0-9]+"
```

Si on applique cette expression régulière à toutes nos entrées :

```
In [ ]: for sample in samples:
            match = re.match(regexp1, sample)
            print(f"{sample:16s} → {match}")
```

Pour rendre ce résultat un peu plus lisible nous nous définissons une petite fonction de confort :

```
In [ ]: # pour simplement visualiser si on a
        # une correspondance (match) ou pas
        def nice(match):
            # le retour de re.match est soit None, soit un objet match
            return "no" if match is None else "Match!"
```

Avec quoi on peut refaire l'essai sur toutes nos entrées :

```
In [ ]: # la même chose mais un peu moins encombrant
        print(f"REGEXP={regexp1}\n")
        for sample in samples:
            match = re.match(regexp1, sample)
            print(f"{sample:>16s} → {nice(match)}")
```

Ici plutôt que d'utiliser les raccourcis comme `\w` j'ai préféré écrire explicitement les ensembles de caractères en jeu. De cette façon, on limite le nombre de caractères pouvant correspondre car, par défaut en Python 3, `\w` correspondra à tout caractère Unicode considéré comme alphanumérique (par exemple à, é, è, ß, ø, etc.). Il y a deux morceaux qui interviennent tour à tour : * `[0-9]+` signifie une suite d'au moins un caractère dans l'intervalle `[0-9]` ; * `[A-Za-z]+` pour une suite d'au moins un caractère dans l'intervalle `[A-Z]` ou dans l'intervalle `[a-z]`.

Et comme tout à l'heure on a simplement juxtaposé les morceaux dans le bon ordre pour construire l'expression régulière complète.

Nommer un morceau (un groupe)

```
In [ ]: # on se concentre sur une entrée correcte
        haystack = samples[1]
        haystack
```

Maintenant, on va même pouvoir **donner un nom** à un morceau de la regexp, ici on désigne par `needle` le groupe de chiffres du milieu :

```
In [ ]: # la même regexp, mais on donne un nom au groupe de chiffres central
        regexp2 = "[0-9]+[A-Za-z]+(?P<needle>[0-9]+)[A-Za-z]+[0-9]+"
```

Et une fois que c'est fait, on peut demander à l'outil de nous **retrouver la partie correspondante** dans la chaîne initiale :

```
In [ ]: print(re.match(regexp2, haystack).group('needle'))
```

Dans cette expression on a utilisé un **groupe nommé** `(?P<needle>[0-9]+)`, dans lequel :

— les parenthèses définissent un groupe ;
— `?P<needle>` spécifie que ce groupe pourra être référencé sous le nom `needle` (cette syntaxe très absconse est héritée semble-t-il de Perl).

Un troisième exemple

Enfin, et c'est un trait qui n'est pas présent dans tous les langages, on peut restreindre un morceau de chaîne à être identique à un groupe déjà vu plus tôt dans la chaîne. Dans l'exemple ci-dessus, on pourrait ajouter comme contrainte que le premier et le dernier groupes de chiffres soient identiques, comme ceci :

```
In [ ]: regexp3 = "(?P<id>[0-9]+)[A-Za-z]+(?P<needle>[0-9]+)[A-Za-z]+(?P=id)"
```

Si bien que maintenant, avec les mêmes entrées que tout à l'heure :

```
In [ ]: print(f"REGEXP={regexp3}\n")
        for sample in samples:
            match = re.match(regexp3, sample)
            print(f"{sample:>16s} → {nice(match)}")
```

Comme précédemment, on a défini le groupe nommé `id` comme étant la première suite de chiffres. La nouveauté ici est la **contrainte** qu'on a imposée sur le dernier groupe avec `(?P=id)`. Comme vous le voyez, on n'obtient un *match* qu'avec les entrées dans lesquelles le dernier groupe de chiffres est identique au premier.

Comment utiliser la bibliothèque

Avant d'apprendre à écrire une expression régulière, disons quelques mots du mode d'emploi de la bibliothèque.

Fonctions de commodité et *workflow* Comme vous le savez peut-être, une expression régulière décrite sous forme de chaîne, comme par exemple `"\w*[am]\W"`, peut être traduite dans un **automate fini** qui permet de faire le filtrage avec une chaîne. C'est ce qui explique le *workflow* que nous avons résumé dans cette figure.

La méthode recommandée pour utiliser la bibliothèque, lorsque vous avez le même *motif* à appliquer à un grand nombre de chaînes, est de : * compiler **une seule fois** votre chaîne en un automate, qui est matérialisé par un objet de la classe `re.RegexObject`, en utilisant `re.compile` ; * puis d'**utiliser directement cet objet** autant de fois que vous avez de chaînes.

Nous avons utilisé dans les exemples plus haut (et nous continuerons plus bas pour une meilleure lisibilité) des **fonctions de commodité** du module, qui sont pratiques, par exemple, pour mettre au point une expression régulière en mode interactif, mais qui ne **sont pas forcément** adaptées dans tous les cas.

Ces fonctions de commodité fonctionnent toutes sur le même principe :
`re.match(regexp, sample)` $\Longleftrightarrow$ `re.compile(regexp).match(sample)`
Donc, à chaque fois que l'on utilise une fonction de commodité, on recompile la chaîne en automate, ce qui, dès qu'on a plus d'une chaîne à traiter, représente un surcoût.

```
In [ ]: # au lieu de faire comme ci-dessus :

        # imaginez 10**6 chaînes dans samples
        for sample in samples:
            match = re.match(regexp3, sample)
            print(f"{sample:>16s} → {nice(match)}")
```

```
In [ ]: # dans du vrai code on fera plutôt :

        # on compile la chaîne en automate une seule fois
        re_obj3 = re.compile(regexp3)

        # ensuite, on part directement de l'automate
        for sample in samples:
            match = re_obj3.match(sample)
            print(f"{sample:>16s} → {nice(match)}")
```

Cette deuxième version ne compile qu'une fois la chaîne en automate, et est donc plus efficace.

Les méthodes sur la classe `RegexObject` Les objets de la classe `RegexObject` représentent donc l'automate à état fini qui est le résultat de la compilation de l'expression régulière. Pour résumer ce qu'on a déjà vu, les méthodes les plus utiles sur un objet `RegexObject` sont : * `match` et `search`, qui cherchent un *match* soit uniquement au début (`match`) ou n'importe où dans la chaîne (`search`) ; * `findall` et `split` pour chercher toutes les occurences (`findall`) ou leur négatif (`split`) ; * `sub` (qui aurait pu sans doute s'appeler `replace`, mais c'est comme ça) pour remplacer les occurrences de pattern.

Exploiter le résultat Les **méthodes** disponibles sur la classe `re.MatchObject` sont documentées en détail ici. On en a déjà rencontré quelques-unes, en voici à nouveau un aperçu rapide :

```
In [ ]: # exemple
        sample = "    Isaac Newton, physicist"
        match = re.search(r"(\w+) (?P<name>\w+)", sample)
```

`re` et `string` pour retrouver les données d'entrée du match :

```
In [ ]: match.string
```

```
In [ ]: match.re
```

`group`, `groups`, `groupdict` pour retrouver les morceaux de la chaîne d'entrée qui correspondent aux **groupes** de la regexp. On peut y accéder par rang, ou par nom (comme on l'a vu plus haut avec `needle`) :

```
In [ ]: match.groups()
```

```
In [ ]: match.group(1)
```

```
In [ ]: match.group('name')
```

```
In [ ]: match.group(2)
```

```
In [ ]: match.groupdict()
```

Comme on le voit pour l'accès par rang **les indices commencent à 1** pour des raisons historiques (on peut déjà référencer `\1` en sed depuis la fin des années 70).

On peut aussi accéder au **groupe 0** comme étant la partie de la chaîne de départ qui a effectivement été filtrée par l'expression régulière, et qui peut tout à fait être au beau milieu de la chaîne de départ, comme dans notre exemple :

```
In [ ]: match.group(0)
```

`expand` permet de faire une espèce de `str.format` avec les valeurs des groupes :

```
In [ ]: match.expand(r"last_name \g<name> first_name \1")
```

`span` pour connaître les index dans la chaîne d'entrée pour un groupe donné :

```
In [ ]: begin, end = match.span('name')
        sample[begin:end]
```

Les différents modes (*flags*) Enfin il faut noter qu'on peut passer à `re.compile` un certain nombre de drapeaux (*flags*) qui modifient globalement l'interprétation de la chaîne, et qui peuvent rendre service.

Vous trouverez une liste exhaustive de ces *flags* ici. Ils ont en général un nom long et parlant, et un alias court sur un seul caractère. Les plus utiles sont sans doute :

- `IGNORECASE` (*alias* `I`) pour ne pas faire la différence entre minuscules et majuscules ;
- `ASCII` (*alias* `A`) limite `\w` aux caractères `[a-zA-Z0-9_]` ;
- `LOCALE` (*alias* `L`) cette fois `\w` dépend de l'environnement de localisation courant (`LC_ALL`, `LC_CTYPE`) courant ; attention, ça ne fonctionne que sur les encodages sur 8 bits et c'est donc déprécié ;
- `MULTILINE` (*alias* `M`) permet de traiter `^` et `$` comme début et fin de ligne dans une chaîne contenant des fins de ligne ;
- `DOTALL` (*alias* `S`) fait correspondre `.` (le point) aussi à une fin de ligne (par défaut, c'est tout caractère sauf une fin de ligne).

Comme c'est souvent le cas, on doit passer à `re.compile` un **ou logique** (caractère `|`) des différents flags que l'on veut utiliser, c'est-à-dire qu'on fera par exemple :

```
In [ ]: regexp = "a*b+"
        re_obj = re.compile(regexp, flags=re.IGNORECASE | re.DEBUG)

In [ ]: # on ignore la casse des caractères
        print(regexp, "->", nice(re_obj.match("AabB")))
```

Comment construire une expression régulière

Nous pouvons à présent voir comment construire une expression régulière, en essayant de rester synthétique (la documentation du module `re` en donne une version exhaustive).

La brique de base : le caractère Au commencement il faut spécifier des caractères :

- **un seul** caractère :
- vous le citez tel quel, en le précédent d'un backslash `\` s'il a par ailleurs un sens spécial dans le micro-langage de regexps (comme `+`, `*`, `[`, etc.) ;
- l'**attrape-tout** (*wildcard*) :
- un point `.` signifie "n'importe quel caractère" ;
- **un ensemble** de caractères avec la notation `[...]` qui permet de décrire par exemple :
- `[a1=]` un ensemble in extenso, ici un caractère parmi `a`, `1`, ou `=` ;
- `[a-z]` un intervalle de caractères, ici de `a` à `z` ;
- `[15e-g]` un mélange des deux, ici un ensemble qui contiendrait `1`, `5`, `e`, `f` et `g` ;
- `[^15e-g]` une **négation**, qui a `^` comme premier caractère dans les `[]`, ici tout sauf l'ensemble précédent ;
- un **ensemble prédéfini** de caractères, avec entre autres les notations :
- `\w` les caractères alphanumériques, et `\W` (les autres) ;
- `\s` les caractères "blancs" - espace, tabulation, saut de ligne, etc., et `\S` (les autres) ;
- `\d` pour les chiffres, et `\D` (les autres).

```
In [ ]: sample = "abcd"

        for regexp in ['abcd', 'ab[cd][cd]', 'ab[a-z]d', r'abc.', r'abc\.']:
            match = re.match(regexp, sample)
            print(f"{sample} / {regexp:<10s} → {nice(match)}")
```

Pour ce dernier exemple, comme on a échappé le `.` avec un backslash, il faut que la chaîne en entrée contienne vraiment un `.` :

```
In [ ]: print(nice(re.match (r"abc\.", "abc.")))
```

En série ou en parallèle Si je fais une analogie avec les montages électriques, jusqu'ici on a vu le montage en série, on met des expressions régulières bout à bout qui filtrent (`match`) la chaîne en entrée séquentiellement du début à la fin. On a *un peu* de marge pour spécifier des alternatives, lorsqu'on fait par exemple :

```
"ab[cd]ef"
```

mais c'est limité à **un seul** caractère. Si on veut reconnaitre deux mots qui n'ont pas grand-chose à voir comme `abc` **ou** `def`, il faut en quelque sorte mettre deux regexps en parallèle, et c'est ce que permet l'opérateur `|` :

```
In [ ]: regexp = "abc|def"

        for sample in ['abc', 'def', 'aef']:
            match = re.match(regexp, sample)
            print(f"{sample} / {regexp} → {nice(match)}")
```

Fin(s) de chaîne Selon que vous utilisez `match` ou `search`, vous précisez si vous vous intéressez uniquement à un match en début (`match`) ou n'importe où (`search`) dans la chaîne.

Mais indépendamment de cela, il peut être intéressant de "coller" l'expression en début ou en fin de ligne, et pour ça il existe des caractères spéciaux :

- `^` lorsqu'il est utilisé comme un caractère (c'est à dire pas en début de `[]`) signifie un début de ligne ;
- `\A` est équivalent sauf en mode MULTILINE où il indique le début de la chaîne ;
- `$` correspond à une fin de ligne ;
- `\Z` est équivalent sauf en MULTILIGNE où il indique la fin de la chaîne.

Attention à entrer le `^` correctement, il vous faut le caractère ASCII et non un voisin dans la ménagerie Unicode.

```
In [ ]: sample = 'abcd'

        for regexp in [r'bc', r'\Aabc', r'^abc',
                       r'\Abc', r'^bc', r'bcd\Z',
                       r'bcd$', r'bc\Z', r'bc$' ]:
            match = re.match(regexp, sample)
            search = re.search(regexp, sample)
            print(f"{sample} / {regexp:5s} match → {nice(match):6s} search → {nice(search)}
```

On a en effet bien le motif `bc` dans la chaîne en entrée, mais il n'est ni au début ni à la fin.

Parenthéser - (grouper) Pour pouvoir faire des montages élaborés, il faut pouvoir parenthéser.

```
In [ ]: # une parenthése dans une RE
        # pour mettre en ligne :
        # un début 'a',
        # un milieu 'bc' ou 'de'
        # et une fin 'f'
        regexp = "a(bc|de)f"

In [ ]: for sample in ['abcf', 'adef',  'abef', 'abf']:
            match = re.match(regexp, sample)
            print(f"{sample:>4s} → {nice(match)}")
```

Les parenthèses jouent un rôle additionel de **groupe**, ce qui signifie qu'on **peut retrouver** le texte correspondant à l'expression régulière comprise dans les `()`. Par exemple, pour la première correspondance :

```
In [ ]: sample = 'abcf'
        match = re.match(regexp, sample)
        print(f"{sample}, {regexp} → {match.groups()}")
```

Dans cet exemple, on n'a utilisé qu'un seul groupe `()`, et le morceau de chaîne qui correspond à ce groupe se trouve donc être le seul groupe retourné par `MatchObject.group`.

Compter les répétitions Vous disposez des opérateurs suivants :

- `*` l'étoile qui signifie zéro ou plus - par exemple, `(ab)*` correspond pour `''` ou `'ab'` ou `'abab'` ou etc. ;
- `+` le plus qui signifie au moins une occurrence - e.g. `(ab)+` pour `ab` ou `abab` ou `ababab` ou etc ;
- `?` qui indique une option, c'est-à-dire 0 ou 1 occurence - autrement dit `(ab)?` matche `''` ou `ab` ;
- `{n}` pour exactement n occurrences de `(ab)` - e.g. `(ab){3}` qui serait exactement équivalent à `ababab` ;
- `{m,n}` entre m et n fois inclusivement.

```
In [ ]: samples = [n*'ab' for n in [0, 1, 3, 4]] + ['baba']

        for regexp in ['(ab)*', '(ab)+', '(ab){3}', '(ab){3,4}']:
            # on ajoute \A \Z pour matcher toute la chaîne
            line_regexp = r"\A{}\Z".format(regexp)
            for sample in samples:
                match = re.match(line_regexp, sample)
                print(f"{sample:>8s} / {line_regexp:14s} → {nice(match)}")
```

Groupes et contraintes Nous avons déjà vu un exemple de groupe nommé (voir `needle` plus haut), les opérateurs que l'on peut citer dans cette catégorie sont :

- `(...)` les parenthèses définissent un groupe anonyme ;
- `(?P<name>...)` définit un groupe nommé ;
- `(?:...)` permet de mettre des parenthèses mais sans créer un groupe, pour optimiser l'exécution puisqu'on n'a pas besoin de conserver les liens vers la chaîne d'entrée ;

— `(?P=name)` qui ne correspond que si l'on retrouve à cet endroit de l'entrée la même sous-chaîne que celle trouvée pour le groupe `name` en amont ;
— enfin `(?=...)`, `(?!...)` et `(?<=...)` permettent des contraintes encore plus élaborées, nous vous laissons le soin d'expérimenter avec elles si vous êtes intéressés ; sachez toutefois que l'utilisation de telles constructions peut en théorie rendre l'interprétation de votre expression régulière beaucoup moins efficace.

Greedy *vs* non-greedy Lorsqu'on stipule une répétition un nombre indéfini de fois, il se peut qu'il existe **plusieurs** façons de filtrer l'entrée avec l'expression régulière. Que ce soit avec `*`, ou `+`, ou `?`, l'algorithme va toujours essayer de trouver la **séquence la plus longue**, c'est pourquoi on qualifie l'approche de *greedy* - quelque chose comme glouton en français.

```
In [ ]: # un fragment d'HTML
        line='<h1>Title</h1>'

        # si on cherche un texte quelconque entre crochets
        # c'est-à-dire l'expression régulière "<.*>"
        re_greedy = '<.*>'

        # on obtient ceci
        # on rappelle que group(0) montre la partie du fragment
        # HTML qui matche l'expression régulière
        match = re.match(re_greedy, line)
        match.group(0)
```

Ça n'est pas forcément ce qu'on voulait faire, aussi on peut spécifier l'approche inverse, c'est-à-dire de trouver la **plus-petite** chaîne qui correspond, dans une approche dite *non-greedy*, avec les opérateurs suivants :

— `*?` : `*` mais *non-greedy* ;
— `+?` : `+` mais *non-greedy* ;
— `??` : `?` mais *non-greedy*.

```
In [ ]: # ici on va remplacer * par *? pour rendre l'opérateur * non-greedy
        re_non_greedy = re_greedy = '<.*?>'

        # mais on continue à cherche un texte entre <> naturellement
        # si bien que cette fois, on obtient
        match = re.match(re_non_greedy, line)
        match.group(0)
```

S'agissant du traitement des fins de ligne Il peut être utile, pour conclure cette présentation, de préciser un peu le comportement de la bibliothèque vis-à-vis des fins de ligne.

Historiquement, les expressions régulières telles qu'on les trouve dans les bibliothèques C, donc dans `sed`, `grep` et autre utilitaires Unix, sont associées au modèle mental où on filtre les entrées ligne par ligne.

Le module `re` en garde des traces, puisque :

```
In [ ]: # un exemple de traitement des 'newline'
        sample = """une entrée
        sur
        plusieurs
```

```
        lignes
        """
```

```
In [ ]: match = re.compile("(.*)").match(sample)
        match.groups()
```

Vous voyez donc que l'attrape-tout `'.'` en fait n'attrape pas le caractère de fin de ligne `\n`, puisque si c'était le cas et compte tenu du coté *greedy* de l'algorithme on devrait voir ici tout le contenu de `sample`. Il existe un *flag* `re.DOTALL` qui permet de faire de `.` un vrai attrape-tout qui capture aussi les *newline* :

```
In [ ]: match = re.compile("(.*)", flags=re.DOTALL).match(sample)
        match.groups()
```

Cela dit, le caractère *newline* est par ailleurs considéré comme un caractère comme un autre, on peut le mentionner **dans une regexp** comme les autres. Voici quelques exemples pour illustrer tout ceci :

```
In [ ]: # on peut limiter la correspondance de \w aux caractères US-ASCII
        # en utilisant le drapeau re.ASCII
        match = re.compile("([\w ]*)", re.ASCII).match(sample)
        match.groups()
```

```
In [ ]: # par défaut en Python 3 \w couvre les caractères Unicode
        # considérés comme alphanumérique, ici le 'é'
        match = re.compile("([\w ]*)").match(sample)
        match.groups()
```

```
In [ ]: # si on ajoute \n à la liste des caractères attendus
        # on obtient bien tout le contenu initial

        # attention ici il ne FAUT PAS utiliser un raw string,
        # car on veut vraiment écrire un newline dans la regexp

        match = re.compile("([\w \n]*)").match(sample)
        match.groups()
```

Conclusion

La mise au point d'expressions régulières est certes un peu exigeante, et demande pas mal de pratique, mais permet d'écrire en quelques lignes des fonctionnalités très puissantes, c'est un investissement très rentable :)

Je vous signale enfin l'existence de **sites web** qui évaluent une expression régulière **de manière interactive** et qui peuvent rendre la mise au point moins fastidieuse.

Je vous signale notamment https://pythex.org/, et il en existe beaucoup d'autres.

Pour en savoir plus

Pour ceux qui ont quelques rudiments de la théorie des langages, vous savez qu'on distingue en général : * l'**analyse lexicale**, qui découpe le texte en morceaux (qu'on appelle des *tokens*) ; * et l'**analyse syntaxique** qui décrit pour simplifier à l'extrême l'ordre dans lequel on peut trouver les tokens.

Avec les expression régulières, on adresse le niveau de l'analyse lexicale. Pour l'analyse syntaxique, qui est franchement au delà des objectifs de ce cours, il existe de nombreuses alternatives, parmi lesquelles : * `pyparsing` * `PLY (Python Lex-Yacc)` * `ANTLR` qui est un outil écrit en Java mais qui peut générer des parsers en Python * ...

2.5 Expressions régulières

Nous vous proposons dans ce notebook quelques exercices sur les expressions régulières. Faisons quelques remarques avant de commencer :

— nous nous concentrons sur l'écriture de l'expression régulière en elle-même, et pas sur l'utilisation de la bibliothèque ;
— en particulier, tous les exercices font appel à `re.match` entre votre *regexp* et une liste de chaînes d'entrée qui servent de jeux de test.

Liens utiles Pour travailler sur ces exercices, vous pouvez profitablement avoir sous la main :

— la documentation officielle ;
— et cet outil interactif sur https://pythex.org/ qui permet d'avoir un retour presque immédiat, et donc d'accélérer la mise au point.

2.5.1 Exercice - niveau basique

Identificateurs Python

```
In [1]: # évaluez cette cellule pour charger l'exercice
        from corrections.regexp_pythonid import exo_pythonid
```

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

        <ipython-input-1-ce7e6fb238d0> in <module>()
          1 # évaluez cette cellule pour charger l'exercice
    ----> 2 from corrections.regexp_pythonid import exo_pythonid


        ModuleNotFoundError: No module named 'corrections'
```

On vous demande d'écrire une expression régulière qui décrit les noms de variable en Python. Pour cet exercice on se concentre sur les caractères ASCII. On exclut donc les noms de variables qui pourraient contenir des caractères exotiques comme les caractères accentués ou autres lettres grecques.

Il s'agit donc de reconnaître toutes les chaînes qui commencent par une lettre ou un _, suivi de lettres, chiffres ou _.

```
In [ ]: # quelques exemples de résultat attendus
        exo_pythonid.example()
```

```
In [ ]: # à vous de jouer: écrivez ici
        # sous forme de chaîne votre expression régulière

        regexp_pythonid = r"<votre_regexp>"
```

```
In [ ]: # évaluez cette cellule pour valider votre code
        exo_pythonid.correction(regexp_pythonid)
```

2.5.2 Exercice - niveau intermédiaire (1)

Lignes avec nom et prénom

```
In [ ]: # pour charger l'exercice
        from corrections.regexp_agenda import exo_agenda
```

On veut reconnaître dans un fichier toutes les lignes qui contiennent un nom et un prénom.

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

Plus précisément, on cherche les chaînes qui :

— commencent par une suite - possiblement vide - de caractères alphanumériques (vous pouvez utiliser `\w`) ou tiret haut (-) qui constitue le prénom ;
— contiennent ensuite comme séparateur le caractère 'deux-points' : ;
— contiennent ensuite une suite - cette fois jamais vide - de caractères alphanumériques, qui consitue le nom ;
— et enfin contiennent un deuxième caractère : mais optionnellement seulement.

On vous demande de construire une expression régulière qui définit les deux groupes `nom` et `prenom`, et qui rejette les lignes qui ne satisfont pas ces critères.

```
In [ ]: # entrez votre regexp ici
        # il faudra la faire terminer par \Z
        # regardez ce qui se passe si vous ne le faites pas

        regexp_agenda = r"<votre regexp>\Z"
```

```
In [ ]: # évaluez cette cellule pour valider votre code
        exo_agenda.correction(regexp_agenda)
```

2.5.3 Exercice - niveau intermédiaire (2)

Numéros de téléphone

```
In [ ]: # pour charger l'exercice
        from corrections.regexp_phone import exo_phone
```

Cette fois on veut reconnaître des numéros de téléphone français, qui peuvent être :

— soit au format contenant 10 chiffres dont le premier est un `0` ;
— soit un format international commençant par `+33` suivie de 9 chiffres.

Dans tous les cas on veut trouver dans le groupe 'number' les 9 chiffres vraiment significatifs, comme ceci :

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

```
In [ ]: # votre regexp
        # à nouveau il faut terminer la regexp par \Z
        regexp_phone = r"<votre regexp>\Z"
```

```
In [ ]: # évaluez cette cellule pour valider votre code
        exo_phone.correction(regexp_phone)
```

2.5.4 Exercice - niveau avancé

Vu comment sont conçus les exercices, vous ne pouvez pas passer à `re.compile` un drapeau comme `re.IGNORECASE` ou autre ; sachez cependant que vous pouvez ***embarquer* ces drapeaux dans la *regexp*** elle-même ; par exemple pour rendre la regexp insensible à la casse de caractères, au lieu d'appeler `re.compile` avec le flag `re.I`, vous pouvez utiliser `(?i)` comme ceci :

```
In [ ]: import re
```

```
In [ ]: # on peut embarquer les flags comme IGNORECASE
        # directement dans la regexp
        # c'est équivalent de faire ceci

        re_obj = re.compile("abc", flags=re.IGNORECASE)
        re_obj.match("ABC").group(0)
```

```
In [ ]: # ou cela

        re.match("(?i)abc","ABC").group(0)
```

```
In [ ]: # les flags comme (?i) doivent apparaître
        # en premier dans la regexp
        re.match("abc(?i)","ABC").group(0)
```

Pour plus de précisions sur ce trait, que nous avons laissé de coté dans le complément pour ne pas trop l'alourdir, voyez la documentation sur les expressions régulières et cherchez la première occurrence de `iLmsux`.

Décortiquer une URL

On vous demande d'écrire une expression régulière qui permette d'analyser des URLs.
Voici les conventions que nous avons adoptées pour l'exercice :

— la chaîne contient les parties suivantes :
— `<protocol>://<location>/<path>` ;
— l'url commence par le nom d'un protocole qui doit être parmi `http`, `https`, `ftp`, `ssh` ;
— le nom du protocole peut contenir de manière indifférente des minuscules ou des majuscules ;
— ensuite doit venir la séquence `://` ;
— ensuite on va trouver une chaîne `<location>` qui contient :
— potentiellement un nom d'utilisateur, et s'il est présent, potentiellement un mot de passe ;
— obligatoirement un nom de `hostname` ;
— potentiellement un numéro de port ;
— lorsque les 4 parties sont présentes dans `<location>`, cela se présente comme ceci :
— `<location> = <user>:<password>@<hostname>:<port>` ;
— si l'on note entre crochets les parties optionnelles, cela donne :
— `<location> = [<user>[:<password>]@]<hostname>[:<port>]` ;
— le champ `<user>` ne peut contenir que des caractères alphanumériques ; si le `@` est présent le champ `<user>` ne peut pas être vide ;
— le champ `<password>` peut contenir tout sauf un `:` et de même, si le `:` est présent le champ `<password>` ne peut pas être vide ;
— le champ `<hostname>` peut contenir un suite non-vide de caractères alphanumériques, underscores, ou `.` ;
— le champ `<port>` ne contient que des chiffres, et il est non vide si le `:` est spécifié ;

— le champ `<path>` peut être vide.

Enfin, vous devez définir les groupes `proto`, `user`, `password`, `hostname`, `port` et `path` qui sont utilisés pour vérifier votre résultat. Dans la case `Résultat attendu`, vous trouverez soit `None` si la regexp ne filtre pas l'intégralité de l'entrée, ou bien une liste ordonnée de tuples qui donnent la valeur de ces groupes ; vous n'avez rien à faire pour construire ces tuples, c'est l'exercice qui s'en occupe.

```
In [ ]: # pour charger l'exercice
        from corrections.regexp_url import exo_url

In [ ]: # exemples du résultat attendu
        exo_url.example()

In [ ]: # n'hésitez pas à construire votre regexp petit à petit

        regexp_url = "<votre_regexp>"

In [ ]: exo_url.correction(regexp_url)
```

2.6 Les slices en Python

2.6.1 Complément - niveau basique

Ce support de cours reprend les notions de *slicing* vues dans la vidéo.

Nous allons illustrer les slices sur la chaîne suivante, rappelez-vous toutefois que ce mécanisme fonctionne avec toutes les séquences que l'on verra plus tard, comme les listes ou les tuples.

```
In [ ]: chaine = "abcdefghijklmnopqrstuvwxyz"
        print(chaine)
```

Slice sans pas

On a vu en cours qu'une slice permet de désigner toute une plage d'éléments d'une séquence. Ainsi on peut écrire :

```
In [ ]: chaine[2:6]
```

Conventions de début et fin

Les débutants ont parfois du mal avec les bornes. Il faut se souvenir que :

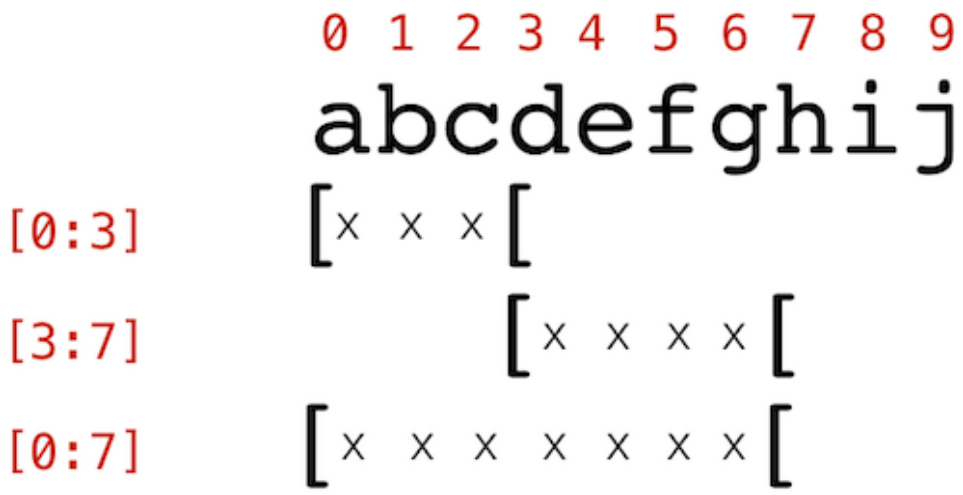
— les indices **commencent** comme toujours **à zéro** ;
— le premier indice `debut` est **inclus** ;
— le second indice `fin` est **exclu** ;
— on obtient en tout `fin-debut` items dans le résultat.

Ainsi, ci-dessus, le résultat contient `6 - 2 = 4` éléments.

Pour vous aider à vous souvenir des conventions de début et de fin, souvenez-vous qu'on veut pouvoir facilement juxtaposer deux slices qui ont une borne commune.

C'est-à-dire qu'avec :

```
In [ ]: # chaine[a:b] + chaine[b:c] == chaine[a:c]
        chaine[0:3] + chaine[3:7] == chaine[0:7]
```



début et fin

Bornes omises On peut omettre une borne :

```
In [ ]: # si on omet la première borne, cela signifie que
        # la slice commence au début de l'objet
        chaine[:6]

In [ ]: # et bien entendu c'est la même chose si on omet la deuxième borne
        chaine[24:]

In [ ]: # ou même omettre les deux bornes, auquel cas on
        # fait une copie de l'objet - on y reviendra plus tard
        chaine[:]
```

Indices négatifs On peut utiliser des indices négatifs pour compter à partir de la fin :

```
In [ ]: chaine[3:-3]

In [ ]: chaine[-3:]
```

Slice avec pas

Il est également possible de préciser un *pas*, de façon à ne choisir par exemple, dans la plage donnée, qu'un élément sur deux :

```
In [ ]: # le pas est précisé après un deuxième deux-points (:)
        # ici on va choisir un caractère sur deux dans la plage [3:-3]
        chaine[3:-3:2]
```

Comme on le devine, le troisième élément de la slice, ici `2`, détermine le pas. On ne retient donc, dans la chaîne `defghi...` que `d`, puis `f`, et ainsi de suite.

On peut préciser du coup la borne de fin (ici `-3`) avec un peu de liberté, puisqu'ici on obtiendrait un résultat identique avec `-4`.

```
In [ ]: chaine[3:-4:2]
```

Pas négatif

Il est même possible de spécifier un pas négatif. Dans ce cas, de manière un peu contre-intuitive, il faut préciser un début (le premier indice de la slice) qui soit *plus à droite* que la fin (le second indice).

Pour prendre un exemple, comme l'élément d'indice `-3`, c'est-à-dire `x`, est plus à droite que l'élément d'indice `3`, c'est-à-dire `d`, évidemment si on ne précisait pas le pas (qui revient à choisir un pas égal à `1`), on obtiendrait une liste vide :

```
In [ ]: chaine[-3:3]
```

Si maintenant on précise un pas négatif, on obtient cette fois :

```
In [ ]: chaine[-3:3:-2]
```

Conclusion

À nouveau, souvenez-vous que tous ces mécanismes fonctionnent avec de nombreux autres types que les chaînes de caractères. En voici deux exemples qui anticipent tous les deux sur la suite, mais qui devraient illustrer les vastes possiblités qui sont offertes avec les slices.

Listes Par exemple sur les listes :

```
In [ ]: liste = [0, 2, 4, 8, 16, 32, 64, 128]
        liste
```

```
In [ ]: liste[-1:1:-2]
```

Et même ceci, qui peut être déroutant. Nous reviendrons dessus.

```
In [ ]: liste[2:4] = [100, 200, 300, 400, 500]
        liste
```

2.6.2 Complément - niveau avancé

`numpy` La bibliothèque `numpy` permet de manipuler des tableaux ou des matrices. En anticipant (beaucoup) sur son usage que nous reverrons bien entendu en détails, voici un aperçu de ce que l'on peut faire avec des slices sur des objets `numpy` :

```
In [ ]: # ces deux premières cellules sont à admettre
        # on construit un tableau ligne
        import numpy as np

        un_cinq = np.array([1, 2, 3, 4, 5])
        un_cinq
```

```
In [ ]: # ces deux premières cellules sont à admettre
        # on le combine avec lui-même - et en utilisant une slice un peu magique
        # pour former un tableau carré 5x5

        array = 10 * un_cinq[:, np.newaxis] + un_cinq
        array
```

Sur ce tableau de taille 5x5, nous pouvons aussi faire du slicing et extraire le sous-tableau 3x3 au centre :

```
In [ ]: centre = array[1:4, 1:4]
        centre
```

On peut bien sûr également utiliser un pas :

```
In [ ]: coins = array[::4, ::4]
        coins
```

Ou bien retourner complètement dans une direction :

```
In [ ]: tete_en_bas = array[::-1,:]
        tete_en_bas
```

2.7 Méthodes spécifiques aux listes

2.7.1 Complément - niveau basique

Voici quelques unes des méthodes disponibles sur le type `list`.

Trouver l'information

Pour commencer, rappelons comment retrouver la liste des méthodes définies sur le type `list` :

```
In [ ]: help(list)
```

Ignorez les méthodes dont le nom commence et termine par `__` (nous parlerons de ceci en semaine 6), vous trouvez alors les méthodes utiles listées entre `append` et `sort`.

Certaines de ces méthodes ont été vues dans la vidéo sur les séquences, c'est le cas notamment de `count` et `index`.

Nous allons à présent décrire les autres, partiellement et brièvement. Un autre complément décrit la méthode `sort`. Reportez-vous au lien donné en fin de notebook pour obtenir une information plus complète.

Donnons-nous pour commencer une liste témoin :

```
In [ ]: liste = [0, 1, 2, 3]
        print('liste', liste)
```

Avertissements :

- soyez bien attentifs au nombre de fois où vous exécutez les cellules de ce notebook ;
- par exemple une liste renversée deux fois peut donner l'impression que `reverse` ne marche pas ;
- n'hésitez pas à utiliser le menu *Cell -> Run All* pour réexécuter en une seule fois le notebook entier.

`append`

La méthode `append` permet d'ajouter **un élément** à la fin d'une liste :

```
In [ ]: liste.append('ap')
        print('liste', liste)
```

`extend`

La méthode `extend` réalise la même opération, mais avec **tous les éléments** de la liste qu'on lui passe en argument :

```
In [ ]: liste2 = ['ex1', 'ex2']
        liste.extend(liste2)
        print('liste', liste)
```

`append` *vs* `+`

Ces deux méthodes `append` et `extend` sont donc assez voisines ; avant de voir d'autres méthodes de `list`, prenons un peu le temps de comparer leur comportement avec l'addition `+` de liste. L'élément clé ici, on l'a déjà vu dans la vidéo, est que la liste est un objet **mutable**. `append` et `extend` **modifient** la liste sur laquelle elles travaillent, alors que l'addition **crée un nouvel objet**.

```
In [ ]: # pour créer une liste avec les n premiers entiers, on utilise
        # la fonction built-in range(), que l'on convertit en liste
        # on aura l'occasion d'y revenir
        a1 = list(range(3))
        print(a1)
```

```
In [ ]: a2 = list(range(10, 13))
        print(a2)
```

```
In [ ]: # le fait d'utiliser + crée une nouvelle liste
        a3 = a1 + a2
```

```
In [ ]: # si bien que maintenant on a trois objets différents
        print('a1', a1)
        print('a2', a2)
        print('a3', a3)
```

Comme on le voit, après une addition, les deux termes de l'addition sont inchangés. Pour bien comprendre, voyons exactement le même scénario sous pythontutor :

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

Note : une fois que vous avez évalué la cellule avec `%%ipythontutor`, vous devez cliquer sur le bouton `Forward` pour voir pas à pas le comportement du programme.

```
In [ ]: %%ipythontutor height=230 ratio=0.7
        a1 = list(range(3))
        a2 = list(range(10, 13))
        a3 = a1 + a2
```

Alors que si on avait utilisé `extend`, on aurait obtenu ceci :

```
In [ ]: %%ipythontutor height=200 ratio=0.75
        e1 = list(range(3))
        e2 = list(range(10, 13))
        e3 = e1.extend(e2)
```

Ici on tire profit du fait que la liste est un objet mutable : `extend` **modifie** l'objet sur lequel on l'appelle (ici `e1`). Dans ce scénario on ne créée en tout que deux objets, et du coup il est inutile pour extend de renvoyer quoi que ce soit, et c'est pourquoi `e3` ici vaut None.

C'est pour cette raison que :

— l'addition est disponible sur tous les types séquences - on peut toujours réaliser l'addition puisqu'on crée un nouvel objet pour stocker le résultat de l'addition ;
— mais `append` et `extend` ne sont par exemple **pas disponibles** sur les chaînes de caractères, qui sont **immuables** - si `e1` était une chaîne, on ne pourrait pas la modifier pour lui ajouter des éléments.

`insert`

Reprenons notre inventaire des méthodes de `list`, et pour cela rappelons nous le contenu de la variable `liste` :

```
In [ ]: liste
```

La méthode `insert` permet, comme le nom le suggère, d'insérer un élément à une certaine position ; comme toujours les indices commencent à zéro et donc :

```
In [ ]: # insérer à l'index 2
        liste.insert(2, '1 bis')
        print('liste', liste)
```

On peut remarquer qu'un résultat analogue peut être obtenu avec une affectation de slice ; par exemple pour insérer au rang 5 (i.e. avant `ap`), on pourrait aussi bien faire :

```
In [ ]: liste[5:5] = ['3 bis']
        print('liste', liste)
```

`remove`

La méthode `remove` détruit la **première occurence** d'un objet dans la liste :

```
In [ ]: liste.remove(3)
        print('liste', liste)
```

`pop`

La méthode `pop` prend en argument un indice ; elle permet d'extraire l'élément à cet indice. En un seul appel on obtient la valeur de l'élément et on l'enlève de la liste :

```
In [ ]: popped = liste.pop(0)
        print('popped', popped, 'liste', liste)
```

Si l'indice n'est pas précisé, c'est le dernier élément de la liste qui est visé :

```
In [ ]: popped = liste.pop()
        print('popped', popped, 'liste', liste)
```

`reverse`

Enfin `reverse` renverse la liste, le premier élément devient le dernier :

```
In [ ]: liste.reverse()
        print('liste', liste)
```

On peut remarquer ici que le résultat se rapproche de ce qu'on peut obtenir avec une opération de slicing comme ceci :

```
In [ ]: liste2 = liste[::-1]
        print('liste2', liste2)
```

À la différence toutefois qu'avec le slicing c'est une copie de la liste initiale qui est retournée, la liste de départ quant à elle n'est pas modifiée.

Pour en savoir plus

https ://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Note spécifique aux notebooks

`help` avec `?` Je vous signale en passant que dans un notebook vous pouvez obtenir de l'aide avec un point d'interrogation `?` inséré avant ou après un symbole. Par exemple pour obtenir des précisions sur la méthode `list.pop`, on peut faire soit :

```
In [ ]: # fonctionne dans tous les environnements Python
        help(list.pop)
```

```
In [ ]: # spécifique aux notebooks
        # l'affichage obtenu est légèrement différent
        # tapez la touche 'Esc' - ou cliquez la petite croix
        # pour faire disparaitre le dialogue qui apparaît en bas
        list.pop?
```

Complétion avec `Tab` Dans un notebook vous avez aussi la complétion ; si vous tapez - dans une cellule de code - le début d'un symbole connu dans l'environnement :

```
In [ ]: # placez votre curseur à la fin de la ligne après 'li'
        # et appuyez sur la touche 'Tab'
        li
```

Vous voyez apparaître un dialogue avec les noms connus qui commencent par `li` ; utilisez les flèches pour choisir, et 'Return' pour sélectionner.

2.8 Objets mutables et objets immuables

2.8.1 Complément - niveau basique

Les chaînes sont des objets immuables

Voici un exemple d'un fragment de code qui illustre le caractère immuable des chaînes de caractères. Nous l'exécutons sous pythontutor, afin de bien illustrer les relations entre variables et objets.

```
In [ ]: # il vous faut charger cette cellule
        # pour pouvoir utiliser les suivantes
        %load_ext ipythontutor
```

Note : une fois que vous avez évalué la cellule avec `%%ipythontutor`, vous devez cliquer sur le bouton `Forward` pour voir pas à pas le comportement du programme.

Le scénario est très simple, on crée deux variables `s1` et `s2` vers le même objet `'abc'`, puis on fait une opération `+=` sur la variable `s1`.

Comme l'objet est une chaîne, il est donc immuable, on ne **peut pas modifier l'objet** directement ; pour obtenir l'effet recherché (à savoir que `s1` s'allonge de `'def'`), Python **crée un deuxième objet**, comme on le voit bien sous pythontutor :

```
In [ ]: %%ipythontutor heapPrimitives=true
        # deux variables vers le même objet
        s1 = 'abc'
        s2 = s1
        # on essaie de modifier l'objet
        s1 += 'def'
        # pensez à cliquer sur `Forward`
```

Les listes sont des objets mutables

Voici ce qu'on obtient par contraste pour le même scénario mais qui cette fois utilise des listes, qui sont des objets mutables :

```
In [ ]: %%ipythontutor heapPrimitives=true ratio=0.8
        # deux variables vers le même objet
        liste1 = ['a', 'b', 'c']
        liste2 = liste1
        # on modifie l'objet
        liste1 += ['d', 'e', 'f']
        # pensez à cliquer sur `Forward`
```

Conclusion

Ce comportement n'est pas propre à l'usage de l'opérateur `+=` - que pour cette raison d'ailleurs nous avons tendance à déconseiller.

Les objets mutables et immuables ont par essence un comportement différent, il est très important d'avoir ceci présent à l'esprit.

Nous aurons notamment l'occasion d'approfondir cela dans la séquence consacrée aux références partagées, en semaine 3.

2.9 Tris de listes

2.9.1 Complément - niveau basique

Python fournit une méthode standard pour trier une liste, qui s'appelle, sans grande surprise, `sort`.

La méthode `sort`

Voyons comment se comporte `sort` sur un exemple simple :

```
In [ ]: liste = [8, 7, 4, 3, 2, 9, 1, 5, 6]
        print('avant tri', liste)
        liste.sort()
        print('apres tri', liste)
```

On retrouve ici, avec l'instruction `liste.sort()` un cas d'appel de méthode (ici `sort`) sur un objet (ici `liste`), comme on l'avait vu dans la vidéo sur la notion d'objet.

La première chose à remarquer est que la liste d'entrée a été modifiée, on dit "en place", ou encore "par effet de bord". Voyons cela sous pythontutor :

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

```
In [ ]: %%ipythontutor height=200 ratio=0.8
        liste = [3, 2, 9, 1]
        liste.sort()
```

On aurait pu imaginer que la liste d'entrée soit restée inchangée, et que la méthode de tri renvoie une copie triée de la liste, ce n'est pas le choix qui a été fait, cela permet d'économiser des allocations mémoire autant que possible et d'accélérer sensiblement le tri.

La fonction `sorted`

Si vous avez besoin de faire le tri sur une copie de votre liste, la fonction `sorted` vous permet de le faire :

```
In [ ]: %%ipythontutor height=200 ratio=0.8
        liste1 = [3, 2, 9, 1]
        liste2 = sorted(liste1)
```

Tri décroissant

Revenons à la méthode `sort` et aux tris *en place*. Par défaut la liste est triée par ordre croissant, si au contraire vous voulez l'ordre décroissant, faites comme ceci :

```
In [ ]: liste = [8, 7, 4, 3, 2, 9, 1, 5, 6]
        print('avant tri', liste)
        liste.sort(reverse=True)
        print('apres tri décroissant', liste)
```

Nous n'avons pas encore vu à quoi correspond cette formule `reverse=True` dans l'appel à la méthode - ceci sera approfondi dans le chapitre sur les appels de fonction - mais dans l'immédiat vous pouvez utiliser cette technique telle quelle.

Chaînes de caractères

Cette technique fonctionne très bien sur tous les types numériques (enfin, à l'exception des complexes ; en guise d'exercice, pourquoi ?), ainsi que sur les chaînes de caractères :

```
In [ ]: liste = ['spam', 'egg', 'bacon', 'beef']
        liste.sort()
        print('après tri', liste)
```

Comme on s'y attend, il s'agit cette fois d'un **tri lexicographique**, dérivé de l'ordre sur les caractères. Autrement dit, c'est l'ordre du dictionnaire. Il faut souligner toutefois, pour les personnes n'ayant jamais été exposées à l'informatique, que cet ordre, quoique déterministe, est arbitraire en dehors des lettres de l'alphabet.

Ainsi par exemple :

```
In [ ]: # deux caractères minuscules se comparent
        # comme on s'y attend
        'a' < 'z'
```

Bon, mais par contre :

```
In [ ]: # si l'un est en minuscule et l'autre en majuscule,
        # ce n'est plus le cas
        'Z' < 'a'
```

Ce qui à son tour explique ceci :

```
In [ ]: # la conséquence de 'Z' < 'a', c'est que
        liste = ['abc', 'Zoo']
        liste.sort()
        print(liste)
```

Et lorsque les chaînes contiennent des espaces ou autres ponctuations, le résultat du tri peut paraître surprenant :

```
In [ ]: # attention ici notre premiere chaîne commence par un espace
        # et le caractère 'Espace' est plus petit
        # que tous les autres caractères imprimables
        liste = [' zoo', 'ane']
        liste.sort()
        print(liste)
```

À suivre

Il est possible de définir soi-même le critère à utiliser pour trier une liste, et nous verrons cela bientôt, une fois que nous aurons introduit la notion de fonction.

2.10 Indentations en Python

2.10.1 Complément - niveau basique

Imbrications

Nous l'avons vu dans la vidéo, la pratique la plus courante est d'utiliser systématiquement une indentation de 4 espaces :

```
In [ ]: # la convention la plus généralement utilisée
        # consiste à utiliser une indentation de 4 espaces
        if 'g' in 'egg':
            print('OUI')
        else:
            print('NON')
```

Voyons tout de suite comment on pourrait écrire plusieurs tests imbriqués :

```
In [ ]: entree = 'spam'

        # pour imbriquer il suffit d'indenter de 8 espaces
        if 'a' in entree:
            if 'b' in entree:
                cas11 = True
                print('a et b')
            else:
                cas12 = True
                print('a mais pas b')
        else:
            if 'b' in entree:
                cas21 = True
                print('b mais pas a')
            else:
                cas22 = True
                print('ni a ni b')
```

Dans cette construction assez simple, remarquez bien **les deux points ' :'** à chaque début de bloc, c'est-à-dire à chaque fin de ligne `if` ou `else`.

Cette façon d'organiser le code peut paraître très étrange, notamment aux gens habitués à un autre langage de programmation, puisqu'en général les syntaxes des langages sont conçues de manière à être insensibles aux espaces et à la présentation.

Comme vous le constaterez à l'usage cependant, une fois qu'on s'y est habitué cette pratique est très agréable, une fois qu'on a écrit la dernière ligne du code, on n'a pas à réfléchir à refermer le bon nombre d'accolades ou de *end*.

Par ailleurs, comme pour tous les langages, votre éditeur favori connaît cette syntaxe et va vous aider à respecter la règle des 4 caractères. Nous ne pouvons pas publier ici une liste des commandes disponibles par éditeur, nous vous invitons le cas échéant à échanger entre vous sur le forum pour partager les recettes que vous utilisez avec votre éditeur / environnement de programmation favori.

2.10.2 Complément - niveau intermédiaire

Espaces *vs* tabulations

Version courte Il nous faut par contre donner quelques détails sur un problème que l'on rencontre fréquemment sur du code partagé entre plusieurs personnes quand celles-ci utilisent des environnement différents.

Pour faire court, ce problème est **susceptible d'apparaître dès qu'on utilise des tabulations**, plutôt que des espaces, pour implémenter les indentations. Aussi, le message à retenir ici est **de ne jamais utiliser de tabulations dans votre code Python**. Tout bon éditeur Python devrait faire cela par défaut.

Version longue En version longue, il existe un code ASCII pour un caractère qui s'appelle *Tabulation* (alias Control-i, qu'on note aussi `^I`) ; l'interprétation de ce caractère n'étant pas clairement spécifiée, il arrive qu'on se retrouve dans une situation comme la suivante.

Bernard utilise l'éditeur `vim` ; sous cet éditeur il lui est possible de mettre des tabulations dans son code, et de choisir la valeur de ces tabulations. Aussi il va dans les préférences de `vim`, choisit Tabulation=4, et écrit un programme qu'il voit comme ceci :

```
In [ ]: if 'a' in entree:
            if 'b' in entree:
                cas11 = True
                print('a et b')
            else:
                cas12 = True
                print('a mais pas b')
```

Sauf qu'en fait, il a mis un mélange de tabulations et d'espaces, et en fait le fichier contient (avec `^I` pour tabulation) :

```
if 'a' in entree:
^Iif 'b' in entree:
^I^Icas11 = True
    ^Iprint('a et b')
^Ielse:
^I^Icas12 = True
    ^Iprint('a mais pas b')
```

Remarquez le mélange de Tabulations et d'espaces dans les deux lignes avec `print`. Bernard envoie son code à Alice qui utilise `emacs`. Dans son environnement, `emacs` affiche une tabulation comme 8 caractères. Du coup Alice "voit" le code suivant :

```
if 'a' in entree:
        if 'b' in entree:
                cas11 = True
            print('a et b')
        else:
                cas12 = True
            print('a mais pas b')
```

Bref, c'est la confusion la plus totale. Aussi répétons-le, **n'utilisez jamais de tabulations dans votre code Python**.

Ce qui ne veut pas dire qu'il ne faut pas utiliser la touche `Tab` avec votre éditeur - au contraire, c'est une touche très utilisée - mais faites bien la différence entre le fait d'appuyer sur la touche `Tab` et le fait que le fichier sauvé sur disque contient effectivement un caractère tabulation. Votre éditeur favori propose très certainement une option permettant de faire les remplacements idoines pour ne pas écrire de tabulation dans vos fichiers, tout en vous permettant d'indenter votre code avec la touche `Tab`.

Signalons enfin que Python 3 est plus restrictif que Python 2 à cet égard, et interdit de mélanger des espaces et des tabulations sur une même ligne. Ce qui n'enlève rien à notre recommandation.

2.10.3 Complément - niveau avancé

Vous pouvez trouver du code qui ne respecte pas la convention des 4 caractères.

Version courte En version courte : **Utilisez toujours des indentations de 4 espaces**.

Version longue En version longue, et pour les curieux : Python **n'impose pas** que les indentations soient de 4 caractères. Aussi vous pouvez rencontrer un code qui ne respecte pas cette convention, et il nous faut, pour être tout à fait précis sur ce que Python accepte ou non, préciser ce qui est réellement requis par Python.

La règle utilisée pour analyser votre code, c'est que toutes les instructions **dans un même bloc** soient présentées avec le même niveau d'indentation. Si deux lignes successives - modulo les blocs imbriqués - ont la même indentation, elles sont dans le même bloc.

Voyons quelques exemples. Tout d'abord le code suivant est **légal**, quoique, redisons-le pour la dernière fois, **pas du tout recommandé** :

```
In [ ]: # code accepté mais pas du tout recommandé
        if 'a' in 'pas du tout recommande':
          succes = True
          print('OUI')
        else:
              print('NON')
```

En effet, les deux blocs (après `if` et après `else`) sont des blocs distincts, ils sont libres d'utiliser deux indentations différentes (ici 2 et 6).

Par contre la construction ci-dessous n'est pas légale :

```
In [ ]: # ceci n'est pas correct et est rejeté par Python
        if 'a' in entree:
            if 'b' in entree:
                cas11 = True
                print('a et b')
              else:
                cas12 = True
                print('a mais pas b')
```

En effet les deux lignes `if` et `else` font logiquement partie du même bloc, elles **doivent** donc avoir la même indentation. Avec cette présentation le lecteur Python émet une erreur et ne peut pas interpréter le code.

2.11 Bonnes pratiques de présentation de code

2.11.1 Complément - niveau basique

La PEP 8

On trouve dans la PEP 8 (en anglais) les conventions de codage qui s'appliquent à toute la bibliothèque standard, et qui sont certainement un bon point de départ pour vous aider à trouver le style de présentation qui vous convient.

Nous vous recommandons en particulier les sections sur :

— l'indentation ;
— les espaces ;
— les commentaires.

Un peu de lecture : le module `pprint`

Voici par exemple le code du module `pprint` (comme PrettyPrint) de la biblothèque standard qui permet d'imprimer des données.

La fonction du module - le pretty printing - est évidemment accessoire ici, mais vous pouvez y voir illustré :

— le *docstring* pour le module : les lignes de 11 à 35 ;
— les indentations, comme nous l'avons déjà mentionné sont à 4 espaces, et sans tabulation ;
— l'utilisation des espaces, notamment autour des affectations et opérateurs, des définitions de fonction, des appels de fonctions ;
— les lignes qui restent dans une largeur "raisonnable" (79 caractères) ;
— vous pouvez regarder notamment la façon de couper les lignes pour respecter cette limite en largeur.

```
In [ ]: from modtools import show_module_html
        import pprint
        show_module_html(pprint, lineno_width=3)
```

Espaces

Comme vous pouvez le voir dans `pprint.py`, les règles principales concernant les espaces sont les suivantes :

— S'agissant des **affectations** et **opérateurs**, on fera :

```
x = y + z
```

Et non pas :

```
x=y+z       # pas d'espaces
```

Ni :

```
x = y+z     # pas d'espaces autour du +
```

Ni encore :

```
x=y + z     # pas d'espace autour du =
```

L'idée étant d'aérer de manière homogène pour faciliter la lecture.

— On **déclare une fonction** comme ceci :

```
def foo(x, y, z):
```

Et non pas comme ceci :

```
def foo (x, y, z):     # une espace en trop avant la parenthèse ouvrante
```

Ni surtout pas comme ceci :

```
def foo(x,y,z):        # pas d'espace entre les paramètres
```

— La même règle s'applique naturellement aux **appels de fonction** :

```
foo(x, y, z)
```

Et non pas :

```
foo (x, y, z)     # une espace en trop avant la parenthèse ouvrante
```

Ni :

```
foo(x,y,z)        # pas d'espace entre les paramètres
```

Il est important de noter qu'il s'agit ici de **règles d'usage** et non pas de règles syntaxiques ; tous les exemples barrés ci-dessus sont en fait **syntaxiquement corrects**, l'interpréteur les accepterait sans souci ; mais ces règles sont **très largement adoptées**, et obligatoires pour intégrer du code dans la bibliothèque standard.

Coupures de ligne

Nous allons à présent zoomer dans ce module pour voir quelques exemples de coupure de ligne. Par contraste avec ce qui précède, il s'agit cette fois surtout de **règles syntaxiques**, qui peuvent rendre un code non valide si elles ne sont pas suivies.

Coupure de ligne sans *backslash* (\)

```
In [ ]: show_module_html(pprint,
                        beg="def pprint",
                        end="def pformat",
                        lineno_width=3)
```

La fonction `pprint` (ligne ~47) est une commodité (qui crée une instance de `PrettyPrinter`, sur lequel on envoie la méthode `pprint`).

Vous voyez ici qu'il n'est **pas nécessaire** d'insérer un *backslash* (\) à la fin des lignes 50 et 51, car il y a une parenthèse ouvrante qui n'est pas fermée à ce stade.

De manière générale, lorsqu'une parenthèse ouvrante `(` - idem avec les crochets `[` et accolades `{` - n'est pas fermée sur la même ligne, l'interpréteur suppose qu'elle sera fermée plus loin et n'impose pas de *backslash*.

Ainsi par exemple on peut écrire sans *backslash* :

```
valeurs = [
    1,
    2,
    3,
    5,
    7,
]
```

Ou encore :

```
x = un_nom_de_fonction_tres_tres_long(
        argument1, argument2,
        argument3, argument4,
    )
```

À titre de rappel, signalons aussi les chaînes de caractères à base de `"""` ou `'''` qui permettent elles aussi d'utiliser pluseurs lignes consécutives sans *backslash*, comme :

```
texte = """Les sanglots longs
Des violons
De l'automne"""
```

Coupure de ligne avec *backslash* (\) Par contre, il est des cas où le backslash est nécessaire :

```
In [ ]: show_module_html(pprint,
                        beg="components), readable, recursive",
                        end="elif len(object) ",
                        lineno_width=3)
```

Dans ce fragment au contraire, vous voyez en ligne 522 qu'**il a fallu cette fois** insérer un *backslash* \ comme caractère de continuation pour que l'instruction puisse se poursuivre en ligne 523.

Coupures de lignes - épilogue Dans tous le cas où une instruction est répartie sur plusieurs lignes, c'est naturellement l'indentation de **la première ligne** qui est significative pour savoir à quel bloc rattacher cette instruction.

Notez bien enfin qu'on peut toujours mettre un *backslash* même lorsque ce n'est pas nécessaire, mais on évite cette pratique en règle générale car les *backslash* nuisent à la lisibilité.

2.11.2 Complément - niveau intermédiaire

Outils liés à PEP 8

Il existe plusieurs outils liés à la PEP 8, pour vérifier si votre code est conforme, ou même le modifier pour qu'il le devienne.

Ce qui nous donne un excellent prétexte pour parler un peu de https ://pypi.python.org, qui est la plateforme qui distribue les logiciels disponibles via l'outil `pip3`.

Je vous signale notamment :

— https ://pypi.python.org/pypi/pep8/ pour vérifier ;
— https ://pypi.python.org/pypi/autopep8/ pour modifier automatiquement votre code et le rendre conforme.

Les deux-points ':'

Dans un autre registre entièrement, vous pouvez vous reporter à ce lien si vous êtes intéressé par la question de savoir pourquoi on a choisi un délimiteur (le caractère deux-points `:`) pour terminer les instructions comme `if`, `for` et `def`.

2.12 L'instruction `pass`

2.12.1 Complément - niveau basique

Nous avons vu qu'en Python les blocs de code sont définis par leur indentation.

Une fonction vide

Cette convention a une limitation lorsqu'on essaie de définir un bloc vide. Voyons par exemple comment on définirait en C une fonction qui ne fait rien :

```
/* une fonction C qui ne fait rien */
void foo() {}
```

Comme en Python on n'a pas d'accolade pour délimiter les blocs de code, il existe une instruction `pass`, qui ne fait rien. À l'aide de cette instruction on peut à présent définir une fonction vide comme ceci :

```
In [ ]: # une fonction Python qui ne fait rien
        def foo():
            pass
```

Une boucle vide

Pour prendre un second exemple un peu plus pratique, et pour anticiper un peu sur l'instruction `while` que nous verrons très bientôt, voici un exemple d'une boucle vide, c'est à dire sans corps, qui permet de "dépiler" dans une liste jusqu'à l'obtention d'une certaine valeur :

```
In [ ]: liste = list(range(10))
        print('avant', liste)
        while liste.pop() != 5:
            pass
        print('après', liste)
```

On voit qu'ici encore l'instruction pass a toute son utilité.

2.12.2 Complément - niveau intermédiaire

Un `if` sans `then`

```
In [ ]: # on utilise dans ces exemples une condition fausse
        condition = False
```

Imaginons qu'on parte d'un code hypothétique qui fasse ceci :

```
In [ ]: # la version initiale
        if condition:
            print("non")
        else:
            print("bingo")
```

Et que l'on veuille modifier ce code pour simplement supprimer l'impression de `non`. La syntaxe du langage **ne permet pas** de simplement commenter le premier `print` :

```
# si on commente le premier print
# la syntaxe devient incorrecte
if condition:
#    print "non"
else:
    print "bingo"
```

Évidemment ceci pourrait être récrit autrement en inversant la condition, mais parfois on s'efforce de limiter au maximum l'impact d'une modification sur le code. Dans ce genre de situation on préférera écrire plutôt :

```
In [ ]: # on peut s'en sortir en ajoutant une instruction pass
        if condition:
        #    print "non"
            pass
        else:
            print("bingo")
```

Une classe vide

Enfin, comme on vient de le voir dans la vidéo, on peut aussi utiliser `pass` pour définir une classe vide comme ceci :

```
In [ ]: class Foo:
            pass
```

```
In [ ]: foo = Foo()
```

2.13 Fonctions avec ou sans valeur de retour

2.13.1 Complément - niveau basique

Le style procédural

Une procédure est une fonction qui se contente de dérouler des instructions. Voici un exemple d'une telle fonction :

```
In [ ]: def affiche_carre(n):
            print("le carre de", n, "vaut", n*n)
```

qui s'utiliserait comme ceci :

```
In [ ]: affiche_carre(12)
```

Le style fonctionnel

Mais en fait, dans notre cas, il serait beaucoup plus commode de définir une fonction qui **retourne** le carré d'un nombre, afin de pouvoir écrire quelque chose comme :

```
surface = carre(15)
```

Quitte à imprimer cette valeur ensuite si nécessaire. Jusqu'ici nous avons fait beaucoup appel à `print`, mais dans la pratique, imprimer n'est pas un but en soi, au contraire bien souvent.

L'instruction `return`

Voici comment on pourrait écrire une fonction `carre` qui **retourne** (on dit aussi **renvoie**) le carré de son argument :

```
In [ ]: def carre(n):
            return n*n

        if carre(8) <= 100:
            print('petit appartement')
```

La sémantique (le mot savant pour "comportement") de l'instruction `return` est assez simple. La fonction qui est en cours d'exécution **s'achève** immédiatement, et l'objet cité dans l'instruction `return` est retourné à l'appelant, qui peut utiliser cette valeur comme n'importe quelle expression.

Le singleton `None`

Le terme même de fonction, si vous vous rappelez vos souvenirs de mathématiques, suggère qu'on calcule un résultat à partir de valeurs d'entrée. Dans la pratique il est assez rare qu'on définisse une fonction qui ne retourne rien.

En fait **toutes** les fonctions retournent quelque chose. Lorsque le programmeur n'a pas prévu d'instruction `return`, Python retourne un objet spécial, baptisé `None`. Voici par exemple ce qu'on obtient si on essaie d'afficher la valeur de retour de notre première fonction, qui, on le rappelle, ne retourne rien :

```
In [ ]: # ce premier appel provoque l'impression d'une ligne
        retour = affiche_carre(15)
```

```
In [ ]: # voyons ce qu'a retourné la fonction affiche_carre
        print('retour =', retour)
```

L'objet `None` est un singleton prédéfini par Python, un peu comme `True` et `False`. Ce n'est pas par contre une valeur booléenne, nous aurons l'occasion d'en reparler.

Un exemple un peu plus réaliste

Pour illustrer l'utilisation de `return` sur un exemple plus utile, voyons le code suivant :

```
In [ ]: def premier(n):
            """
            Retourne un booléen selon que n est premier ou non
            Retourne None pour les entrées négatives ou nulles
            """
            # retourne None pour les entrées non valides
            if n <= 0:
                return
            # traiter le cas singulier
            elif n == 1:
                return False
            # chercher un diviseur dans [2..n-1]
            # bien sûr on pourrait s'arrêter à la racine carrée de n
            # mais ce n'est pas notre sujet
            else:
                for i in range(2, n):
                    if n % i == 0:
                        # on a trouvé un diviseur,
                        # on peut sortir de la fonction
                        return False
            # à ce stade, le nombre est bien premier
            return True
```

Cette fonction teste si un entier est premier ou non ; il s'agit naturellement d'une version d'école, il existe d'autres méthodes beaucoup plus adaptées à cette tâche. On peut toutefois vérifier que cette version est fonctionnelle pour de petits entiers comme suit. On rappelle que `1` n'est pas considéré comme un nombre premier :

```
In [ ]: for test in [-2, 1, 2, 4, 19, 35]:
            print(f"premier({test:2d}) = {premier(test)}")
```

`return` sans valeur Pour les besoins de cette discussion, nous avons choisi de retourner `None` pour les entiers négatifs ou nuls, une manière comme une autre de signaler que la valeur en entrée n'est pas valide.

Ceci n'est pas forcément une bonne pratique, mais elle nous permet ici d'illustrer que dans le cas où on ne mentionne pas de valeur de retour, Python retourne `None`.

`return` interrompt la fonction Comme on peut s'en convaincre en instrumentant le code - ce que vous pouvez faire à titre d'exercice en ajoutant des fonctions `print` - dans le cas d'un nombre qui n'est pas premier la boucle `for` ne va pas jusqu'à son terme.

On aurait pu d'ailleurs tirer profit de cette propriété pour écrire la fonction de manière légèrement différente comme ceci :

```
In [ ]: def premier_sans_else(n):
            """
            Retourne un booléen selon que n est premier ou non
            Retourne None pour les entrées négatives ou nulles
            """
            # retourne None pour les entrées non valides
            if n <= 0:
                return
            # traiter le cas singulier
            if n == 1:
                return False
            # par rapport à la première version, on a supprimé
            # la clause else: qui est inutile
            for i in range(2, n):
                if n % i == 0:
                    # on a trouve un diviseur
                    return False
            # a ce stade c'est que le nombre est bien premier
            return True
```

C'est une question de style et de goût. En tous cas, les deux versions sont tout à fait équivalentes, comme on le voit ici :

```
In [ ]: for test in [-2, 2, 4, 19, 35]:
            print(f"pour n = {test:2d} : premier → {premier(test)}\n"
                  f"    premier_sans_else → {premier_sans_else(test)}\n")
```

Digression sur les chaînes Vous remarquerez dans cette dernière cellule, si vous regardez bien le paramètre de `print`, qu'on peut accoler deux chaines (ici deux *f-strings*) sans même les ajouter ; un petit détail pour éviter d'alourdir le code :

```
In [ ]: # quand deux chaines apparaissent immédiatement
        # l'une après l'autre sans opérateur, elles sont concaténées
        "abc" "def"
```

2.14 Formatage

2.14.1 Exercice - niveau basique

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

Vous devez écrire une fonction qui prend deux arguments :

— une chaîne de caractères qui designe le prénom d'un élève ;
— un entier qui indique la note obtenue.

Elle devra retourner une chaîne de caractères selon que la note est

— $0 \leqslant note < 10$
— $10 \leqslant note < 16$
— $16 \leqslant note \leqslant 20$

comme on le voit sur les exemples :

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

```
In [ ]: # à vous de jouer
        def label(prenom, note):
            "votre code"
```

```
In [ ]: # pour corriger
        exo_label.correction(label)
```

2.15 Séquences

2.15.1 Exercice - niveau basique

Slicing

Commençons par créer une chaîne de caractères. Ne vous inquiétez pas si vous ne comprenez pas encore le code d'initialisation utilisé ci-dessous.

Pour les plus curieux, l'instruction `import` permet de charger dans votre programme une boîte à outils que l'on appelle un module. Python vient avec de nombreux modules qui forment la bibliothèque standard. Le plus difficile avec les modules de la bibliothèque standard est de savoir qu'ils existent. En effet, il y en a un grand nombre et bien souvent il existe un module pour faire ce que vous souhaitez.

Ici en particulier nous utilisons le module `string`.

```
In [ ]: import string
        chaine = string.ascii_lowercase
        print(chaine)
```

Pour chacune des sous-chaînes ci-dessous, écrire une expression de slicing sur `chaine` qui renvoie la sous-chaîne. La cellule de code doit retourner `True`.

Par exemple, pour obtenir "def" :

```
In [ ]: chaine[3:6] == "def"
```

1) Écrivez une slice pour obtenir "vwx" (n'hésitez pas à utiliser les indices négatifs) :

```
In [ ]: chaine[ <votre_code> ] == "vwx"
```

2) Une slice pour obtenir "wxyz" (avec une seule constante) :

```
In [ ]: chaine[ <votre_code> ] == "wxyz"
```

3) Une slice pour obtenir "dfhjlnprtvxz" (avec deux constantes) :

```
In [ ]: chaine[ <votre_code> ] == "dfhjlnprtvxz"
```

4) Une slice pour obtenir "xurolifc" (avec deux constantes) :

```
In [ ]: chaine[ <votre_code> ] == "xurolifc"
```

2.15.2 Exercice - niveau intermédiaire

Longueur

```
In [ ]: # il vous faut évaluer cette cellule magique
        # pour charger l'exercice qui suit
        # et autoévaluer votre réponse
        from corrections.exo_inconnue import exo_inconnue
```

On vous donne une chaîne `composite` dont on sait qu'elle a été calculée à partir de deux chaînes `inconnue` et `connue` comme ceci :

```
composite = connue + inconnue + connue
```

On vous donne également la chaîne `connue`. Imaginez par exemple que vous avez (ce ne sont pas les vraies valeurs) :

```
connue = '0bf1'
composite = '0bf1a9730e150bf1'
```

alors, dans ce cas :

```
inconnue = 'a9730e15'
```

L'exercice consiste à écrire une fonction qui retourne la valeur de `inconnue` à partir de celles de `composite` et `connue`. Vous pouvez admettre que `connue` n'est pas vide, c'est-à-dire qu'elle contient au moins un caractère.

Vous pouvez utiliser du *slicing*, et la fonction `len()`, qui retourne la longueur d'une chaîne :

```
In [ ]: len('abcd')
```

```
In [ ]: # à vous de jouer
        def inconnue(composite, connue):
            "votre code"
```

Une fois votre code évalué, vous pouvez évaluer la cellule suivante pour vérifier votre résultat.

```
In [ ]: # correction
        exo_inconnue.correction(inconnue)
```

Lorsque vous évaluez cette cellule, la correction vous montre :

- dans la première colonne l'appel qui est fait à votre fonction ;
- dans la seconde colonne la valeur attendue pour `inconnue` ;
- dans la troisième colonne ce que votre code a réellement calculé.

Si toutes les lignes sont **en vert** c'est que vous avez réussi cet exercice.

Vous pouvez essayer autant de fois que vous voulez, mais il vous faut alors à chaque itération :

- évaluer votre cellule-réponse (là où vous définissez la fonction `inconnue`) ;
- et ensuite évaluer la cellule correction pour la mettre à jour.

2.16 Listes

2.16.1 Exercice - niveau basique

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

Vous devez écrire une fonction `laccess` qui prend en argument une liste, et qui retourne :

— `None` si la liste est vide ;
— sinon le dernier élément de la liste si elle est de taille paire ;
— et sinon l'élément du milieu.

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

```
In [ ]: # écrivez votre code ici
        def laccess(liste):
            return "votre code"
```

```
In [ ]: # pour le corriger
        exo_laccess.correction(laccess)
```

Une fois que votre code fonctionne, vous pouvez regarder si par hasard il marcherait aussi avec des chaînes :

```
In [ ]: from corrections.exo_laccess import exo_laccess_strings
```

```
In [ ]: exo_laccess_strings.correction(laccess)
```

2.17 Compréhensions

2.17.1 Exercice - niveau basique

Liste des valeurs d'une fonction

```
In [ ]: # Pour charger l'exercice
        from corrections.exo_liste_p import exo_liste_P
```

On se donne une fonction polynomiale :
$P(x) = 2x^2 - 3x - 2$
On vous demande d'écrire une fonction `liste_P` qui prend en argument une liste de nombres réels x et qui retourne la liste des valeurs $P(x)$.

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

Écrivez votre code dans la cellule suivante (*On vous suggère d'écrire une fonction `P` qui implémente le polynôme mais ça n'est pas strictement indispensable, seul le résultat de `liste_P` compte*) :

```
In [ ]: def P(x):
            "<votre code>"

        def liste_P(liste_x):
            "votre code"
```

Et vous pouvez le vérifier en évaluant cette cellule :

```
In [ ]: # pour vérifier votre code
        exo_liste_P.correction(liste_P)
```

2.17.2 Récréation

Si vous avez correctement implémenté la fonction `liste_P` telle que demandé dans le premier exercice, vous pouvez visualiser le polynôme `P` en utilisant `matplotlib` avec le code suivant :

```
In [ ]: # on importe les bibliothèque
        import numpy as np
        import matplotlib.pyplot as plt

In [ ]: # un échantillon des X entre -10 et 10
        X = np.linspace(-10, 10)

        # et les Y correspondants
        Y = liste_P(X)

In [ ]: # on n'a plus qu'à dessiner
        plt.plot(X, Y)
        plt.show()
```

2.18 Compréhensions

2.18.1 Exercice - niveau intermédiaire

Mise au carré

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

On vous demande à présent d'écrire une fonction dans le même esprit que ci-dessus. Cette fois, chaque ligne contient, séparés par des point-virgules, une liste d'entiers, et on veut obtenir une nouvelle chaîne avec les carrés de ces entiers, séparés par des deux-points.

À nouveau les lignes peuvent être remplies de manière approximative, avec des espaces, des tabulations, ou même des points-virgules en trop, que ce soit au début, à la fin, ou au milieu d'une ligne.

```
In [ ]: # exemples
        exo_carre.example()

In [ ]: # écrivez votre code ici
        def carre(ligne):
            "<votre_code>"

In [ ]: # pour corriger
        exo_carre.correction(carre)
```

SEMAINE 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 [ ]: # 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, 2 entiers :

```
In [ ]: # 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 [ ]: # 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 newline
                print(line, end='')
```

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

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

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

```
In [ ]: # Si 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 enleve les fins de ligne
                l1 = l1.strip()
                for l2 in entree:
                    # on enleve les fins de ligne
                    l2 = l2.strip()
                    print(l1, "x", l2)
```

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, aussi pour examiner ce texte nous allons utiliser la fonction `repr()`, et voici pourquoi :

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

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

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

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

```
In [ ]: # read() retourne TOUT le contenu
        # ne pas utiliser avec de très gros fichier 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="")
```

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

On voit donc que chaque bloc contient bien 4 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-sortie 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 quand nous avons interagi avec eux avec des objets de type `str` :

```
In [ ]: # 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))
```

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 : 1. créer un fichier en mode texte, et y insérer du texte en UTF-8 1. relire le fichier en mode binaire, et retrouver le codage des différents caractères.

```
In [ ]: # phase 1 : on écrit un fichier avec du texte en UTF-8
        # on ouvre le 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 [ ]:
```
        # phase 2: on rouvre le fichier en mode binaire
        with open('strbytes', 'rb') as rawinput:
            # on relit 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)}]")
```

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 [ ]:
```
        # 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")
```

Ce qui correpond au fait que nos 4 caractères non-ASCII (3 `é` et 1 `à`) sont tous encodés par UTF-8 comme 2 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 à * au glossaire sur la notion de `object file`, * et aussi et surtout au module `io` qui décrit plus en détails 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`

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 [ ]: from pathlib import Path
```

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

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

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

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

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

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

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

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

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

```
In [ ]: # 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
```

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

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

Détruire un fichier

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

```
In [ ]: # 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")
```

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

```
In [ ]: # 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
```

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

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

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

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

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

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

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 librairie standard, et qui permettent de lire ou écrire des données dans des fichiers.

Le problème

Les données dans un programme python sont stockés 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 qu'on trouve dans les langages modernes comme python, ruby ou JavaScript.

La librairie standard python contient le module json que nous illustrons très rapidement ici :

```
In [ ]: 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 de : * `tuple`, qui se fait encoder comme une liste ; * `complex`, `set` et `frozenset`, qu'on ne peut pas encoder du tout (sans étendre la librairie).

C'est ce qui explique ce qui suit :

```
In [ ]: # 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])

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

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 librairies 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, ou MacOS) est un logiciel (*kernel*) qui a l'exclusivité pour interagir physiquement avec le matériel (CPU, mémoire, disques, 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 4 opérations élémentaires * 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 qu'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 lorsqu'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 [ ]: import sys
        for channel in (sys.stdin, sys.stdout, sys.stderr):
            print(channel)
```

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 [ ]: # 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')
```

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

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

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

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 [ ]: 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 que d'abord 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 item 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, ou 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 4 syntaxes qu'on a vues ci-dessus, on obtiendrait dans ce cas

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

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

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

```
In [ ]: simple3 == simple4
```

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

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

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

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

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

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

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

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

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

In [ ]: # 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)
```

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 le 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 occurences 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 qu'on dispose d'un tuple `couple` dont on sait qu'il a deux éléments :

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

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

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 lorsqu'on construit un tuple - et on peut tout aussi bien écrire, et c'est le cas d'usage le plus fréquent d'omission des paranthèses pour le tuple :

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

Autres types

Cette technique fonctionne aussi bien avec d'autres types. Par exemple je peux utiliser :

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

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

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

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

En réalité, les seules contraintes fixées par `python` sont que * le terme à droite du signe = est un *iterable* (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 tous 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 [ ]: a = 1
        b = 2
        a, b = b, a
        print('a', a, 'b', b)
```

Extended unpacking

Le *extended unpacking* a été introduit en python3 ; commençons par en voir un exemple :

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

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

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

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

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

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 [ ]: entree = [1, 2, 3]

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

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

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

Ou encore, sans doute un peu plus lisible :

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

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

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

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 [ ]: # 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}")
```

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

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

À chaque itération, on trouve dans `entree` un tuple (d'abord `(1, 2)`, puis à l'iteration 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 qu'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 [ ]: 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 [ ]: list(zip(villes, populations))
```

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

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

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

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 [ ]: # 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)
```

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

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

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

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

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-latin-15, qui contient le même texte que "Une charogne", mais encodé en iso-latin-15, 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 même, 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 isolatin, sans que cet encodage soit passé en paramètre à `comptage`.

C'est d'ailleurs le propos de la librairie `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 [ ]: annuaire = {'marc': 35, 'alice': 30, 'eric': 38}
        print(annuaire)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

On remarque d'abord que les entrées sont listées dans le désordre, plus précisement, 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 [ ]: for clé in annuaire.keys():
            print(clé)
```

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

La fonction `len`

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

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

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

`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 ordonnes

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

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 customisation (une sous-classe) du type `dict`, qui cette fois possède cette bonne propriété :

```
In [ ]: 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)
```

`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 [ ]: # on lit dans un fichier des couples (x, y)

        tuples = [
            (1, 2),
            (2, 1),
            (1, 3),
            (2, 4),
        ]
```

```
In [ ]: # 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)
```

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

```
In [ ]: from collections import defaultdict

        # on indique que les valeurs doivent être créés 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)
```

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

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

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

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

        sorted(compteurs.items())
```

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 [ ]: print('avant', annuaire)
        # ceci sera sans effet car eric est déjà présent
        print('set_default eric', annuaire.setdefault('eric', 50))
        # par contre ceci va insérer une entrée dans le dictionnaire
        print('set_default inconnu', annuaire.setdefault('inconnu', 50))
        # comme on le voit
        print('après', annuaire)
```

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 avec python2), mais des **objets itérables** :

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

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

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

et un test d'appartenance avec `in`

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

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

Mais ce ne sont pas des listes

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

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

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

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

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

In [ ]: # 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)
```

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

python2 Ceci est naturellement en fort contraste avec tout ce qui se passait en python2, 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 intermédaire

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 contruites à partir des types `builtin`. 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 (clé) = 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 [ ]: d = {}
```

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

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

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

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

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

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

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

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

Pour conclure, il faut retenir qu'un objet n'est éligible pour être utilisé comme clé que s'il est **composé de types immuables du 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 *builtin*. 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 age et une adresse mail.

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

```
In [ ]: personnes = [
            {'nom': 'pierre',  'age': 25, 'email': 'pierre@foo.com'},
            {'nom': 'paul',    'age': 18, 'email': 'paul@bar.com'},
            {'nom': 'jacques', 'age': 52, 'email': 'jacques@cool.com'},
        ]
```

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

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

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

Ce qui nous donne

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

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

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

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 [ ]: 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 [ ]: personnes2 = [
            Personne('pierre', 25, 'pierre@foo.com'),
            Personne('paul', 18, 'paul@bar.com'),
            Personne('jacques', 52, 'jacques@cool.com'),
        ]
```

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

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

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

```
In [ ]: # 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 [ ]: print(index2['pierre'])
```

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, par exemple une entre *s1* et *s2* de longueur 10.

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()
```

```
In [ ]: # 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 qu'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'un seul *lookup* 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 qu'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)

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

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'ensembles. On aurait donc aussi bien pu faire :

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

Créer un ensemble vide

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

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

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

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

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

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 [ ]: # pour rappel
        heteroclite
```

Test d'appartenance

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

Cardinal

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

Manipulations

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

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

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

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

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

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

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

In [ ]: # 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)
```

La capture d'exception avec `try` et `except` sert à capturer une erreur d'exécution du programme (qu'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 [ ]: # 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)
```

Opérations classiques sur les ensembles

Donnons-nous deux ensembles simples :

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

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 [ ]: A2 | A3
```

Intersection

```
In [ ]: A2 & A3
```

Différence

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

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

Différence symétrique

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

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

Comparaisons

Ici encore on se donne deux ensembles :

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

Égalité

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

Inclusion

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

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

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

Ensembles disjoints

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

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 éventuels 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 reference,
— `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 que 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 correspondants 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 qu'on peut aussi utiliser - après toutes les clauses except - une clause

— else, qui va être exécutée si aucune exception n'est attrapée, et/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 [ ]: # 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 [ ]: # sans exception
        return_with_finally(1)
```

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

else

La logique ici est assez similaire, sauf que le code du else n'est exécutée 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 rapidement, en pratique on rencontre assez peu souvent une clause else dans un try.

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

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

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

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

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

Pour en savoir plus

Voyez le tutorial sur les exceptions dans la documentation officelle.

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

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

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

Scénario 2

```
In [ ]: # 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)
```

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

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 [ ]: undef = None

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

plutôt que

```
In [ ]: if undef == None:
            print('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 [ ]: id(True)
```

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

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 [ ]: a is b
```

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

Ou encore, plus surprenant :

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

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 [ ]: infini_1 = [None]
```

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

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

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

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

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 [ ]: 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)
```

On voit à nouveau réapparaître les élipses, 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 vues 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 [ ]: import copy
        #help(copy.copy)
        #help(copy.deepcopy)
```

Un exemple

Nous allons voir le résultat des deux formes de copies 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** correpondant 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 [ ]: # 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 [ ]: print('source == shallow_copy:', source == shallow_copy)
        print('source == deep_copy:', source == deep_copy)
```

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 [ ]: # 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}")
```

```
In [ ]: # 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}")
```

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

```
In [ ]: # voir la cellule ci-dessous si ceci vous parait peu clair
        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}")
```

On retrouve ici ce qu'on avait déjà remarqué sous pythontutor, à savoir que les trois derniers objets - immutables - 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 [ ]: print("avant, source      ", source)
        print("avant, shallow_copy", shallow_copy)
        source[0].append(4)
        print("après, source      ", source)
        print("après, shallow_copy", shallow_copy)
```

Si par contre on remplace complètement un élément de premier niveau dans la source, cela ne sera pas répercuté dans la copie superficielle

```
In [ ]: print("avant, source      ", source)
        print("avant, shallow_copy", shallow_copy)
        source[0] = 'remplacement'
        print("après, source      ", source)
        print("après, shallow_copy", shallow_copy)
```

Copie et circularité

Le module `copy` est capable de copier - même en profondeur - des objets contenant des références circulaires.

```
In [ ]: l = [None]
        l[0] = l
        l
```

```
In [ ]: copy.copy(l)
```

```
In [ ]: copy.deepcopy(l)
```

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 [ ]: # la variable a n'est pas définie
        try:
            print('a=', a)
        except NameError as e:
            print("a n'est pas définie")
```

```
In [ ]: # 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")
```

```
In [ ]: # 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")
```

Sur une liste

On peut enlever d'une liste les éléments qui correspondent à une *slice* :

```
In [ ]: # on se donne une liste
        l = list(range(12))
        print(l)
```

```
In [ ]: # 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)
```

Sur un dictionnaire

Avec `del` on peut enlever une clé, et donc la valeur correspondante, d'un dictionnaire :

```
In [ ]: # partons d'un dictionaire simple
        d = dict(foo='bar', spam='eggs', a='b')
        print(d)
```

```
In [ ]: # on peut enlever une clé avec del
        del d['a']
        print(d)
```

On peut passer plusieurs arguments à `del`

```
In [ ]: # Voyons où en sont nos données
        print('l', l)
        print('d', d)

In [ ]: # on peut invoquer 'del' avec plusieurs expressions
        # séparées par une virgule

        del l[3:], d['spam']

        print('l', l)
        print('d', d)
```

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

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 [ ]: # 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 [ ]: a.append(1)
        print('a', a, 'b', b)
```

Ceci est à mettre en contraste avec plusieurs affectations séparées :

```
In [ ]: # si on utilise deux affectations différentes
        a = []
        b = []

        # alors on peut changer a sans changer b
        a.append(1)
        print('a', a, 'b', 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 aux instructions += 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 [ ]: # Premier exemple avec un entier

        # on commence avec une référence partagée
        a = b = 3
        a is b
```

```
In [ ]: # 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)
```

```
In [ ]: # Deuxième exemple, cette fois avec une liste

        # la même référence partagée
        a = b = []
        a is b
```

```
In [ ]: # 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)
```

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 [ ]: # 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))
```

```
In [ ]: # 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))
```

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)
```

SEMAINE 4

FONCTIONS ET PORTÉE DES VARIABLES

4.1 Passage d'arguments par référence

4.1.1 Complément - niveau intermédiaire

Entre le code qui appelle une fonction et le code de la fonction elle-même

```
In [ ]: def ma_fonction(dans_fonction):
            print(dans_fonction)

        dans_appelant = ["texte"]
        ma_fonction(dans_appelant)
```

on peut se demander quelle est exactement la nature de la relation entre l'appelant et l'appelé, c'est-à-dire ici `dans_appelant` et `dans_fonction`.

C'est l'objet de ce complément.

Passage par valeur - passage par référence

Si vous avez appris d'autres langages de programmation comme C ou C++, on a pu vous parler de deux modes de passage de paramètres : * par valeur : cela signifie qu'on communique à la fonction, non pas l'entité dans l'appelant, mais seulement **sa valeur** ; en clair, **une copie** ; * par référence : cela signifie qu'on passe à la fonction une **référence** à l'argument dans l'appelant, donc essentiellement les deux codes **partagent** la même mémoire.

Python fait du passage par référence

Certains langages comme Pascal - et C++ si on veut - proposent ces deux modes. En python, tous les passages de paramètres se font **par référence**.

```
In [ ]: # chargeons la magie pour pythontutor
        %load_ext ipythontutor
```

```
In [ ]: %%ipythontutor curInstr=4
        def ma_fonction(dans_fonction):
            print(dans_fonction)

        dans_appelant = ["texte"]
        ma_fonction(dans_appelant)
```

Ce qui signifie qu'on peut voir le code ci-dessus comme étant - pour simplifier - équivalent à ceci :

```
In [ ]: dans_appelant = ["texte"]

        # ma_fonction (dans_appelant)
        # → on entre dans la fonction
        dans_fonction = dans_appelant
        print(dans_fonction)
```

On peut le voir encore d'une autre façon en instrumentant le code comme ceci -- on rappelle que la fonction built-in `id` retourne l'adresse mémoire d'un objet :

```
In [ ]: def ma_fonction(dans_fonction):
            print('dans ma_fonction', dans_fonction , id(dans_fonction))

        dans_appelant = ["texte"]
        print('dans appelant   ', dans_appelant, id(dans_appelant))
        ma_fonction(dans_appelant)
```

Des références partagées

On voit donc que l'appel de fonction crée des références partagées, exactement comme l'affectation, et que tout ce que nous avons vu au sujet des références partagées s'applique exactement à l'identique :

```
In [ ]: # on ne peut pas modifier un immuable dans une fonction
        def increment(n):
            n += 1

        compteur = 10
        increment(compteur)
        print(compteur)
```

```
In [ ]: # on peut par contre ajouter dans une liste
        def insert(liste, valeur):
            liste.append(valeur)

        liste = ["un"]
        insert(liste, "texte")
        print(liste)
```

Pour cette raison, il est important de bien préciser, quand vous documentez une fonction, si elle fait des effets de bord sur ses arguments (c'est-à-dire qu'elle modifie ses arguments), ou si elle produit une copie. Rappelez-vous par exemple le cas de la méthode `sort` sur les listes, et de la fonction de commodité `sorted`, que nous avions vues en semaine 2.

De cette façon, on saura s'il faut ou non copier l'argument avant de le passer à votre fonction.

4.2 Rappels sur *docstring*

4.2.1 Complément - niveau basique

Comment documenter une fonction

Pour rappel, il est recommandé de toujours documenter les fonctions en ajoutant une chaîne comme première instruction.

```
In [ ]: def flatten(containers):
            "returns a list of the elements of the elements in containers"
            return [element for container in containers for element in container]
```

Cette information peut être consultée, soit interactivement :

```
In [ ]: help(flatten)
```

Soit programmativement :

```
In [ ]: flatten.__doc__
```

Sous quel format ?

L'usage est d'utiliser une chaîne simple (délimitée par « " » ou « ' ») lorsque le *docstring* tient sur une seule ligne, comme ci-dessus.

Lorsque ce n'est pas le cas - et pour du vrai code, c'est rarement le cas - on utilise des chaînes multi-lignes (délimitées par « """ » ou « ''' »). Dans ce cas le format est très flexible, car le *docstring* est normalisé, comme on le voit sur ces deux exemples, où le rendu final est identique :

```
In [ ]: # un style de docstring multi-lignes
        def flatten(containers):
            """
        provided that containers is a list (or more generally an iterable)
        of elements that are themselves iterables, this function
        returns a list of the items in these elements
            """
            return [element for container in containers for element in container]

        help(flatten)
```

```
In [ ]: # un autre style, qui donne le même résultat
        def flatten(containers):
            """
            provided that containers is a list (or more generally an iterable)
            of elements that are themselves iterables, this function
            returns a list of the items in these elements
            """
            return [element for container in containers for element in container]

        help(flatten)
```

Quelle information ?

On remarquera que dans ces exemples, le *docstring* ne répète pas le nom de la fonction ou des arguments (en mots savants, sa *signature*), et que ça n'empêche pas `help` de nous afficher cette information.

Le PEP 257 qui donne les conventions autour du *docstring* précise bien ceci :

> The one-line docstring should NOT be a "signature" reiterating the function/method parameters (which can be obtained by introspection). Don't do :

```
def function(a, b):
    """function(a, b) -> list"""


    <...>
```

The preferred form for such a docstring would be something like :

```
def function(a, b):
    """Do X and return a list."""
```

(Of course "Do X" should be replaced by a useful description !)

Pour en savoir plus

Vous trouverez tous les détails sur *docstring* dans le PEP 257.

4.3 `isinstance`

4.3.1 Complément - niveau basique

Typage dynamique

En première semaine, nous avons rapidement mentionné les concepts de typage statique et dynamique.

Avec la fonction prédéfinie `isinstance` - qui peut être par ailleurs utile dans d'autres contextes - vous pouvez facilement : * vérifier qu'un argument d'une fonction a bien le type attendu, * et traiter différemment les entrées selon leur type.

Voyons tout de suite sur un exemple simple comment on pourrait définir une fonction qui travaille sur un entier, mais qui par commmodité peut aussi accepter un entier passé comme une chaîne de caractères, ou même une liste d'entiers (auquel cas on renvoie la liste des factorielles) :

```
In [ ]: def factoriel(argument):
            # si on reçoit un entier
            if isinstance(argument, int):              # (*)
                return 1 if argument <= 1 else argument * factoriel(argument - 1)
            # convertir en entier si on reçoit une chaîne
            elif isinstance(argument, str):
                return factoriel(int(argument))
            # la liste des résultats si on reçoit un tuple ou une liste
            elif isinstance(argument, (tuple, list)):  # (**)
                return [factoriel(i) for i in argument]
            # sinon on lève une exception
            else:
                raise TypeError(argument)
```

```
In [ ]: print("entier", factoriel(4))
        print("chaine", factoriel("8"))
        print("tuple", factoriel((4, 8)))
```

Remarquez que la fonction `isinstance` **possède elle-même** une logique de ce genre, puisqu'en ligne 3 `(*)` nous lui avons passé en deuxième argument un type (`int`), alors qu'en ligne 11 `(**)` on lui a passé un tuple de deux types. Dans ce second cas naturellement, elle vérifie si l'objet (le premier argument) est **de l'un des types** mentionnés dans le tuple.

4.3.2 Complément - niveau intermédiaire

Le module `types`

Le module `types` définit un certain nombre de constantes qui peuvent être utiles dans ce contexte - vous trouverez une liste exhaustive à la fin de ce notebook. Par exemple :

```
In [ ]: from types import FunctionType
        isinstance(factoriel, FunctionType)
```

Mais méfiez vous toutefois des fonctions *built-in*, qui sont de type `BuiltinFunctionType`

```
In [ ]: from types import BuiltinFunctionType
        isinstance(len, BuiltinFunctionType)
```

```
In [ ]: # alors qu'on pourrait penser que
        isinstance(len, FunctionType)
```

`isinstance` *vs* `type`

Il est recommandé d'utiliser `isinstance` par rapport à la fonction `type`. Tout d'abord, cela permet, on vient de le voir, de prendre en compte plusieurs types.

Mais aussi et surtout `isinstance` supporte la notion d'héritage qui est centrale dans le cadre de la programmation orientée objet, sur laquelle nous allons anticiper un tout petit peu par rapport aux présentations de la semaine prochaine.

Avec la programmation objet, vous pouvez définir vos propres types. On peut par exemple définir une classe `Animal` qui convient pour tous les animaux, puis définir une sous-classe `Mammifere`. On dit que la classe `Mammifere` *hérite* de la classe `Animal`, et on l'appelle sous-classe parce qu'elle représente une partie des animaux ; et donc tout ce qu'on peut faire sur les animaux peut être fait sur les mammifères.

En voici une implémentation très rudimentaire, uniquement pour illustrer le principe de l'héritage. Si ce qui suit vous semble difficile à comprendre, pas d'inquiétude, nous reviendrons sur ce sujet lorsque nous parlerons des classes.

```
In [ ]: class Animal:
            def __init__(self, name):
                self.name = name

        class Mammifere(Animal):
            def __init__(self, name):
                Animal.__init__(self, name)
```

Ce qui nous intéresse dans l'immédiat c'est que `isinstance` permet dans ce contexte de faire des choses qu'on ne peut pas faire directement avec la fonction `type`, comme ceci :

```
In [ ]: # c'est comme ceci qu'on peut créer un objet de type `Animal` (méthode __init__)
        requin = Animal('requin')
        # idem pour un Mammifere
        baleine = Mammifere('baleine')

        # bien sûr ici la réponse est 'True'
        print("l'objet baleine est-il un mammifere ?", isinstance(baleine, Mammifere))
```

```
In [ ]: # ici c'est moins évident, mais la réponse est 'True' aussi
        print("l'objet baleine est-il un animal ?", isinstance(baleine, Animal))
```

Vous voyez qu'ici, bien que l'objet baleine est de type `Mammifere`, on peut le considérer comme étant **aussi** de type `Animal`.

Ceci est motivé de la façon suivante : comme on l'a dit plus haut, tout ce qu'on peut faire (en termes notamment d'envoi de méthodes) sur un objet de type `Animal`, on peut le faire sur un objet de type `Mammifere`. Dit en termes ensemblistes, l'ensemble des mammifères est inclus dans l'ensemble des animaux.

Annexe - Les symboles du module `types`

Vous pouvez consulter la documentation du module `types`.

```
In [ ]: # voici par ailleurs la liste de ses attributs
        import types
        dir(types)
```

4.4 *Type hints*

4.4.1 Complément - niveau intermédiaire

Langages compilés

Nous avons évoqué en première semaine le typage, lorsque nous avons comparé python avec les langages compilés. Dans un langage compilé avec typage statique, on **doit fournir du typage**, ce qui fait qu'on écrit typiquement une fonction comme ceci :

```
int factoriel(int n) {
  return (n<=1) ? 1 : n * factoriel(n-1);
}
```

ce qui signifie que la fonction factoriel prend un premier argument qui est un entier, et qu'elle retourne également un entier.

Nous avons vu également que, par contraste, pour écrire une fonction en python, on n'a **pas besoin** de préciser **le type** des arguments ni du retour de la fonction.

Vous pouvez aussi typer votre code python

Cependant depuis la version 3.5, python supporte un mécanisme **totalement optionnel** qui vous permet d'annoter les arguments des fonctions avec des informations de typage, ce mécanisme est connu sous le nom de *type hints*, et ça se présente comme ceci :

typer une variable

```
In [ ]: # pour typer une variable avec les type hints
        nb_items : int = 0
```

```
In [ ]: nb_items
```

typer les paramètres et le retour d'une fonction

```
In [ ]: # une fonction factorielle avec des type hints
        def fact(n : int) -> int:
            return 1 if n <= 1 else n * fact(n-1)
```

```
In [ ]: fact(12)
```

Usages

À ce stade, on peut entrevoir les usages suivants à ce type d'annotation :

— tout d'abord, et évidemment, cela peut permettre de mieux documenter le code ;
— les environnements de développement sont susceptibles de vous aider de manière plus effective ; si quelque part vous écrivez `z = fact(12)`, le fait de savoir que `z` est entier permet de fournir une complétion plus pertinente lorsque vous commencez à écrire `z.[TAB]` ;
— on peut espérer trouver des erreurs dans les passages d'arguments à un stade plus précoce du développement.

Par contre ce qui est très très clairement annoncé également, c'est que ces informations de typage sont **totalement facultatives**, et que le langage les **ignore totalement**.

```
In [ ]: # l'interpréteur ignore totalement ces informations
        def fake_fact(n : str) -> str:
            return 1 if n <= 1 else n * fake_fact(n-1)

        # on peut appeler fake_fact avec un int alors
        # que c'est déclaré pour des str
        fake_fact(12)
```

Le modèle préconisé est d'utiliser des **outils extérieurs**, qui peuvent faire une analyse statique du code pour exploiter ces informations à des fins de validation. Dans cette catégorie, le plus célèbre est sans doute `mypy`. Notez aussi que les IDE comme PyCharm sont également capables de tirer parti de ces annotations.

Est-ce répandu ?

Parce qu'ils ont été introduits pour la première fois avec python-3.5, en 2015 donc, puis améliorés dans la 3.6 pour le typage des variables, l'usage des *type hints* n'est pour l'instant pas très répandu, en proportion de code en tous cas. En outre, il aura fallu un temps de latence avant que tous les outils (IDE's, producteurs de documentation, outils de test, validateurs...) ne soient améliorés pour en tirer un profit maximal.

On peut penser que cet usage va se répandre avec le temps, peut-être / sans doute pas de manière systématique, mais *a minima* pour lever certaines ambigüités.

Comment annoter son code

Maintenant que nous en avons bien vu la finalité, voyons un très bref aperçu des possibilités offertes pour la construction des types dans ce contexte de *type hints*. N'hésitez pas à vous reporter à la documentation officielle du module `typing` pour un exposé plus exhaustif.

le module `typing` L'ensemble des symboles que nous allons utiliser dans la suite de ce complément provient du module `typing`

exemples simples

```
In [ ]: from typing import List
```

```
In [ ]: # une fonction qui
        # attend un paramètre qui soit une liste d'entiers,
        # et qui retourne une liste de chaines
        def foo(x: List[int]) -> List[str]:
            pass
```

avertissement : `list` vs `List` Remarquez bien dans l'exemple ci-dessus que nous avons utilisé `typing.List` plutôt que le type builtin `list`, alors que l'on a pu par contre utiliser `int` et `str`.

Les raisons pour cela sont de deux ordres :

— tout d'abord, si je devais utiliser `list` pour construire un type comme *liste d'entiers*, il me faudrait écrire quelque chose comme `list(int)` ou encore `list[int]`, et cela serait source de confusion car ceci a déjà une signification dans le langage ;

— de manière plus profonde, il faut distinguer entre `list` qui est un type concret (un objet qui sert à construire des instances), de `List` qui dans ce contexte doit plus être vu comme un type abstrait.

Pour bien voir cela, considérez l'exemple suivant :

```
In [ ]: from typing import Iterable

In [ ]: def lower_split(sep: str, inputs : Iterable[str]) -> str:
            return sep.join([x.lower() for x in inputs])

In [ ]: lower_split('--', ('AB', 'CD', 'EF'))
```

On voit bien dans cet exemple que `Iterable` ne correspond pas à un type concret particulier, c'est un type abstrait dans le sens du *duck typing*.

un exemple plus complet Voici un exemple tiré de la documentation du module `typing` qui illustre davantage de types construits à partir des types *builtin* du langage :

```
In [ ]: from typing import Dict, Tuple, List

        ConnectionOptions = Dict[str, str]
        Address = Tuple[str, int]
        Server = Tuple[Address, ConnectionOptions]

        def broadcast_message(message: str, servers: List[Server]) -> None:
            ...

        # The static type checker will treat the previous type signature as
        # being exactly equivalent to this one.
        def broadcast_message(
                message: str,
                servers: List[Tuple[Tuple[str, int], Dict[str, str]]]) -> None:
            ...
```

J'en profite d'ailleurs (ça n'a rien a voir, mais...) pour vous signaler un objet python assez étrange :

```
In [ ]: # L'objet ... existe bel et bien en python
        el = ...
        el
```

qui sert principalement pour le slicing multi-dimensionnel de numpy. Mais ne nous égarons pas...

typage partiel Puisque c'est un mécanisme optionnel, vous pouvez tout à fait ne typer qu'une partie de vos variables et paramètres :

```
In [ ]: # imaginez que vous ne typez pas n2, ni la valeur de retour

        # c'est équivalent de dire ceci
        def partially_typed(n1: int, n2):
            return None
```

```
In [ ]: # ou cela
        from typing import Any

        def partially_typed(n1: int, n2: Any) -> Any:
            return None
```

aliases On peut facilement se définir des alias ; lorsque vous avez implémenté un système d'identifiants basé sur le type `int`, il est préférable de faire :

```
In [ ]: from typing import NewType

        UserId = NewType('UserId', int)

        user1_id : UserId = 0
```

plutôt que ceci, qui est beaucoup moins parlant :

```
In [ ]: user1_id : int = 0
```

4.4.2 Complément - niveau avancé

`Generic` Pour ceux qui connaissent déjà la notion de classe (les autres peuvent ignorer la fin de ce complément) :

Grâce aux constructions `TypeVar` et `Generic`, il est possible de manipuler une notion de *variable de type*, que je vous montre sur un exemple tiré à nouveau de la documentation du module `typing` :

```
In [ ]: from typing import TypeVar, Generic
        from logging import Logger

        T = TypeVar('T')

        class LoggedVar(Generic[T]):
            def __init__(self, value: T, name: str, logger: Logger) -> None:
                self.name = name
                self.logger = logger
                self.value = value

            def set(self, new: T) -> None:
                self.log('Set ' + repr(self.value))
                self.value = new

            def get(self) -> T:
```

```
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

qui vous donne je l'espère une idée de ce qu'il est possible de faire, et jusqu'où on peut aller avec les *type hints*. Si vous êtes intéressé par cette feature je vous invite à poursuivre la lecture ici.

Pour en savoir plus

— la documentation officielle sur le module typing ;
— la page d'accueil de l'outil mypy.
— le PEP-525 sur le typage des paramètres et retours de fonctions, implémenté dans python-3.5 ;
— le PEP-526 sur le typage des variables, implémenté dans 3.6.

4.5 Conditions & Expressions Booléennes

4.5.1 Complément - niveau basique

Nous présentons rapidement dans ce notebook comment construire la condition qui contrôle l'exécution d'un `if`.

Tests considérés comme vrai

Lorsqu'on écrit une instruction comme

```
if <expression>:
   <do_something>
```

le résultat de l'expression peut **ne pas être un booléen**.
Par exemple, pour n'importe quel type numérique, la valeur 0 est considérée comme fausse. Cela signifie que

```
In [ ]: # ici la condition s'évalue à 0, donc on ne fait rien
        if 3 - 3:
            print("ne passera pas par là")
```

```
In [ ]: # par contre si vous vous souvenez de notre cours sur les flottants
        # ici la condition donne un tout petit réel mais pas 0.
        if 0.1 + 0.2 - 0.3:
            print("par contre on passe ici")
```

De même, une chaîne vide, une liste vide, un tuple vide, sont considérés comme faux. Bref, vous voyez l'idée générale.

```
In [ ]: if "":
            print("ne passera pas par là")
        if []:
            print("ne passera pas par là")
        if ():
            print("ne passera pas par là")
```

```
In [ ]: # assez logiquement, None aussi
        # est considéré comme faux
        if None:
            print("ne passe toujours pas par ici")
```

Égalité

Les tests les plus simples se font à l'aide des opérateurs d'égalité, qui fonctionnent sur presque tous les objets. L'opérateur == vérifie si deux objets ont la même valeur :

```
In [ ]: bas = 12
        haut = 25.82

        # égalité ?
        if bas == haut:
            print('==')
```

```
In [ ]: # non égalité ?
        if bas != haut:
            print('!=')
```

En genéral, deux objets de types différents ne peuvent pas être égaux.

```
In [ ]: # ces deux objets se ressemblent
        # mais ils ne sont pas du même type !
        if [1, 2] != (1, 2):
            print('!=')
```

Par contre, des `float`, des `int` et des `complex` peuvent être égaux entre eux :

```
In [ ]: bas_reel = 12.
```

```
In [ ]: print(bas, bas_reel)
```

```
In [ ]: # le réel 12 et
        # l'entier 12 sont égaux
        if bas == bas_reel:
            print('int == float')
```

```
In [ ]: # ditto pour int et complex
        if (12 + 0j) == 12:
            print('int == complex')
```

Signalons à titre un peu anecdotique une syntaxe ancienne : historiquement et **seulement en python2** on pouvait aussi noter <> le test de non égalité. On trouve ceci dans du code ancien mais il faut éviter de l'utiliser :

```
In [ ]: %%python2

        # l'ancienne forme de !=
        if 12 <> 25:
            print("<> est obsolete et ne fonctionne qu'en python2")
```

Les opérateurs de comparaison

Sans grande surprise on peut aussi écrire

```
In [ ]: if bas <= haut:
            print('<=')
        if bas < haut:
            print('<')

In [ ]: if haut >= bas:
            print('>=')
        if haut > bas:
            print('>')
```

À titre de curiosité, on peut même écrire en un seul test une appartenance à un intervalle, ce qui donne un code plus lisible

```
In [ ]: x = (bas + haut) / 2
        print(x)

In [ ]: # deux tests en une expression
        if bas <= x <= haut:
            print("dans l'intervalle")
```

On peut utiliser les comparaisons sur une palette assez large de types, comme par exemple avec les listes

```
In [ ]: # on peut comparer deux listes, mais ATTENTION
        [1, 2] <= [2, 3]
```

Il est parfois utile de vérifier le sens qui est donné à ces opérateurs selon le type ; ainsi par exemple sur les ensembles ils se réfèrent à l'**inclusion**.

Il faut aussi se méfier avec les types numériques, si un complexe est impliqué, comme dans l'exemple suivant :

```
In [ ]: # on ne peut pas par contre comparer deux nombres complexes
        try:
            2j <= 3j
        except Exception as e:
            print("OOPS", type(e), e)
```

Connecteurs logiques et / ou / non

On peut bien sûr combiner facilement plusieurs expressions entre elles, grâce aux opérateurs `and`, `or` et `not`

```
In [ ]: # il ne faut pas faire ceci, mettez des parenthèses
        if 12 <= 25. or [1, 2] <= [2, 3] and not 12 <= 32:
            print("OK mais pourrait être mieux")
```

En termes de priorités : le plus simple si vous avez une expression compliquée reste de mettre les parenthèses qui rendent son évaluation claire et lisible pour tous. Aussi on préfèrera de beaucoup la formulation équivalente :

```
In [ ]: # c'est mieux avec un parenthésage
        if 12 <= 25. or ([1, 2] <= [2, 3] and not 12 <= 32):
            print("OK, c'est équivalent et plus clair")

In [ ]: # attention, si on fait un autre parenthésage, on change le sens
        if (12 <= 25. or [1, 2] <= [2, 3]) and not 12 <= 32 :
            print("ce n'est pas équivalent, ne passera pas par là")
```

Pour en savoir plus

Reportez-vous à la section sur les opérateurs booléens dans la documentation python.

4.6 Évaluation des tests

4.6.1 Complément - niveau basique

Quels tests sont évalués ?

On a vu dans la vidéo que l'instruction conditionnelle `if` permet d'implémenter simplement des branchements à plusieurs choix, comme dans cet exemple :

```
In [ ]: s = 'berlin'
        if 'a' in s:
            print('avec a')
        elif 'b' in s:
            print('avec b')
        elif 'c' in s:
            print('avec c')
        else:
            print('sans a ni b ni c')
```

Comme on s'en doute, les expressions conditionnelles **sont évaluées jusqu'à obtenir un résultat vrai** - ou considéré comme vrai -, et le bloc correspondant est alors exécuté. Le point important ici est qu'**une fois qu'on a obtenu un résultat vrai**, on sort de l'expression conditionnelle **sans évaluer les autres conditions**. En termes savant, on parle d'évaluation paresseuse : on s'arrête dès qu'on peut.

Dans notre exemple, on aura évalué à la sortie `'a' in s`, et aussi `'b' in s`, mais pas `'c' in s`

Pourquoi c'est important ?

C'est important de bien comprendre quels sont les tests qui sont réellement évalués pour deux raisons :

— d'abord, pour des raisons de performance ; comme on n'évalue que les tests nécessaires, si un des tests prend du temps, il est peut-être préférable de le faire en dernier ;
— mais aussi et surtout, il se peut tout à fait qu'un test fasse des **effets de bord**, c'est-à-dire qu'il modifie un ou plusieurs objets.

Dans notre premier exemple, les conditions elles-mêmes sont inoffensives ; la valeur de `s` reste *identique*, que l'on *évalue ou non* les différentes conditions.

Mais nous allons voir ci-dessous qu'il est relativement facile d'écrire des conditions qui **modifient** par **effet de bord** les objets mutables sur lesquelles elles opèrent, et dans ce cas il est crucial de bien assimiler la règle des évaluations des expressions dans un `if`.

4.6.2 Complément - niveau intermédiaire

Rappel sur la méthode pop

Pour illustrer la notion d'**effet de bord**, nous revenons sur la méthode de liste `pop()` qui, on le rappelle, renvoie un élément de liste **après l'avoir effacé** de la liste.

```
In [ ]: # on se donne une liste
        liste = ['premier', 'deuxieme', 'troisieme']
        print(f"liste={liste}")

In [ ]: # pop(0) renvoie le premier élément de la liste, et raccourcit la liste
        element = liste.pop(0)
        print(f"après pop(0), element={element} et liste={liste}")

In [ ]: # et ainsi de suite
        element = liste.pop(0)
        print(f"après pop(0), element={element} et liste={liste}")
```

Conditions avec effet de bord

Une fois ce rappel fait, voyons maintenant l'exemple suivant :

```
In [ ]: liste = list(range(5))
        print('liste en entree:', liste, 'de taille', len(liste))

In [ ]: if liste.pop(0) <= 0:
            print('cas 1')
        elif liste.pop(0) <= 1:
            print('cas 2')
        elif liste.pop(0) <= 2:
            print('cas 3')
        else:
            print('cas 4')
        print('liste en sortie de taille', len(liste))
```

Avec cette entrée, le premier test est vrai (car `pop(0)` renvoie 0), aussi on n'exécute en tout `pop()` qu'**une seule fois**, et donc à la sortie la liste n'a été raccourcie que d'un élément.

Exécutons à présent le même code avec une entrée différente :

```
In [ ]: liste = list(range(5, 10))
        print('en entree: liste=', liste, 'de taille', len(liste))

In [ ]: if liste.pop(0) <= 0:
            print('cas 1')
        elif liste.pop(0) <= 1:
            print('cas 2')
        elif liste.pop(0) <= 2:
            print('cas 3')
        else:
            print('cas 4')
        print('en sortie: liste=', liste, 'de taille', len(liste))
```

On observe que cette fois la liste a été **raccourcie 3 fois**, car les trois tests se sont révélés faux.

Cet exemple vous montre qu'il faut être attentif avec des conditions qui font des effets de bord. Bien entendu, ce type de pratique est de manière générale à utiliser avec beaucoup de discernement.

Court-circuit (*short-circuit*)

La logique que l'on vient de voir est celle qui s'applique aux différentes branches d'un `if` ; c'est la même logique qui est à l'œuvre aussi lorsque python évalue une condition logique à base de `and` et `or`. C'est ici aussi une forme d'évaluation paresseuse.

Pour illustrer cela, nous allons nous définir deux fonctions toutes simples qui renvoient `True` et `False` mais avec une impression de sorte qu'on voit lorsqu'elles sont exécutées :

```
In [ ]: def true():
            print('true')
            return True

In [ ]: def false():
            print('false')
            return False

In [ ]: true()
```

Ceci va nous permettre d'illustrer notre point, qui est que lorsque python évalue un `and` ou un `or`, il **n'évalue la deuxième condition que si c'est nécessaire**. Ainsi par exemple :

```
In [ ]: false() and true()
```

Dans ce cas, python évalue la première partie du `and` - qui provoque l'impression de `false` - et comme le résultat est faux, il n'est **pas nécessaire** d'évaluer la seconde condition, on sait que de toute façon le résultat du `and` est forcément faux. C'est pourquoi vous ne voyez pas l'impression de `true`.

De manière symétrique avec un `or` :

```
In [ ]: true() or false()
```

À nouveau ici il n'est pas nécessaire d'évaluer `false()`, et donc seul `true` est imprimé à l'évaluation.

À titre d'exercice, essayez de dire combien d'impressions sont émises lorsqu'on évalue cette expression un peu plus compliquée :

```
In [ ]: true() and (false() or true()) or (true () and false())
```

4.7 Une forme alternative du `if`

4.7.1 Complément - niveau basique

Expressions et instructions

Les constructions python que nous avons vues jusqu'ici peuvent se ranger en deux familles :

— d'une part les **expressions** sont les fragments de code qui **retournent une valeur** ;

— c'est le cas lorsqu'on invoque n'importe quel opérateur numérique, pour les appels de fonctions, ...
— d'autre part les **instructions** ;
— dans cette famille, nous avons vu par exemple l'affectation et `if`, et nous en verrons bien d'autres.

La différence essentielle est que les expressions peuvent être combinées entre elles pour faire des expressions arbitrairement grosses. Aussi, si vous avez un doute pour savoir si vous avez affaire à une expression ou à une instruction, demandez vous si vous pourriez utiliser ce code **comme membre droit d'une affectation**. Si oui, vous avez une expression.

`if` est une instruction

La forme du `if` qui vous a été présentée pendant la vidéo ne peut pas servir à renvoyer une valeur, c'est donc une **instruction**.

Imaginons maintenant qu'on veuille écrire quelque chose d'aussi simple que *"affecter à y la valeur 12 ou 35, selon que x est vrai ou non"*.

Avec les notions introduites jusqu'ici, il nous faudrait écrire ceci :

```
In [ ]: x = True  # ou quoi que ce soit d'autre
        if x:
            y = 12
        else:
            y = 35
        print(y)
```

Expression conditionnelle

Il existe en python une expression qui fait le même genre de test ; c'est la forme dite d'**expression conditionnelle**, qui est une **expression à part entière**, avec la syntaxe :

```
<resultat_si_vrai> if <condition> else <resultat_si_faux>
```

Ainsi on pourrait écrire l'exemple ci-dessus de manière plus simple et plus concise comme ceci :

```
In [ ]: y = 12 if x else 35
        print(y)
```

Cette construction peut souvent rendre le style de programmation plus fonctionnel et plus fluide.

4.7.2 Complément - niveau intermédiaire

Imbrications

Puisque cette forme est une expression, on peut l'utiliser dans une autre expression conditionnelle, comme ici :

```
In [ ]: # on veut calculer en fonction d'une entrée x
        # une sortie qui vaudra
        # -1 si x < -10
        # 0 si -10 <= x <= 10
        # 1 si x > 10
```

```
x = 5 # ou quoi que ce soit d'autre

valeur = -1 if x < -10 else (0 if x <= 10 else 1)

print(valeur)
```

Remarquez bien que cet exemple est équivalent à la ligne

```
valeur = -1 if x < -10 else 0 if x <= 10 else 1
```

mais qu'il est fortement recommandé d'utiliser, comme on l'a fait, un parenthésage pour lever toute ambigüité.

Pour en savoir plus

— La section sur les expressions conditionnelles de la documentation python.
— Le PEP308 qui résume les discussions ayant donné lieu au choix de la syntaxe adoptée.

De manière générale, les PEP rassemblent les discussions préalables à toutes les évolutions majeures du langage python.

4.8 Récapitulatif sur les conditions dans un `if`

4.8.1 Complément - niveau basique

Dans ce complément nous résumons ce qu'il faut savoir pour écrire une condition dans un `if`.

Expression *vs* instruction

Nous avons déjà introduit la différence entre instruction et expression, lorsque nous avons vu l'expression conditionnelle : * une expression est un fragment de code qui "retourne quelque chose", * alors qu'une instruction permet bien souvent de faire une action, mais ne retourne rien.

Ainsi parmi les notions que nous avons vues jusqu'ici, nous pouvons citer dans un ordre arbitraire :

Instructions	Expressions
affectation	appel de fonction
`import`	opérateurs `is`, `in`, `==`, ...
instruction `if`	expression conditionnelle
instruction `for`	compréhension(s)

Toutes les expressions sont éligibles

Comme condition d'une instruction `if`, on peut mettre n'importe quelle expression. On l'a déjà signalé, il n'est pas nécessaire que cette expression retourne un booléen :

```
In [ ]: # dans ce code le test
        # if n % 3:
        # est équivalent à
```

```
        # if n % 3 != 0:

        for n in (18, 19):
            if n % 3:
                print(f"{n} non divisible par trois")
            else:
                print(f"{n} divisible par trois")
```

Une valeur est-elle "vraie" ?

Se pose dès lors la question de savoir précisément quelles valeurs sont considérées comme *vraies* par l'instruction `if`.

Parmi les types de base, nous avons déjà eu l'occasion de l'évoquer, les valeurs *fausses* sont typiquement : * 0 pour les valeurs numériques ; * les objets vides pour les chaînes, listes, ensembles, dictionnaires, etc.

Pour en avoir le cœur net, pensez à utiliser dans le terminal interactif la fonction `bool`. Comme pour toutes les fonctions qui portent le nom d'un type, la fonction `bool` est un constructeur qui fabrique un objet booléen.

Si vous appelez `bool` sur un objet, la valeur de retour - qui est donc par construction une valeur booléenne - vous indique, cette fois sans ambigüité - comment se comportera `if` avec cette entrée.

```
In [ ]: def show_bool(x):
            print(f"condition {repr(x):>10} considérée comme {bool(x)}")

In [ ]: for exp in [None, "", 'a', [], [1], (), (1, 2), {}, {'a': 1}, set(), {1}]:
            show_bool(exp)
```

Quelques exemples d'expressions

Référence à une variable et dérivés

```
In [ ]: a = list(range(4))
        print(a)

In [ ]: if a:
            print("a n'est pas vide")
        if a[0]:
            print("on ne passe pas par ici")
        if a[1]:
            print("a[1] n'est pas nul")
```

Appels de fonction ou de méthode

```
In [ ]: chaine = "jean"
        if chaine.upper():
            print("la chaine mise en majuscule n'est pas vide")

In [ ]: # on rappelle qu'une fonction qui ne fait pas 'return' retourne None
        def procedure(a, b, c):
            "cette fonction ne retourne rien"
            pass
```

```
if procedure(1, 2, 3):
    print("ne passe pas ici car procedure retourne None")
else:
    print("par contre on passe ici")
```

Compréhensions Il découle de ce qui précède qu'on peut tout à fait mettre une compréhension comme condition, ce qui peut être utile pour savoir si au moins un élément remplit une condition, comme par exemple :

```
In [ ]: inputs = [23, 65, 24]

        # y a-t-il dans inputs au moins un nombre
        # dont le carré est de la forme 10*n+5
        def condition(n):
            return (n * n) % 10 == 5

        if [value for value in inputs if condition(value)]:
            print("au moins une entrée convient")
```

Opérateurs Nous avons déjà eu l'occasion de rencontrer la plupart des opérateurs de comparaison du langage, dont voici à nouveau les principaux :

Famille	Exemples
Égalité	`==`, `!=`, `is`, `is not`
Appartenance	`in`
Comparaison	`<=`, `<`, `>`, `>=`
Logiques	`and`, `or`, `not`

4.8.2 Complément - niveau intermédiaire

Remarques sur les opérateurs

Voici enfin quelques remarques sur ces opérateurs

opérateur d'égalité == L'opérateur == ne fonctionne en général (sauf pour les nombres) que sur des objets de même type ; c'est-à-dire que notamment un tuple ne sera jamais égal à une liste :

```
In [ ]: [] == ()

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

opérateur logiques Comme c'est le cas avec par exemple les opérateurs arithmétiques, les opérateurs logiques ont une *priorité*, qui précise le sens des phrases non parenthésées. C'est-à-dire pour être explicite, que de la même manière que

```
12 + 4 * 8
```

est équivalent à

```
12 + ( 4 * 8 )
```

pour les booléens il existe une règle de ce genre et

```
a and not b or c and d
```

est équivalent à

```
(a and (not b)) or (c and d)
```

Mais en fait, il est assez facile de s'emmêler dans ces priorités, et c'est pourquoi il est **très fortement conseillé** de parenthéser.

opérateurs logiques (2) Remarquez aussi que les opérateurs logiques peuvent être appliqués à des valeurs qui ne sont pas booléennes :

```
In [ ]: 2 and [1, 2]
```

```
In [ ]: None or "abcde"
```

Dans la logique de l'évaluation paresseuse qu'on a vue récemment, remarquez que lorsque l'évaluation d'un `and` ou d'un `or` ne peut pas être court-circuitée, le résultat est alors toujours le résultat de la dernière expression évaluée :

```
In [ ]: 1 and 2 and 3
```

```
In [ ]: 1 and 2 and 3 and '' and 4
```

```
In [ ]: [] or "" or {}
```

```
In [ ]: [] or "" or {} or 4 or set()
```

Expression conditionnelle dans une instruction `if`

En toute rigueur on peut aussi mettre un `<> if <> else <>` - donc une expression conditionnelle - comme condition dans une instruction `if`. Nous le signalons pour bien illustrer la logique du langage, mais cette pratique n'est bien sûr pas du tout conseillée.

```
In [ ]: # cet exemple est volontairement tiré par les cheveux
        # pour bien montrer qu'on peut mettre n'importe quelle expression comme condition
        a = 1
        # ceci est franchement illisible
        if 0 if not a else 2:
            print("une construction illisible")
        # et encore pire
        if 0 if a else 3 if a + 1 else 2:
            print("encore pire")
```

Pour en savoir plus

https ://docs.python.org/3/tutorial/datastructures.html#more-on-conditions

Types définis par l'utilisateur

Pour anticiper un tout petit peu, nous verrons que les classes en python vous donnent le moyen de définir vos propres types d'objets. Nous verrons à cette occasion qu'il est possible d'indiquer à python quels sont les objets de type `MaClasse` qui doivent être considérés comme `True` ou comme `False`.

De manière plus générale, tous les traits natifs du langage sont redéfinissables sur les classes. Nous verrons par exemple également comment donner du sens à des phrases comme

```
mon_objet = MaClasse()
if mon_objet:
    <faire quelque chose>
```

ou encore

```
mon_objet = MaClasse()
for partie in mon_objet:
    <faire quelque chose sur partie>
```

Mais n'anticipons pas trop, rendez-vous en semaine 6.

4.9 Expression conditionnelle

4.9.1 Exercice - niveau basique

Analyse et mise en forme

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

Un fichier contient, dans chaque ligne, des informations (champs) séparées par des virgules. Les espaces et tabulations présents dans la ligne ne sont pas significatifs et doivent être ignorés.

Dans cet exercice de niveau basique, on suppose que chaque ligne a exactement 3 champs, qui représentent respectivement le prénom, le nom, et le rang d'une personne dans un classement. Une fois les espaces et tabulations ignorés, on ne fait pas de vérification sur le contenu des 3 champs.

On vous demande d'écrire la fonction `libelle`, qui sera appelée pour chaque ligne du fichier. Cette fonction : * prend en argument une ligne (chaîne de caractères) * retourne une chaîne de caractères mise en forme (voir plus bas) * ou bien retourne `None` si la ligne n'a pas pu être analysée, parce qu'elle ne vérifie pas les hypothèses ci-dessus (c'est notamment le cas si on ne trouve pas exactement les 3 champs)

La mise en forme consiste à retourner

```
Nom.Prenom (message)
```

le *message* étant lui-même le *rang* mis en forme pour afficher '1er', '2nd' ou '*n*-ème' selon le cas. Voici quelques exemples

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

```
In [ ]: # écrivez votre code ici
        def libelle(ligne):
            "<votre_code>"

In [ ]: # pour le vérifier
        exo_libelle.correction(libelle)
```

4.10 La boucle `while` ... `else`

4.10.1 Complément - niveau basique

Boucles sans fin - `break`

Utiliser `while` plutôt que `for` est une affaire de style et d'habitude. Cela dit en python, avec les notions d'itérable et d'itérateur, on a tendance à privilégier l'usage du `for` pour les boucles finies et déterministes.

Le `while` reste malgré tout d'un usage courant, et notamment avec une condition `True`.

Par exemple le code de l'interpreteur interactif de python pourrait ressembler, vu de très loin, à quelque chose comme ceci

```
while True:
    print(eval(read()))
```

Notez bien par ailleurs que les instructions `break` et `continue` fonctionnent, à l'intérieur d'une boucle `while`, exactement comme dans un `for`, c'est-à-dire que : * `continue` termine l'itération courante mais reste dans la boucle, alors que * `break` interrompt l'itération courante et sort également de la boucle.

4.10.2 Complément - niveau intermédiaire

Rappel sur les conditions

On peut utiliser dans une boucle `while` toutes les formes de conditions que l'on a vues à l'occasion de l'instruction `if`.

Dans le contexte de la boucle `while` on comprend mieux, toutefois, pourquoi le langage autorise d'écrire des conditions dont le résultat n'est **pas nécessairement un booléen**. Voyons cela sur un exemple simple :

```
In [ ]: # une autre façon de parcourir une liste
        liste = ['a', 'b', 'c']

        while liste:
            element = liste.pop()
            print(element)
```

Une curiosité : la clause `else`

Signalons enfin que la boucle `while` - au même titre d'ailleurs que la boucle `for`, peut être assortie d'une clause `else`, qui est exécutée à la fin de la boucle, **sauf dans le cas d'une sortie avec** `break`

```
In [ ]: # Un exemple de while avec une clause else
```

```
        # si break_mode est vrai on va faire un break
        # après le premier élément de la liste
        def scan(liste, break_mode):

            # un message qui soit un peu parlant
            message = "avec break" if break_mode else "sans break"
            print(message)
            while liste:
                print(liste.pop())
                if break_mode:
                    break
            else:
                print('else...')
```

```
In [ ]: # sortie de la boucle sans break
        # on passe par else
        scan(['a'], False)
```

```
In [ ]: # on sort de la boucle par le break
        scan(['a'], True)
```

Ce trait est toutefois **très rarement** utilisé.

4.11 Calculer le PGCD

4.11.1 Exercice - niveau basique

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

On vous demande d'écrire une fonction qui calcule le pgcd de deux entiers, en utilisant l'algorithme d'Euclide.

Les deux paramètres sont supposés être des entiers positifs ou nuls (pas la peine de le vérifier).

Dans le cas où un des deux paramètres est nul, le pgcd vaut l'autre paramètre. Ainsi par exemple :

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

Remarque on peut tout à fait utiliser une fonction récursive pour implémenter l'algorithme d'Euclide. Par exemple cette version de `pgcd` fonctionne très bien aussi (en supposant a>=b)

```
def pgcd(a, b):
   "Le pgcd avec une fonction récursive"
   if not b:
       return a
   return pgcd(b, a % b)
```

Cependant, il vous est demandé ici d'utiliser une boucle `while`, qui est le sujet de la séquence, pour implémenter `pgcd`.

```
In [ ]: # à vous de jouer
        def pgcd(a, b):
            "<votre code>"
```

```
In [ ]: # pour vérifier votre code
        exo_pgcd.correction(pgcd)
```

4.12 Exercice

4.12.1 Niveau basique

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

On se propose d'écrire une fonction `taxes` qui calcule le montant de l'impôt sur le revenu au Royaume-uni.

Le barême est publié ici par le gouvernement anglais, voici les données utilisées pour l'exercice :

Tranche	Revenu imposable	Taux
Non imposable	jusque £11.500	0%
Taux de base	£11.501 à £45.000	20%
Taux élevé	£45.001 à £150.000	40%
Taux supplémentaire	au delà de £150.000	45%

Donc naturellement il s'agit d'écrire une fonction qui prend en argument le revenu imposable, et retourne le montant de l'impôt, **arrondi à l'entier inférieur**.

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

Indices

— évidemment on parle ici d'une fonction continue ;
— aussi en termes de programmation, je vous encourage à séparer la définition des tranches de la fonction en elle-même.

```
In [ ]: def taxes(income):
            # ce n'est pas la bonne réponse
            return (income-11_500) * (20/100)
```

```
In [ ]: exo_taxes.correction(taxes)
```

Représentation graphique Comme d'habitude vous pouvez voir la représentation graphique de votre fonction :

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
```

```
In [ ]: %matplotlib inline
        plt.ion()
```

```
In [ ]: X = np.linspace(0, 200_000)
        Y = [taxes(x) for x in X]
        plt.plot(X, Y);
```

```
In [ ]: # et pour changer la taille de la figure
        plt.figure(figsize=(10, 8))
        plt.plot(X, Y);
```

4.13 Le module builtins

4.13.1 Complément - niveau avancé

Ces noms qui viennent de nulle part

Nous avons vu déjà un certain nombre de **fonctions *built-in*** comme par exemple

```
In [ ]: open, len, zip
```

Ces noms font partie du **module** builtins. Il est cependant particulier puisque tout se passe **comme si** on avait fait avant toute chose :

```
from builtins import *
```

sauf que cet import est implicite.

On peut réaffecter un nom *built-in*

Quoique ce soit une pratique déconseillée, il est tout à fait possible de redéfinir ces noms ; on peut faire par exemple

```
In [ ]: # on réaffecte le nom open à un nouvel objet fonction
        def open(encoding='utf-8', *args):
            print("ma fonction open")
            pass
```

qui est naturellement **très vivement déconseillé**. Notez, cependant, que la coloration syntaxique vous montre clairement que le nom que vous utilisez est un builtins (en vert dans un notebook).

On ne peut pas réaffecter un mot clé

À titre de digression, rappelons que les noms prédéfinis dans le module builtins sont, à cet égard aussi, très différents des mots-clés comme if, def, with et autres for qui eux, ne peuvent pas être modifiés en aucune manière :

```
>>> lambda = 1
  File "<stdin>", line 1
    lambda = 1
           ^
SyntaxError: invalid syntax
```

Retrouver un objet *built-in*

Il faut éviter de redéfinir un nom prédéfini dans le module builtins ; un bon éditeur de texte vous signalera les fonctions *built-in* avec une coloration syntaxique spécifique. Cependant, on peut vouloir redéfinir un nom *built-in* pour changer un comportement par défaut, puis vouloir revenir au comportement original.

Sachez que vous pouvez toujours "retrouver" alors la fonction *built-in* en l'important explicitement du module builtins. Par exemple, pour réaliser notre ouverture de fichier, nous pouvons toujours faire :

```
In [ ]: # nous ne pouvons pas utiliser open puisque
        open()
```

```
In [ ]: # pour être sûr d'utiliser la bonne fonction open

        import builtins

        with builtins.open("builtins.txt", "w", encoding="utf-8") as f:
            f.write("quelque chose")
```

Ou encore, de manière équivalente :

```
In [ ]: from builtins import open as builtins_open

        with builtins_open("builtins.txt", "r", encoding="utf-8") as f:
            print(f.read())
```

Liste des fonctions prédéfinies

Vous pouvez trouver la liste des fonctions prédéfinies ou *built-in* avec la fonction `dir` sur le module `builtins` comme ci-dessous (qui vous montre aussi les exceptions prédéfinies, qui commencent par une majuscule), ou dans la documentation sur les fonctions prédéfinies :

```
In [ ]: dir(builtins)
```

Vous remarquez que les exceptions (les symboles qui commencent par des majuscules) représentent à elles-seules une proportion subtantielle de cet espace de noms.

4.14 Visibilité des variables de boucle

4.14.1 Complément - niveau basique

Une astuce

Dans ce complément, nous allons beaucoup jouer avec le fait qu'une variable soit définie ou non. Pour nous simplifier la vie, et surtout rendre les cellules plus indépendantes les unes des autres si vous devez les rejouer, nous allons utiliser la formule un peu magique suivante :

```
In [ ]: # on détruit la variable i si elle existe
        if 'i' in locals():
            del i
```

qui repose d'une part sur l'instruction `del` que nous avons déjà vue, et sur la fonction *builtin* `locals` que nous verrons plus tard ; cette formule a l'avantage qu'on peut l'exécuter dans n'importe quel contexte, que `i` soit définie ou non.

Une variable de boucle reste définie au-delà de la boucle

Une variable de boucle est définie (assignée) dans la boucle et **reste *visible*** une fois la boucle terminée. Le plus simple est de le voir sur un exemple :

```
In [ ]: # La variable 'i' n'est pas définie
        try:
            i
        except NameError as e:
            print('OOPS', e)
```

```
In [ ]: # si à présent on fait une boucle
        # avec i comme variable de boucle
        for i in [0]:
            pass

        # alors maintenant i est définie
        i
```

On dit que la variable *fuite* (en anglais "*leak*"), dans ce sens qu'elle continue d'exister au delà du bloc de la boucle à proprement parler.

On peut être tenté de tirer profit de ce trait, en lisant la valeur de la variable après la boucle ; l'objet de ce complément est de vous inciter à la prudence, et d'attirer votre attention sur certains points qui peuvent être sources d'erreur.

Attention aux boucles vides

Tout d'abord, il faut faire attention à ne pas écrire du code qui dépende de ce trait **si la boucle peut être vide**. En effet, si la boucle ne s'exécute pas du tout, la variable n'est **pas affectée** et donc elle n'est **pas définie**. C'est évident, mais ça peut l'être moins quand on lit du code réel, comme par exemple :

```
In [ ]: # on détruit la variable i si elle existe
        if 'i' in locals():
            del i
```

```
In [ ]: # une façon très scabreuse de calculer la longueur de l
        def length(l):
            for i, x in enumerate(l):
                pass
            return i + 1

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

Ça a l'air correct, sauf que :

```
In [ ]: length([])
```

Ce résultat mérite une explication. Nous allons voir très bientôt l'exception `UnboundLocalError`, mais pour le moment sachez qu'elle se produit lorsqu'on a dans une fonction une variable locale et une variable globale de même nom. Alors, pourquoi l'appel `length([1, 2, 3])` retourne-t-il sans encombre, alors que pour l'appel `length([])` il y a une exception ? Cela est lié à la manière dont python détermine qu'une variable est locale.

Une variable est locale dans une fonction si elle est assignée dans la fonction explicitement (avec une opération d'affectation) ou implicitement (par exemple avec une boucle `for` comme ici) ; nous reviendrons sur ce point un peu plus tard. Mais pour les fonctions, pour une raison d'efficacité, une variable est définie comme locale à la phase de pré-compilation, c'est-à-dire *avant* l'exécution du code. Le pré-compilateur ne peut pas savoir quel sera l'argument passé à la fonction, il peut simplement savoir qu'il y a une boucle `for` utilisant la variable `i`, il en conclut que `i` est locale pour toute la fonction.

Lors du premier appel, on passe une liste à la fonction, liste qui est parcourue par la boucle `for`. En sortie de boucle, on a bien une variale locale `i` qui vaut 3. Lors du deuxième appel par contre, on passe une liste vide à la fonction, la boucle `for` ne peut rien parcourir, donc elle termine immédiatement. Lorsque l'on arrive à la ligne `return i + 1` de la fonction, la variable

`i` n'a pas de valeur (on doit donc chercher `i` dans le module), mais `i` a été définie par le pré-compilateur comme étant locale, on a donc dans la même fonction une variable `i` locale et une référence à une variable `i` globale, ce qui provoque l'exception `UnboundLocalError`.

Comment faire alors ?

Utiliser une autre variable La première voie consiste à déclarer une variable externe à la boucle et à l'affecter à l'intérieur de la boucle, c'est-à-dire :

```
In [ ]: # on veut chercher le premier de ces nombres qui vérifie une condition
        candidates = [3, -15, 1, 8]

        # pour fixer les idées disons qu'on cherche un multiple de 5; peu importe
        def checks(candidate):
            return candidate % 5 == 0
```

```
In [ ]: # plutôt que de faire ceci
        for item in candidates:
            if checks(item):
                break
        print('trouvé solution', item)
```

```
In [ ]: # il vaut mieux faire ceci
        solution = None
        for item in candidates:
            if checks(item):
                solution = item
                break

        print('trouvé solution', solution)
```

Au minimum initialiser la variable Au minimum, si vous utilisez la variable de boucle après la boucle, il est vivement conseillé de l'**initialiser** explicitement **avant** la boucle, pour vous prémunir contre les boucles vides, comme ceci :

```
In [ ]: # la fonction length de tout à l'heure
        def length1(l):
            for i, x in enumerate(l):
                pass
            return i + 1
```

```
In [ ]: # une version plus robuste
        def length2(l):
            # on initialise i explicitement
            # pour le cas où l est vide
            i = -1
            for i, x in enumerate(l):
                pass
            # comme cela i est toujours déclarée
            return i + 1
```

```
In [ ]: length1([])
```

```
In [ ]: length2([])
```

Les compréhensions

Notez bien que par contre, les variables de compréhension **ne fuitent pas** (contrairement à ce qui se passait en python2) :

```
In [ ]: # on détruit la variable i si elle existe
        if 'i' in locals():
            del i

In [ ]: # en python3, les variables de compréhension ne fuitent pas
        [i**2 for i in range(3)]

In [ ]: # ici i est à nouveau indéfinie
        try:
            i
        except NameError as e:
            print("OOPS", e)
```

4.15 L'exception UnboundLocalError

4.15.1 Complément - niveau intermédiaire

Nous résumons ici quelques cas simples de portée de variables.

Variable locale

Les **arguments** attendus par la fonction sont considérés comme des variables **locales**, c'est-à-dire dans l'espace de noms de la fonction.

Pour définir une autre variable locale, il suffit de la définir (l'affecter), elle devient alors accessible en lecture :

```
In [ ]: def ma_fonction1():
            variable1 = "locale"
            print(variable1)

        ma_fonction1()
```

et ceci que l'on ait ou non une variable globale de même nom

```
In [ ]: variable2 = "globale"

        def ma_fonction2():
            variable2 = "locale"
            print(variable2)

        ma_fonction2()
```

Variable globale

On peut accéder **en lecture** à une variable globale sans précaution particulière :

```
In [ ]: variable3 = "globale"

        def ma_fonction3():
            print(variable3)

        ma_fonction3()
```

Mais il faut choisir !

Par contre on ne **peut pas** faire la chose suivante dans une fonction. On ne peut pas utiliser **d'abord** une variable comme une variable **globale**, **puis** essayer de l'affecter localement - ce qui signifie la déclarer comme une **locale** :

```
In [ ]: # cet exemple ne fonctionne pas et lève UnboundLocalError
        variable4 = "globale"

        def ma_fonction4():
            # on référence la variable globale
            print(variable4)
            # et maintenant on crée une variable locale
            variable4 = "locale"

        # on "attrape" l'exception
        try:
            ma_fonction4()
        except Exception as e:
            print(f"OOPS, exception {type(e)}:\n{e}")
```

Comment faire alors ?

L'intérêt de cette erreur est d'interdire de mélanger des variables locales et globales de même nom dans une même fonction. On voit bien que ça serait vite incompréhensible. Donc une variable dans une fonction peut être **ou bien** locale si elle est affectée dans la fonction **ou bien** globale, mais **pas les deux à la fois**. Si vous avez une erreur `UnboundLocalError`, c'est qu'à un moment donné vous avez fait cette confusion.

Vous vous demandez peut-être à ce stade, mais comment fait-on alors pour modifier une variable globale depuis une fonction ? Pour cela il faut utiliser l'instruction `global` comme ceci :

```
In [ ]: # Pour résoudre ce conflit il faut explicitement
        # déclarer la variable  comme globale
        variable5 = "globale"

        def ma_fonction5():
            global variable5
            # on référence la variable globale
            print("dans la fonction", variable5)
            # cette fois on modifie la variable globale
            variable5 = "changée localement"

        ma_fonction5()
        print("après la fonction", variable5)
```

Nous reviendrons plus longuement sur l'instruction `global` dans la prochaine vidéo.

Bonnes pratiques

Cela étant dit, l'utilisation de variables globales est généralement considérée comme une mauvaise pratique.

Le fait d'utiliser une variable globale en *lecture seule* peut rester acceptable, lorsqu'il s'agit de matérialiser une constante qu'il est facile de changer. Mais dans une application aboutie, ces constantes elles-mêmes peuvent être modifiées par l'utilisateur via un système de configuration, donc on préférera passer en argument un objet *config*.

Et dans les cas où votre code doit recourir à l'utilisation de l'instruction `global`, c'est très probablement que quelque chose peut être amélioré au niveau de la conception de votre code.

Il est recommandé, au contraire, de passer en argument à une fonction tout le contexte dont elle a besoin pour travailler ; et à l'inverse d'utiliser le résultat d'une fonction plutôt que de modifier une variable globale.

4.16 Les fonctions `globals` et `locals`

4.16.1 Complément - niveau intermédiaire

Un exemple

python fournit un accès à la liste des noms et valeurs des variables visibles à cet endroit du code. Dans le jargon des langages de programmation on appelle ceci **l'environnement**.

Cela est fait grâce aux fonctions *builtins* `globals` et `locals`, que nous allons commencer par essayer sur quelques exemples. Nous avons pour cela écrit un module dédié :

```
In [ ]: import env_locals_globals
```

Dont voici le code

```
In [ ]: from modtools import show_module
        show_module(env_locals_globals)
```

et voici ce qu'on obtient lorsqu'on appelle

```
In [ ]: env_locals_globals.temoin(10)
```

Interprétation

Que nous montre cet exemple ?

— D'une part la fonction `globals` nous donne la liste des symboles définis au niveau de **l'espace de noms du module**. Il s'agit évidemment du module dans lequel est définie la fonction, pas celui dans lequel elle est appelée. Vous remarquerez que ceci englobe **tous** les symboles du module `env_locals_globals`, et non pas seulement ceux définis avant `temoin`, c'est-à-dire la variable `globale`, les deux fonctions `display_env` et `temoin`, et la classe `Foo`.

— D'autre part `locals` nous donne les variables locales qui sont accessibles **à cet endroit du code**, comme le montre ce second exemple qui se concentre sur `locals` à différents points d'une même fonction.

```
In [ ]: import env_locals
```

```
In [ ]: # le code de ce module
        show_module(env_locals)

In [ ]: env_locals.temoin(10)
```

4.16.2 Complément - niveau avancé

NOTE : cette section est en pratique devenue obsolète maintenant que les *f-strings* sont présents dans la version 3.6.

Nous l'avons conservée pour l'instant toutefois, pour ceux d'entre vous qui ne peuvent pas encore utiliser les *f-strings* en production. N'hésitez pas à passer si vous n'êtes pas dans ce cas.

Usage pour le formatage de chaînes

Les deux fonctions `locals` et `globals` ne sont pas d'une utilisation très fréquente. Elles peuvent cependant être utiles dans le contexte du formatage de chaînes, comme on peut le voir dans les deux exemples ci-dessous.

Avec `format` On peut utiliser `format` qui s'attend à quelque chose comme :

```
In [ ]: "{nom}".format(nom="Dupont")
```

que l'on peut obtenir de manière équivalente, en anticipant sur la prochaine vidéo, avec le passage d'arguments en `**` :

```
In [ ]: "{nom}".format(**{'nom': 'Dupont'})
```

En versant la fonction `locals` dans cette formule on obtient une forme relativement élégante

```
In [ ]: def format_et_locals(nom, prenom, civilite, telephone):
            return "{civilite} {prenom} {nom} : Poste {telephone}".format(**locals())

        format_et_locals('Dupont', 'Jean', 'Mr', '7748')
```

Avec l'opérateur % De manière similaire, avec l'opérateur % - dont nous rappelons qu'il est obsolète - on peut écrire

```
In [ ]: def pourcent_et_locals(nom, prenom, civilite, telephone):
            return "%(civilite)s %(prenom)s %(nom)s : Poste %(telephone)s"%locals()

        pourcent_et_locals('Dupont', 'Jean', 'Mr', '7748')
```

Avec un *f-string* Pour rappel si vous disposez de python 3.6, vous pouvez alors écrire simplement - et sans avoir recours, donc, à `locals()` ou autre :

```
In [ ]: # attention ceci nécessite python-3.6
        def avec_f_string(nom, prenom, civilite, telephone):
            return f"{civilite} {prenom} {nom} : Poste {telephone}"

        avec_f_string('Dupont', 'Jean', 'Mr', '7748')
```

4.17 Passage d'arguments

4.17.1 Complément - niveau intermédiaire

Motivation

Jusqu'ici nous avons développé le modèle simple qu'on trouve dans tous les langages de programmation, à savoir qu'une fonction a un nombre fixe, supposé connu, d'arguments. Ce modèle a cependant quelques limitations ; les mécanismes de passage d'arguments que propose python, et que nous venons de voir dans les vidéos, visent à lever ces limitations.

Voyons de quelles limitations il s'agit.

Nombre d'arguments non connu à l'avance

Ou encore : introduction à la forme `*arguments` Pour prendre un exemple aussi simple que possible, considérons la fonction `print`, qui nous l'avons vu, accepte un nombre quelconque d'arguments.

```
In [ ]: print("la fonction", "print", "peut", "prendre", "plein", "d'arguments")
```

Imaginons maintenant que nous voulons implémenter une variante de `print`, c'est-à-dire une fonction `error`, qui se comporte exactement comme `print` sauf qu'elle ajoute en début de ligne une balise `ERROR`.

Se posent alors deux problèmes : * D'une part il nous faut un moyen de spécifier que notre fonction prend un nombre quelconque d'arguments. * D'autre part il faut une syntaxe pour repasser tous ces arguments à la fonction `print`.

On peut faire tout cela avec la notation en `*` comme ceci :

```
In [ ]: # accepter n'importe quel nombre d'arguments
        def error(*print_args):
            # et les repasser à l'identique à print en plus de la balise
            print('ERROR', *print_args)

        # on peut alors l'utiliser comme ceci
        error("problème", "dans", "la", "fonction", "foo")
        # ou même sans argument
        error()
```

Légère variation

Pour sophistiquer un peu cet exemple, on veut maintenant imposer à la fonction erreur qu'elle reçoive un argument obligatoire de type entier qui représente un code d'erreur, plus à nouveau un nombre quelconque d'arguments pour `print`.

Pour cela, on peut définir une signature (les paramètres de la fonction) qui * prévoit un argument traditionnel en première position, qui sera obligatoire lors de l'appel, * et le tuple des arguments pour `print`, comme ceci :

```
In [ ]: # le premier argument est obligatoire
        def error1(error_code, *print_args):
            message = f"message d'erreur code {error_code}"
            print("ERROR", message, '--', *print_args)

        # que l'on peut à présent appeler comme ceci
        error1(100, "un", "petit souci avec", [1, 2, 3])
```

Remarquons que maintenant la fonction `error1` ne peut plus être appelée sans argument, puisqu'on a mentionné un paramètre **obligatoire** `error_code`.

Ajout de fonctionnalités

Ou encore : la forme `argument=valeur_par_defaut` Nous envisageons à présent le cas - tout à fait indépendant de ce qui précède - où vous avez écrit une librairie graphique, dans laquelle vous exposez une fonction `ligne` définie comme suit. Évidemment pour garder le code simple, nous imprimons seulement les coordonnées du segment :

```
In [ ]: # première version de l'interface pour dessiner une ligne
        def ligne(x1, y1, x2, y2):
            "dessine la ligne (x1, y1) -> (x2, y2)"
            # restons simple
            print(f"la ligne ({x1}, {y1}) -> ({x2}, {y2})")
```

Vous publiez cette librairie en version 1, vous avez des utilisateurs ; et quelque temps plus tard vous écrivez une version 2 qui prend en compte la couleur. Ce qui vous conduit à ajouter un paramètre pour `ligne`.

Si vous le faites en déclarant

```
def ligne(x1, y1, x2, y2, couleur):
    ...
```

alors tous les utilisateurs de la version 1 vont **devoir changer leur code** - pour rester à fonctionnalité égale - en ajoutant un cinquième argument `'noir'` à leurs appels à `ligne`.

Vous pouvez éviter cet inconvénient en définissant la deuxième version de `ligne` comme ceci :

```
In [ ]: # deuxième version de l'interface pour dessiner une ligne
        def ligne(x1, y1, x2, y2, couleur="noir"):
            "dessine la ligne (x1, y1) -> (x2, y2) dans la couleur spécifiée"
            # restons simple
            print(f"la ligne ({x1}, {y1}) -> ({x2}, {y2}) en {couleur}")
```

Avec cette nouvelle définition, on peut aussi bien

```
In [ ]: # faire fonctionner du vieux code sans le modifier
        ligne(0, 0, 100, 100)
        # ou bien tirer profit du nouveau trait
        ligne(0, 100, 100, 0, 'rouge')
```

Les paramètres par défaut sont très utiles Notez bien que ce genre de situation peut tout aussi bien se produire sans que vous ne publiiez de librairie, à l'intérieur d'une seule application. Par exemple, vous pouvez être amené à ajouter un argument à une fonction parce qu'elle doit faire face à de nouvelles situations imprévues, et que vous n'avez pas le temps de modifier tout le code.

Ou encore plus simplement, vous pouvez choisir d'utiliser ce passage de paramètres dès le début de la conception ; une fonction `ligne` réaliste présentera une interface qui précise les points concernés, la couleur du trait, l'épaisseur du trait, le style du trait, le niveau de transparence, etc. Il n'est vraiment pas utile que tous les appels à `ligne` reprécisent tout ceci intégralement, aussi une bonne partie de ces paramètres seront très constructivement déclarés avec une valeur par défaut.

4.17.2 Complément - niveau avancé

Écrire un wrapper

Ou encore : la forme `**keywords` La notion de *wrapper* - emballage, en anglais - est très répandue en informatique, et consiste, à partir d'un morceau de code souche existant (fonction ou classe) à définir une variante qui se comporte comme la souche, mais avec quelques légères différences.

La fonction `error` était déjà un premier exemple de *wrapper*. Maintenant nous voulons définir un *wrapper* `ligne_rouge`, qui sous-traite à la fonction `ligne` mais toujours avec la couleur rouge.

Maintenant que l'on a injecté la notion de paramètre par défaut dans le système de signature des fonctions, se repose la question de savoir comment passer à l'identique les arguments de `ligne_rouge` à `ligne`.

Évidemment, une première option consiste à regarder la signature de `ligne` :

```
def ligne(x1, y1, x2, y2, couleur="noir")
```

Et à en déduire une implémentation de `ligne_rouge` comme ceci

```
In [ ]: # la version naïve - non conseillée - de ligne_rouge
        def ligne_rouge(x1, y1, x2, y2):
            return ligne(x1, y1, x2, y2, couleur='rouge')

        ligne_rouge(0, 0, 100, 100)
```

Toutefois, avec cette implémentation, si la signature de `ligne` venait à changer, on serait vraisemblablement amené à changer **aussi** celle de `ligne_rouge`, sauf à perdre en fonctionnalité. Imaginons en effet que `ligne` devienne dans une version suivante

```
In [ ]: # on ajoute encore une fonctionnalité à la fonction ligne
        def ligne(x1, y1, x2, y2, couleur="noir", epaisseur=2):
            print(f"la ligne ({x1}, {y1}) -> ({x2}, {y2})"
                  f" en {couleur} - ep. {epaisseur}")
```

Alors le wrapper ne nous permet plus de profiter de la nouvelle fonctionnalité. De manière générale, on cherche au maximum à se prémunir contre de telles dépendances. Aussi, il est de beaucoup préférable d'implémenter `ligne_rouge` comme suit, où vous remarquerez que **la seule hypothèse** faite sur `ligne` est qu'elle accepte un argument nommé `couleur`.

```
In [ ]: def ligne_rouge(*arguments, **keywords):
            # c'est le seul endroit où on fait une hypothèse sur la fonction `ligne`
            # qui est qu'elle accepte un argument nommé 'couleur'
            keywords['couleur'] = "rouge"
            return ligne(*arguments, **keywords)
```

Ce qui permet maintenant de faire

```
In [ ]: ligne_rouge(0, 100, 100, 0, epaisseur=4)
```

Pour en savoir plus - la forme générale

Une fois assimilé ce qui précède, vous avez de quoi comprendre une énorme majorité (99% au moins) du code python.

Dans le cas général, il est possible de combiner les 4 formes d'arguments : * arguments "normaux", dits positionnels * arguments nommés, comme `nom=<valeur>` * forme `*args` * forme `**dargs`

Vous pouvez vous reporter à cette page pour une description détaillée de ce cas général.

À l'appel d'une fonction, il faut résoudre les arguments, c'est-à-dire associer une valeur à chaque paramètre formel (ceux qui apparaissent dans le `def`) à partir des valeurs figurant dans l'appel.

L'idée est que pour faire cela, les arguments de l'appel ne sont pas pris dans l'ordre où ils apparaissent, mais les arguments positionnels sont utilisés en premier. La logique est que, naturellement les arguments positionnnels (ou ceux qui proviennent d'une `*expression`) viennent sans nom, et donc ne peuvent pas être utilisés pour résoudre des arguments nommés.

Voici un tout petit exemple pour vous donner une idée de la complexité de ce mécanisme lorsqu'on mélange toutes les 4 formes d'arguments à l'appel de la fonction (alors qu'on a défini la fonction avec 4 paramètres positionnels)

```
In [ ]: # une fonction qui prend 4 paramètres simples
        def foo(a, b, c, d):
            print(a, b, c, d)

In [ ]: # on peut l'appeler par exemple comme ceci
        foo(1, c=3, *(2,), **{'d':4})

In [ ]: # mais pas comme cela
        try:
            foo (1, b=3, *(2,), **{'d':4})
        except Exception as e:
            print(f"OOPS, {type(e)}, {e}")
```

Si le problème ne vous semble pas clair, vous pouvez regarder la documentation python décrivant ce problème.

4.18 Un piège courant

4.18.1 Complément - niveau basique

N'utilisez pas d'objet mutable pour les valeurs par défaut

En python il existe un piège dans lequel il est très facile de tomber. Aussi si vous voulez aller à l'essentiel : **n'utilisez pas d'objet mutable pour les valeurs par défaut** lors de la définition d'une fonction.

Si vous avez besoin d'écrire une fonction qui prend en argument par défaut une liste ou un dictionnaire vide, voici comment faire

```
In [ ]: # ne faites SURTOUT PAS ça
        def ne_faites_pas_ca(options={}):
            "faire quelque chose"

In [ ]: # mais plutôt comme ceci
        def mais_plutot_ceci(options=None):
```

```
        if options is None:
            options = {}
        "faire quelque chose"
```

4.18.2 Complément - niveau intermédiaire

Que se passe-t-il si on le fait ?

Considérons le cas relativement simple d'une fonction qui calcule une valeur - ici un entier aléatoire entre 0 et 10 -, et l'ajoute à une liste passée par l'appelant.

Et pour rendre la vie de l'appelant plus facile, on se dit qu'il peut être utile de faire en sorte que si l'appelant n'a pas de liste sous la main, on va créer pour lui une liste vide. Et pour ça on fait :

```
In [ ]: import random

        # l'intention ici est que si l'appelant ne fournit pas
        # la liste en entrée, on crée pour lui une liste vide
        def ajouter_un_aleatoire(resultats=[]):
            resultats.append(random.randint(0, 10))
            return resultats
```

Si on appelle cette fonction une première fois, tout semble bien aller

```
In [ ]: ajouter_un_aleatoire()
```

Sauf que, si on appelle la fonction une deuxième fois, on a une surprise !

```
In [ ]: ajouter_un_aleatoire()
```

Pourquoi ?

Le problème ici est qu'une valeur par défaut - ici l'expression `[]` - est évaluée **une fois** au moment de la **définition** de la fonction.

Toutes les fois où la fonction est appelée avec cet argument manquant, on va utiliser comme valeur par défaut **le même objet**, qui la première fois est bien une liste vide, mais qui se fait modifier par le premier appel.

Si bien que la deuxième fois on réutilise la même liste **qui n'est plus vide**. Pour aller plus loin, vous pouvez regarder la documentation python sur ce problème.

4.19 Arguments *keyword-only*

4.19.1 Complément - niveau intermédiaire

Rappel

Nous avons vu dans un précédent complément les 4 familles de paramètres qu'on peut déclarer dans une fonction :

1. paramètres positionnels (usuels)
2. paramètres nommés (forme *name=default*)
3. paramètres **args* qui attrape dans un tuple le relicat des arguments positionnels
4. paramètres ***kwds* qui attrape dans un dictionnaire le relicat des arguments nommés

Pour rappel :

```
In [ ]: # une fonction qui combine les différents
        # types de paramètres
        def foo(a, b=100, *args, **kwds):
            print(f"a={a}, b={b}, args={args}, kwds={kwds}")
```

```
In [ ]: foo(1)
```

```
In [ ]: foo(1, 2)
```

```
In [ ]: foo(1, 2, 3)
```

```
In [ ]: foo(1, 2, 3, bar=1000)
```

Un seul paramètre attrape-tout

Notez également que, de bon sens, on ne peut déclarer qu'un seul paramètre de chacune des formes d'attrape-tout ; on ne peut pas par exemple déclarer

```
# c'est illégal de faire ceci
def foo(*args1, *args2):
    pass
```

car évidemment on ne saurait pas décider de ce qui va dans `args1` et ce qui va dans `args2`.

Ordre des déclarations

L'ordre dans lequel sont déclarés les différents types de paramètres d'une fonction est imposé par le langage. Ce que vous avez peut-être en tête si vous avez appris **python2**, c'est qu'à l'époque on devait impérativement les déclarer dans cet ordre :

positionnnels, nommés, forme `*`, forme `**`

comme dans notre fonction `foo`.

Ça reste une bonne approximation, mais depuis python-3, les choses ont un petit peu changé suite à l'adoption du PEP 3102, qui vise à introduire la notion de paramètre qu'il faut impérativement nommer lors de l'appel (en anglais : *keyword-only* argument)

Pour résumer, il est maintenant possible de déclarer des **paramètres nommés après la forme `*`**

Voyons cela sur un exemple

```
In [ ]: # on peut déclarer un paramètre nommé **après** l'attrape-tout *args
        def bar(a, *args, b=100, **kwds):
                print(f"a={a}, b={b}, args={args}, kwds={kwds}")
```

L'effet de cette déclaration est que, si je veux passer un argument au paramètre `b`, **je dois le nommer**

```
In [ ]: # je peux toujours faire ceci
        bar(1)
```

```
In [ ]: # mais si je fais ceci l'argument 2 va aller dans args
        bar(1, 2)
```

```
In [ ]: # pour passer b=2, je **dois** nommer mon argument
        bar(1, b=2)
```

Ce trait n'est objectivement pas utilisé massivement en python, mais cela peut être utile de le savoir :

— en tant qu'utilisateur d'une librairie, car cela vous impose une certaine façon d'appeler une fonction ;
— en tant que concepteur d'une fonction, car cela vous permet de manifester qu'un paramètre optionnel joue un rôle particulier.

4.20 Passage d'arguments

4.20.1 Exercice - niveau basique

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

Vous devez écrire une fonction `distance` qui prend un nombre quelconque d'arguments numériques non complexes, et qui retourne la racine carrée de la somme des carrés des arguments.

Plus précisément : *distance* $(x_1, ..., x_n) = \sqrt{\sum x_i^2}$

Par convention on fixe que $ distance() = 0 $

```
In [ ]: # des exemples
        exo_distance.example()
```

```
In [ ]: # ATTENTION vous devez aussi définir les arguments de la fonction
        def distance(votre, signature):
            return "votre code"
```

```
In [ ]: # la correction
        exo_distance.correction(distance)
```

4.20.2 Exercice - niveau intermédiaire

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

On vous demande d'écrire une fonction `numbers` * qui prend en argument un nombre quelconque d'entiers, * et qui retourne un tuple contenant * la somme * le minimum * le maximum de ses arguments.

Si aucun argument n'est passé, `numbers` doit renvoyer un tuple contenant 3 entiers `0`.

```
In [ ]: # par exemple
        exo_numbers.example()
```

En guise d'indice, je vous invite à regarder les fonctions *builtin* `sum`, `min` et `max`.

```
In [ ]: # vous devez définir votre propre signature
        def numbers(votre, signature):
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_numbers.correction(numbers)
```

SEMAINE 5

ITÉRATION, IMPORTATION ET ESPACE DE NOMMAGE

5.1 Les instructions `break` et `continue`

5.1.1 Complément - niveau basique

`break` **et** `continue`

En guise de rappel de ces deux notions que nous avons déjà rencontrées dans la séquence consacrée aux boucle `while` la semaine passée, python propose deux instructions très pratiques permettant de contrôler l'exécution à l'intérieur des boucles de répétition, et ceci s'applique indifféremment aux boucles `for` ou `while` :

— `continue` : pour abandonner l'itération courante, et passer à la suivante, en **restant dans la boucle** ;
— `break` : pour abandonner **complètement** la boucle.

Voici un exemple simple d'utilisation de ces deux instructions :

```
In [ ]: for entier in range(1000):
            # on ignore les nombres non multiples de 10
            if entier % 10 != 0:
                continue
            print(f"on traite l'entier {entier}")
            # on s'arrête à 50
            if entier >= 50:
                break
        print("on est sorti de la boucle")
```

Pour aller plus loin, vous pouvez lire cette documentation.

5.2 Une limite de la boucle `for`

5.2.1 Complément - niveau basique

Pour ceux qui veulent suivre le cours au niveau basique, retenez seulement que dans une boucle `for` sur un objet mutable, **il ne faut pas modifier le sujet** de la boucle.

Ainsi par exemple il ne **faut pas faire** quelque chose comme ceci :

```
In [ ]: # on veut enlever de l'ensemble toutes les chaines
        # qui ne contiennent pas par 'bert'
        ensemble = {'marc', 'albert'}

        # ceci semble une bonne idée mais ne fonctionne pas
```

```
        for valeur in ensemble:
            if 'bert' not in valeur:
                ensemble.discard(valeur)
```

Comment faire alors ?

Première remarque, votre premier réflexe pourrait être de penser à une compréhension d'ensemble :

```
In [ ]: ensemble2 = {valeur for valeur in ensemble if 'bert' in valeur}
        ensemble2
```

C'est sans doute la meilleure solution. Par contre, évidemment, on n'a pas modifié l'objet ensemble initial, on a créé un nouvel objet. En supposant que l'on veuille modifier l'objet initial, il nous faut faire la boucle sur une *shallow copy* de cet objet. Notez qu'ici, il ne s'agit d'économiser de la mémoire, puisque l'on fait une *shallow copy*.

```
In [ ]: from copy import copy
        # on veut enlever de l'ensemble toutes les chaines
        # qui ne contiennent pas 'bert'
        ensemble = {'marc', 'albert'}

        # si on fait d'abord une copie tout va bien
        for valeur in copy(ensemble):
            if 'bert' not in valeur:
                ensemble.discard(valeur)

        print(ensemble)
```

Avertissement

Dans l'exemple ci-dessus, on voit que l'interpréteur se rend compte que l'on est en train de modifier l'objet de la boucle, et nous le signifie.

Ne vous fiez pas forcément à cet exemple, il existe des cas -- nous en verrons plus loin dans ce document -- où l'interpréteur peut accepter votre code alors qu'il n'obéit pas à cette règle, et du coup essentiellement se mettre à faire n'importe quoi.

Précisons bien la limite

Pour être tout à fait clair, lorsqu'on dit qu'il ne faut pas modifier l'objet de la boucle `for`, il ne s'agit que du premier niveau.

On ne doit pas modifier la **composition de l'objet en tant qu'itérable**, mais on peut sans souci modifier chacun des objets qui constitue l'itération.

Ainsi cette construction par contre est tout à fait valide :

```
In [ ]: liste = [[1], [2], [3]]
        print('avant', liste)

In [ ]: for sous_liste in liste:
            sous_liste.append(100)
        print('après', liste)
```

Dans cet exemple, les modifications ont lieu sur les éléments de `liste`, et non sur l'objet `liste` lui-même, c'est donc tout à fait légal.

5.2.2 Complément - niveau intermédiaire

Pour bien comprendre la nature de cette limitation, il faut bien voir que cela soulève deux types de problèmes distincts.

Difficulté d'ordre sémantique

D'un point de vue sémantique, si l'on voulait autoriser ce genre de choses, il faudrait définir très précisément le comportement attendu.

Considérons par exemple la situation d'une liste qui a 10 éléments, sur laquelle on ferait une boucle et que, par exemple au 5ème élément, on enlève le 8ème élément. Quel serait le comportement attendu dans ce cas ? Faut-il ou non que la boucle envisage alors le 8-ème élément ?

La situation serait encore pire pour les dictionnaires et ensembles pour lesquels l'ordre de parcours n'est pas spécifié ; ainsi on pourrait écrire du code totalement indéterministe si le parcours d'un ensemble essayait : * d'enlever l'élément b lorsqu'on parcourt l'élément a ; * d'enlever l'élément a lorsqu'on parcourt l'élément b.

On le voit, il n'est déjà pas très simple d'expliciter sans ambigüité le comportement attendu d'une boucle `for` qui serait autorisée à modifier son propre sujet.

Difficulté d'implémentation

Voyons maintenant un exemple de code qui ne respecte pas la règle, et qui modifie le sujet de la boucle en lui ajoutant des valeurs

```
# cette boucle ne termine pas
liste = [1, 2, 3]
for c in liste:
    if c == 3:
        liste.append(c)
```

Nous avons volontairement mis ce code **dans une cellule de texte** et non de code : vous **ne pouvez pas l'exécuter** dans le notebook. Si vous essayez de l'exécuter sur votre ordinateur vous constaterez qu'elle ne termine pas, en fait à chaque itération on ajoute un nouvel élément dans la liste, et du coup la boucle a un élément de plus à balayer ; ce programme ne termine jamais.

5.3 Itérateurs

5.3.1 Complément - niveau intermédaire

Dans ce complément nous allons dire quelques mots du module `itertools` qui fournit sous forme d'itérateurs des utilitaires communs qui peuvent être très utiles. On vous rappelle que l'intérêt premier des itérateurs est de parcourir des données sans créer de structure de données temporaire, donc à coût mémoire faible et constant.

Le module `itertools`

À ce stade, j'espère que vous savez trouver la documentation du module que je vous invite à avoir sous la main.

```
In [ ]: import itertools
```

Comme vous le voyez dans la doc, les fonctionnalités de `itertools` tombent dans 3 catégories : * des itérateurs infinis, comme par exemple `cycle` ; * des itérateurs pour énumérer les combinatoires usuelles en mathématiques, comme les permutations, les combinaisons, le produit cartésien, etc. ; * et enfin des itérateurs correspondants à des traits que nous avons déjà rencontrés, mais implémentés sous forme d'itérateurs.

À nouveau, toutes ces fonctionnalités sont offertes **sous la forme d'itérateurs**.

Pour détailler un tout petit peu cette dernière famille, signalons :

— `chain` qui permet de **concaténer** plusieurs itérables sous la forme d'un **itérateur** :

```
In [ ]: for x in itertools.chain((1, 2), [3, 4]):
            print(x)
```

— `islice` qui fournit un itérateur sur un slice d'un itérable. On peut le voir comme une généralisation de `range` qui parcours n'importe quel itérable.

```
In [ ]: import string
        support = string.ascii_lowercase
        print(f'support={support}')

In [ ]: # range
        for x in range(3, 8):
            print(x)

In [ ]: # islice
        for x in itertools.islice(support, 3, 8):
            print(x)
```

5.4 Programmation fonctionnelle

5.4.1 Complément - niveau basique

Pour résumer

La notion de programmation fonctionnelle consiste essentiellement à pouvoir manipuler les fonctions comme des objets à part entière, et à les passer en argument à d'autres fonctions, comme cela est illustré dans la vidéo.

On peut créer une fonction par l'intermédiaire de : * l'*expression* `lambda:`, on obtient alors une fonction *anonyme* ; * l'*instruction* `def` et dans ce cas on peut accéder à l'objet fonction par son nom.

Pour des raisons de syntaxe surtout, on a davantage de puissance avec `def`.

On peut calculer la liste des résultats d'une fonction sur une liste (plus généralement un itérable) d'entrées par : * `map`, éventuellement combiné à `filter` ; * une compréhension de liste, éventuellement assortie d'un `if`.

Nous allons revoir les compréhensions dans la prochaine vidéo.

5.4.2 Complément - niveau intermédiaire

Pour les curieux qui ont entendu le terme de *map - reduce* , voici la logique derrière l'opération `reduce`, qui est également disponible en python au travers du module `functools`.

`reduce`

La fonction `reduce` permet d'appliquer une opération associative à une liste d'entrées. Pour faire simple, étant donné un opérateur binaire $\otimes$ on veut pouvoir calculer

$x_1 \otimes x_2 ... \otimes x_n$

De manière un peu moins abstraite, on suppose qu'on dispose d'une **fonction binaire** `f` qui implémente l'opérateur $\otimes$, et alors

`reduce` $(f, [x_1, ..x_n]) = f(...f(f(x_1, x_2), x_3), .., x_n)$

En fait `reduce` accepte un troisième argument - qu'il faut comprendre comme l'élément neutre de l'opérateur/fonction en question - et qui est retourné lorsque la liste en entrée est vide.

Par exemple voici - encore - une autre implémentation possible de la fonction `factoriel`.

On utilise ici le module `operator`, qui fournit sous forme de fonctions la plupart des opérateurs du langage, et notamment, dans notre cas, `operator.mul` ; cette fonction retourne tout simplement le produit de ses deux arguments.

```
In [ ]: # la fonction reduce dans python3 n'est plus une builtin comme en python2
        # elle fait partie du module functools
        from functools import reduce

        # la multiplication, mais sous forme de fonction et non d'opérateur
        from operator import mul

        def factoriel(n):
            return reduce(mul, range(1, n+1), 1)

        # ceci fonctionne aussi pour factoriel (0)
        for i in range(5):
            print(f"{i} -> {factoriel(i)}")
```

Cas fréquents de `reduce` Par commodité, python fournit des fonctions built-in qui correspondent en fait à des `reduce` fréquents, comme la somme, et les opérations `min` et `max` :

```
In [ ]: entrees = [8, 5, 12, 4, 45, 7]

        print('sum', sum(entrees))
        print('min', min(entrees))
        print('max', max(entrees))
```

5.5 Tri de listes

5.5.1 Complément - niveau intermédiaire

Nous avons vu durant une semaine précédente comment faire le tri simple d'une liste, en utilisant éventuellement le paramètre `reverse` de la méthode `sort` sur les listes. Maintenant que nous sommes familiers avec la notion de fonction, nous pouvons approfondir ce sujet.

Cas général

Dans le cas général, on est souvent amené à trier des objets selon un critère propre à l'application. Imaginons par exemple que l'on dispose d'une liste de tuples à deux éléments, dont le premier est la latitude et le second la longitude :

```
In [ ]: coordonnees = [(43, 7), (46, -7), (46, 0)]
```

Il est possible d'utiliser la méthode `sort` pour faire cela, mais il va falloir l'aider un peu plus, et lui expliquer comment comparer deux éléments de la liste.

Voyons comment on pourrait procéder pour trier par longitude :

```
In [ ]: def longitude(element):
            return element[1]

        coordonnees.sort(key=longitude)
        print("coordonnées triées par longitude", coordonnees)
```

Comme on le devine, le procédé ici consiste à indiquer à `sort` comment calculer, à partir de chaque élément, une valeur numérique qui sert de base au tri.

Pour cela on passe à la méthode `sort` un argument `key` qui désigne **une fonction**, qui lorsqu'elle est appliquée à un élément de la liste, retourne la valeur qui doit servir de base au tri : dans notre exemple, la fonction `longitude`, qui renvoie le second élément du tuple.

On aurait pu utiliser de manière équivalente une fonction lambda ou la methode `itemgetter` to module `operator`

```
In [ ]: # fonction lambda
        coordonnees = [(43, 7), (46, -7), (46, 0)]
        coordonnees.sort(key=lambda x: x[1])
        print("coordonnées triées par longitude", coordonnees)

        # méthode operator.getitem
        import operator
        coordonnees = [(43, 7), (46, -7), (46, 0)]
        coordonnees.sort(key=operator.itemgetter(1))
        print("coordonnées triées par longitude", coordonnees)
```

Fonction de commodité : `sorted`

On a vu que `sort` réalise le tri de la liste "en place". Pour les cas où une copie est nécessaire, python fournit également une fonction de commodité, qui permet précisément de renvoyer la **copie** triée d'une liste d'entrée. Cette fonction est baptisée `sorted`, elle s'utilise par exemple comme ceci, sachant que les arguments `reverse` et `key` peuvent être mentionnés comme avec `sort` :

```
In [ ]: liste = [8, 7, 4, 3, 2, 9, 1, 5, 6]
        # on peut passer à sorted les mêmes arguments que pour sort
        triee = sorted(liste, reverse=True)
        # nous avons maintenant deux objets distincts
        print('la liste triée est une copie ', triee)
        print('la liste initiale est intacte', liste)
```

Nous avons qualifié `sorted` de fonction de commodité car il est très facile de s'en passer ; en effet on aurait pu écrire à la place du fragment précédent :

```
In [ ]: liste = [8, 7, 4, 3, 2, 9, 1, 5, 6]
        # ce qu'on a fait dans la cellule précédente est équivalent à
        triee = liste[:]
        triee.sort(reverse=True)
```

```
#
print('la liste triée est une copie ', triee)
print('la liste initiale est intacte', liste)
```

Alors que `sort` est une fonction sur les listes, `sorted` peut trier n'importe quel itérable et retourne le résultat dans une liste. Cependant, au final, le coût mémoire est le même. Pour utiliser `sort` on va créer une liste des éléments de l'itérable, puis on fait un tri en place avec `sort`. Avec `sorted` on applique directement le tri sur l'itérable, mais on crée une liste pour stocker le résultat. Dans les deux cas, on a une liste à la fin et aucune structure de données temporaire créée.

Pour en savoir plus

Pour avoir plus d'informations sur `sort` et `sorted` vous pouvez lire cette section de la documentation python sur le tri.

5.5.2 Exercice - niveau basique

Tri de plusieurs listes

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

Écrivez une fonction qui : * accepte en argument une liste de listes, * et qui retourne **la même liste**, mais avec toutes les sous-listes **triées en place**.

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

Écrivez votre code ici :

```
In [ ]: def multi_tri(listes):
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_multi_tri.correction(multi_tri)
```

5.5.3 Exercice - niveau intermédiaire

Tri de plusieurs listes, dans des directions différentes

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

Modifiez votre code pour qu'il accepte cette fois **deux** arguments listes que l'on suppose de tailles égales.

Comme tout à l'heure le premier argument est une liste de listes à trier.

À présent le second argument est une liste (ou un tuple) de booléens, de même cardinal que le premier argument, et qui indiquent l'ordre dans lequel on veut trier la liste d'entrée de même rang. `True` signifie un tri descendant, `False` un tri ascendant.

Comme dans l'exercice `multi_tri`, il s'agit de modifier en place les données en entrée, et de retourner la liste de départ.

```
In [ ]: # Pour être un peu plus clair, voici à quoi on s'attend
        exo_multi_tri_reverse.example()
```

À vous de jouer :

```
In [ ]: def multi_tri_reverse(listes, reverses):
            "<votre_code>"

In [ ]: # et pour vérifier votre code
        exo_multi_tri_reverse.correction(multi_tri_reverse)
```

5.5.4 Exercice - niveau intermédaire

Les deux exercices de ce notebook font référence également à des notions vues en fin de semaine 4, sur le passage d'arguments aux fonctions.

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

On vous demande d'écrire une fonction qui prend en argument : * une fonction `f`, dont vous savez seulement que le premier argument est numérique, et qu'elle ne prend **que des arguments positionnels** (sans valeur par défaut) ; * un nombre quelconque - mais au moins 1 - d'arguments positionnels `args`, dont on sait qu'ils pourraient être passés à `f`.

Et on attend en retour le résultat de `f` appliqués à tous ces arguments, mais avec le premier d'entre eux multiplé par deux.

Formellement : doubler_premier(f, x_1, x_2,..., x_n) = f($2 * x_1$, x_2,..., x_n)

```
In [ ]: # quelques exemples de ce qui est attendu.
        # add et mul sont les opérateurs binaires du module operator,
        # soit l'addition et la multiplication respectivement.
        # distance est la fonction d'un exercice précédent
        exo_doubler_premier.example()

In [ ]: # ATTENTION vous devez aussi définir les arguments de la fonction
        def doubler_premier(votre, signature):
            return "votre code"

In [ ]: exo_doubler_premier.correction(doubler_premier)
```

5.5.5 Exercice - niveau intermédaire

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

Vous devez maintenant écrire une deuxième version qui peut fonctionner avec une fonction quelconque (elle peut avoir des arguments nommés avec valeurs par défaut).

La fonction `doubler_premier_kwds` que l'on vous demande d'écrire maintenant prend donc un premier argument `f` qui est une fonction, un second argument positionnel qui est le premier argument de `f` (et donc qu'il faut doubler), et le reste des arguments de f, qui donc, à nouveau, peuvent être nommés ou non.

```
In [ ]: # quelques exemples de ce qui est attendu
        # avec ces deux fonctions

        def add3(x, y=0, z=0):
            return x + y + z
```

```
def mul3(x=1, y=1, z=1):
    return x * y * z

exo_doubler_premier_kwds.example()
```

Vous remarquerez que l'on n'a pas mentionné dans cette liste d'exemples

```
doubler_premier_kwds (muln, x=1, y=1)
```

que l'on ne demande pas de supporter puisqu'il est bien précisé que doubler_premier a deux arguments positionnels.

```
In [ ]: # ATTENTION vous devez aussi définir les arguments de la fonction
        def doubler_premier_kwds(votre, signature):
            "<votre code>"
```

```
In [ ]: exo_doubler_premier_kwds.correction(doubler_premier_kwds)
```

5.6 Comparaison de fonctions

5.6.1 Exercice - niveau avancé

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

À présent nous allons écrire une version très simplifiée de l'outil qui est utilisé dans ce cours pour corriger les exercices. Vous aurez sans doute remarqué que les fonctions de correction prennent en argument la fonction à corriger.

Par exemple un peu plus bas, la cellule de correction fait

```
exo_compare_all.correction(compare_all)
```

dans lequel `compare_all` est l'objet fonction que vous écrivez en réponse à cet exercice.

On vous demande d'écrire une fonction `compare` qui prend en argument : * deux fonctions `f` et `g` ; imaginez que l'une d'entre elles fonctionne et qu'on cherche à valider l'autre ; dans cette version simplifiée toutes les fonctions acceptent exactement un argument ; * une liste d'entrées `entrees` ; vous pouvez supposer que chacune de ces entrées est dans le domaine de `f` et de `g` (dit autrement, on peut appeler `f` et `g` sur chacune des entrées sans craindre qu'une exception soit levée).

Le résultat attendu pour le retour de `compare` est une liste qui contient autant de booléens que d'éléments dans `entrees`, chacun indiquant si avec l'entrée correspondante on a pu vérifier que `f(entree) == g(entree)`.

Dans cette première version de l'exercice vous pouvez enfin supposer que les entrées ne sont pas modifiées par `f` ou `g`.

Pour information dans cet exercice : * `factorial` correspond à `math.factorial` * `fact` et `broken_fact` sont des fonctions implémentées par nos soins, la première est correcte alors que la seconde retourne 0 au lieu de 1 pour l'entrée 0.

```
In [ ]: # par exemple
        exo_compare_all.example()
```

Ce qui, dit autrement, veut tout simplement dire que `fact` et `factorial` coïncident sur les entrées 0, 1 et 5, alors que `broken_fact` et `factorial` ne renvoient pas la même valeur avec l'entrée `0`.

```
In [ ]: # c'est à vous
        def compare_all(f, g, entrees):
            "<votre code>"
```

```
In [ ]: # pour vérifier votre code
        exo_compare_all.correction(compare_all)
```

5.6.2 Exercice optionnel - niveau avancé

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

`compare` revisitée

Nous reprenons ici la même idée que `compare`, mais en levant l'hypothèse que les deux fonctions attendent un seul argument. Il faut écrire une nouvelle fonction `compare_args` qui prend en entrée : * deux fonctions `f` et `g` comme ci-dessus ; * mais cette fois une liste (ou un tuple) `argument_tuples` de **tuples** d'arguments d'entrée.

Comme ci-dessus on attend en retour une liste `retour` de booléens, de même taille que `argument_tuples`, telle que, si `len(argument_tuples)` vaut n :

$\forall i \in \{1, ..., n\}$, si `argument_tuples[i]` == [a_1,...,a_j], alors

`retour(i) == True` $\iff$ f (a_1,...,a_j) == g (a_1,...,a_j)

Pour information, dans tout cet exercice : * `factorial` correspond à `math.factorial` ; * `fact` et `broken_fact` sont des fonctions implémentées par nos soins, la première est correcte alors que la seconde retourne 0 au lieu de 1 pour l'entrée 0 ; * `add` correspond à l'addition binaire `operator.add` ; * `plus` et `broken_plus` sont des additions binaires que nous avons écrites, l'une étant correcte et l'autre étant fausse lorsque le premier argument est nul.

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

```
In [ ]: # ATTENTION vous devez aussi définir les arguments de la fonction
        def compare_args(votre, signature):
            "<votre_code>"
```

```
In [ ]: exo_compare_args.correction(compare_args)
```

5.7 Construction de liste par compréhension

5.7.1 Révision - niveau basique

Ce mécanisme très pratique permet de construire simplement une liste à partir d'une autre (ou de **tout autre type itérable** en réalité, mais nous y viendrons).

Pour l'introduire en deux mots, disons que la compréhension de liste est à l'instruction `for` ce que l'expression conditionnelle est à l'instruction `if`, c'est-à-dire qu'il s'agit d'une **expression à part entière**.

Cas le plus simple

Voyons tout de suite un exemple :

```
In [ ]: depart = (-5, -3, 0, 3, 5, 10)
        arrivee = [x**2 for x in depart]
        arrivee
```

Le résultat de cette expression est donc une liste, dont les éléments sont les résultats de l'expression `x**2` pour `x` prenant toutes les valeurs de `depart`.

Remarque : si on prend un point de vue un peu plus mathématique, ceci revient donc à appliquer une certaine fonction (ici $x \to x^2$) à une collection de valeurs, et à retourner la liste des résultats. Dans les langages fonctionnels, cette opération est connue sous le nom de `map`, comme on l'a vu dans la séquence précédente.

Digression

```
In [ ]: # profitons de cette occasion pour voir
        # comment tracer une courbe avec matplotlib
        %matplotlib inline
        import matplotlib.pyplot as plt
        plt.ion()
```

```
In [ ]: # si on met le depart et l'arrivee
        # en abscisse et en ordonnee, on trace
        # une version tronquée de la courbe de f: x -> x**2
        plt.plot(depart, arrivee);
```

Restriction à certains éléments

Il est possible également de ne prendre en compte que certains des éléments de la liste de départ, comme ceci :

```
In [ ]: [x**2 for x in depart if x%2 == 0]
```

qui cette fois ne contient que les carrés des éléments pairs de `depart`.

Remarque : pour prolonger la remarque précédente, cette opération s'appelle fréquemment `filter` dans les langages de programmation.

Autres types

On peut fabriquer une compréhension à partir de tout objet itérable, pas forcément une liste, mais le résultat est toujours une liste, comme on le voit sur ces quelques exemples :

```
In [ ]: [ord(x) for x in 'abc']
```

```
In [ ]: [chr(x) for x in (97, 98, 99)]
```

Autres types (2)

On peut également construire par compréhension des dictionnaires et des ensembles :

```
In [ ]: d = {x: ord(x) for x in 'abc'}
        d
```

```
In [ ]: e = {x**2 for x in (97, 98, 99) if x %2 == 0}
        e
```

Pour en savoir plus

Voyez la section sur les compréhensions de liste dans la documentation python.

5.8 Compréhensions imbriquées

5.8.1 Compléments - niveau intermédiaire

Imbrications

On peut également imbriquer plusieurs niveaux pour ne construire qu'une seule liste, comme par exemple :

```
In [ ]: [n + p for n in [2, 4] for p in [10, 20, 30]]
```

Bien sûr on peut aussi restreindre ces compréhensions, comme par exemple :

```
In [ ]: [n + p for n in [2, 4] for p in [10, 20, 30] if n*p >= 40]
```

Observez surtout que le résultat ci-dessus est une liste simple (de profondeur 1), à comparer avec :

```
In [ ]: [[n + p for n in [2, 4]] for p in [10, 20, 30]]
```

qui est de profondeur 2, et où les résultats atomiques apparaissent dans un ordre différent.

Un moyen mnémotechnique pour se souvenir dans quel ordre les compréhensions imbriquées produisent leur résultat, est de penser à la version "naïve" du code qui produirait le même résultat ; dans ce code les clause `for` et `if` apparaissent **dans le même ordre** que dans la compréhension :

```
In [ ]: # notre exemple :
        # [n + p for n in [2, 4] for p in [10, 20, 30] if n*p >= 40]

        # est équivalent à ceci :
        resultat = []
        for n in [2, 4]:
            for p in [10, 20, 30]:
                if n*p >= 40:
                    resultat.append(n + p)
        resultat
```

Ordre d'évaluation de `[[ .. for .. ] .. for .. ]`

Pour rappel, on peut imbriquer des compréhensions de compréhensions. Commençons par poser

```
In [ ]: n = 4
```

On peut alors créer une liste de listes comme ceci :

```
In [ ]: [[(i, j) for i in range(1, j + 1)] for j in range(1, n + 1)]
```

Et dans ce cas, très logiquement, l'évaluation se fait **en commençant par la fin**, ou si on préfère **"par l'extérieur"**, c'est-à-dire que le code ci-dessus est équivalent à :

```
In [ ]: # en version bavarde, pour illustrer l'ordre des "for"
        resultat_exterieur = []
        for j in range(1, n + 1):
            resultat_interieur = []
            for i in range(1, j + 1):
                resultat_interieur.append((i, j))
            resultat_exterieur.append(resultat_interieur)
        resultat_exterieur
```

Avec `if`

Lorsqu'on assortit les compréhensions imbriquées de cette manière de clauses `if`, l'ordre d'évaluation est tout aussi logique. Par exemple, si on voulait se limiter - arbitrairement - aux lignes correspondant à `j` pair, et aux diagonales où `i+j` est pair, on écrirait :

```
In [ ]: [[(i, j) for i in range(1, j + 1) if (i + j)%2 == 0]
                 for j in range(1, n + 1) if j % 2 == 0]
```

ce qui est équivalent à :

```
In [ ]: # en version bavarde à nouveau
        resultat_exterieur = []
        for j in range(1, n + 1):
            if j % 2 == 0:
                resultat_interieur = []
                for i in range(1, j + 1):
                    if (i + j) % 2 == 0:
                        resultat_interieur.append((i, j))
                resultat_exterieur.append(resultat_interieur)
        resultat_exterieur
```

Le point important ici est que l'**ordre** dans lequel il faut lire le code est **naturel**, et dicté par l'imbrication des `[ .. ]`.

5.8.2 Compléments - niveau avancé

Les variables de boucle *fuitent*

Nous avons déjà signalé que les variables de boucle **restent définies** après la sortie de la boucle, ainsi nous pouvons examiner :

```
In [ ]: i, j
```

C'est pourquoi, afin de comparer les deux formes de compréhension imbriquées nous allons explicitement retirer les variables `i` et `j` de l'environnement

```
In [ ]: del i, j
```

Ordre d'évaluation de `[ .. for .. for .. ]`

Toujours pour rappel, on peut également construire une compréhension imbriquée mais **à un seul niveau**. Dans une forme simple cela donne :

```
In [ ]: [(x, y) for x in [1, 2] for y in [1, 2]]
```

Avertissement méfiez-vous toutefois, car il est facile de ne pas voir du premier coup d'oeil qu'ici on évalue les deux clauses `for` **dans un ordre différent**.

Pour mieux le voir, essayons de reprendre la logique de notre tout premier exemple, mais avec une forme de double compréhension *à plat* :

```
In [ ]: [ (i, j) for i in range(1, j + 1) for j in range(1, n + 1) ]
```

On obtient une erreur, l'interpréteur se plaint à propos de la variable `j` (c'est pourquoi nous l'avons effacée de l'environnement au préalable).

Ce qui se passe ici, c'est que, comme nous l'avons déjà mentionné en semaine 3, le code que nous avons écrit est en fait équivalent à :

```
In [ ]: # la version bavarde de cette imbrication à plat, à nouveau :
        # [ (i, j) for i in range(1, j + 1) for j in range(1, n + 1) ]
        # serait
        resultat = []
        for i in range(1, j + 1):
            for j in range(1, n + 1):
                resultat.append((i, j))
```

Et dans cette version * dépliée* on voit bien qu'en effet on utilise `j` avant qu'elle ne soit définie.

Conclusion

La possibilité d'imbriquer des compréhensions avec plusieurs niveaux de `for` dans la même compréhension est un trait qui peut rendre service, car c'est une manière de simplifier la structure des entrées (on passe essentiellement d'une liste de profondeur 2 à une liste de profondeur 1).

Mais il faut savoir ne pas en abuser, et rester conscient de la confusion qui peut en résulter, et en particulier être prudent et prendre le temps de bien se relire. N'oublions pas non plus ces deux phrases du zen de python : "*Flat is better than nested*" et surtout "*Readability counts*".

5.9 Compréhensions

5.9.1 Exercice - niveau basique

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

Il vous est demandé d'écrire une fonction `aplatir` qui prend *un unique* argument `l_conteneurs` qui est une liste (ou plus généralement un itérable) de conteneurs (ou plus généralement d'itérables), et qui retourne la liste de tous les éléments de tous les conteneurs.

```
In [ ]: # par exemple
        exo_aplatir.example()
```

```
In [ ]: def aplatir(conteneurs):
            "<votre_code>"
```

```
In [ ]: # vérifier votre code
        exo_aplatir.correction(aplatir)
```

5.9.2 Exercice - niveau intermédiaire

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

À présent, on passe en argument deux conteneurs (deux itérables) `c1` et `c2` de même taille à la fonction `alternat`, qui doit construire une liste contenant les éléments pris alternativement dans `c1` et dans `c2`.

```
In [ ]: # exemple
        exo_alternat.example()
```

Indice pour cet exercice il peut être pertinent de recourir à la fonction *builtin* `zip`.

```
In [ ]: def alternat(c1, c2):
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_alternat.correction(alternat)
```

5.9.3 Exercice - niveau intermédiaire

On se donne deux ensembles A et B de tuples de la forme

```
(entier, valeur)
```

On vous demande d'écrire une fonction `intersect` qui retourne l'ensemble des objets `valeur` associés (dans A ou dans B) à un entier qui soit présent dans (un tuple de) A *et* dans (un tuple de) B.

```
In [ ]: # un exemple
        from corrections.exo_intersect import exo_intersect
        exo_intersect.example()
```

```
In [ ]: def intersect(A, B):
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_intersect.correction(intersect)
```

5.10 Expressions génératrices

5.10.1 Complément - niveau basique

Comment transformer une compréhension de liste en itérateur ?

Nous venons de voir les fonctions génératrices qui sont un puissant outil pour créer facilement des itérateurs. Nous verrons prochainement comment utiliser ces fonctions génératrices pour tranformer en quelques lignes de code vos propres objets en itérateurs.

Vous savez maintenant qu'en python on favorise la notion d'itérateurs puisqu'ils se manipulent comme des objets itérables et qu'ils sont en général beaucoup plus compacts en mémoire que l'itérable correspondant.

Comme les compréhensions de listes sont fréquemment utilisées en python, mais qu'elles sont des itérables potentiellement gourmands en ressources mémoire, on souhaiterait pouvoir créer un itérateur directement à partir d'une compréhension de liste. C'est possible et très facile en python. Il suffit de remplacer les crochets par des parenthèses, regardons cela.

```
In [ ]: # c'est une compréhension de liste
        comprehension = [x**2 for x in range(100) if x%17 == 0]
        print(comprehension)

In [ ]: # c'est une expression génératrice
        generator = (x**2 for x in range(100) if x%17 == 0)
        print(generator)
```

Ensuite pour utiliser une expression génératrice, c'est très simple, on l'utilise comme n'importe quel itérateur.

```
In [ ]: generator is iter(generator) # generator est bien un itérateur

In [ ]: # affiche les premiers carrés des multiples de 17
        for count, carre in enumerate(generator, 1):
            print(f'Contenu de generator après {count} itérations : {carre}')
```

Avec une expression génératrice on n'est plus limité comme avec les compréhensions par le nombre d'éléments :

```
In [ ]: # trop grand pour une compréhension,
        # mais on peut créer le générateur sans souci
        generator = (x**2 for x in range(10**18) if x%17==0)

        # on va calculer tous les carrés de multiples de 17
        # plus petits que 10**10 et dont les 4 derniers chiffres sont 1316
        recherche = set()

        # le point important, c'est qu'on n'a pas besoin de
        # créer une liste de 10**18 éléments
        # qui serait beaucoup trop grosse pour la mettre dans la mémoire vive

        # avec un générateur, on ne paie que ce qu'on utilise...
        for x in generator:
            if x > 10**10:
                break
            elif str(x)[-4:] == '1316':
                recherche.add(x)
        print(recherche)
```

5.10.2 Complément - niveau intermédiaire

Compréhension *vs* expression génératrice

Digression : liste *vs* itérateur En python3, nous avons déjà rencontré la fonction `range` qui retourne les premiers entiers.

Ou plutôt, c'est **comme si** elle retournait les premiers entiers lorsqu'on fait une boucle `for`

```
In [ ]: # on peut parcourir un range comme si c'était une liste
        for i in range(4):
            print(i)
```

mais en réalité le résultat de `range` exhibe un comportement un peu étrange, en ce sens que :

```
In [ ]: # mais en fait la fonction range ne renvoie PAS une liste (depuis python3)
        range(4)

In [ ]: # et en effet ce n'est pas une liste
        isinstance(range(4), list)
```

La raison de fond pour ceci, c'est que **le fait de construire une liste** est une opération relativement coûteuse - toutes proportions gardées - car il est nécessaire d'allouer de la mémoire pour **stocker tous les éléments** de la liste à un instant donné ; alors qu'en fait dans l'immense majorité des cas, on n'a **pas réellement besoin** de cette place mémoire, tout ce dont on a besoin c'est d'itérer sur un certain nombre de valeurs mais **qui peuvent être calculées** au fur et à mesure que l'on parcourt la liste.

Compréhension et expression génératrice À la lumière de ce qui vient d'être dit, on peut voir qu'une compréhension n'est **pas toujours le bon choix**, car par définition elle **construit une liste** de résultats - de la fonction appliquée successivment aux entrées.

Or dans les cas où, comme pour `range`, on n'a pas réellement besoin de cette liste **en temps que telle** mais seulement de cet artefact pour pouvoir itérer sur la liste des résultats, il est préférable d'utiliser une **expression génératrice**.

Voyons tout de suite sur un exemple à quoi cela ressemblerait.

```
In [ ]: depart = (-5, -3, 0, 3, 5, 10)
        # dans le premier calcul de arrivee
        # pour rappel, la compréhension est entre []
        # arrivee = [x**2 for x in depart]

        # on peut écrire presque la même chose avec des () à la place
        arrivee2 = (x**2 for x in depart)
        arrivee2
```

Comme pour `range`, le résultat de l'expression génératrice ne se laisse pas regarder avec `print`, mais comme pour `range`, on peut itérer sur le résultat :

```
In [ ]: for x, y in zip(depart, arrivee2):
            print(f"x={x} => y={y}")
```

Il n'est pas **toujours** possible de remplacer une compréhension par une expression génératrice, mais c'est **souvent souhaitable**, car de cette façon on peut faire de substantielles économies en termes de performances. On peut le faire dès lors que l'on a seulement besoin d'itérer sur les résultats.

Il faut juste un peu se méfier, car comme on parle ici d'itérateurs, comme toujours si on essaie de faire plusieurs fois une boucle sur le même itérateur, il ne se passe plus rien, car l'itérateur a été épuisé :

```
In [ ]: for x, y in zip(depart, arrivee2):
            print(f"x={x} => y={y}")
```

Pour aller plus loin

Vous pouvez regarder cette intéressante discussion de Guido van Rossum sur les compréhensions et les expressions génératrices.

5.11 Les boucles `for`

5.11.1 Exercice - niveau intermédiaire

Produit scalaire

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

On veut écrire une fonction qui retourne le produit scalaire de deux vecteurs. Pour ceci on va matérialiser les deux vecteurs en entrée par deux listes que l'on suppose de même taille.

On rappelle que le produit de X et Y vaut $\sum_i X_i * Y_i$.

On posera que le produit scalaire de deux listes vides vaut `0`.

Naturellement puisque le sujet de la séquence est les expressions génératrices, on vous demande d'utiliser ce trait pour résoudre cet exercice.

```
In [ ]: # un petit exemple
        exo_produit_scalaire.example()
```

Vous devez donc écrire :

```
In [ ]: def produit_scalaire(X, Y):
            """retourne le produit scalaire de deux listes de même taille"""
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_produit_scalaire.correction(produit_scalaire)
```

5.12 Précisions sur l'importation

5.12.1 Complément - niveau basique

Importations multiples - rechargement

Un module n'est chargé qu'une fois De manière générale, à l'intérieur d'un interpréteur python, un module donné n'est chargé qu'une seule fois. L'idée est naturellement que si plusieurs modules différents importent le même module, (ou si un même module en importe un autre plusieurs fois) on ne paie le prix du chargement du module qu'une seule fois.

Voyons cela sur un exemple simpliste, importons un module pour la première fois :

```
In [ ]: import multiple_import
```

Ce module est très simple, comme vous pouvez le voir

```
In [ ]: from modtools import show_module
        show_module(multiple_import )
```

Si on le charge une deuxième fois (peu importe où, dans le même module, un autre module, une fonction..), vous remarquez qu'il ne produit aucune impression

```
In [ ]: import multiple_import
```

Ce qui confirme que le module a déjà été chargé, donc cette instruction `import` n'a aucun effet autre qu'affecter la variable `multiple_import` de nouveau à l'objet module déjà chargé. En résumé, l'instruction `import` fait l'opération d'affectation autant de fois qu'on appelle `import`, mais elle ne charge le module qu'une seule fois à la première importation.

Une autre façon d'illustrer ce trait est d'importer plusieurs fois le module `this`

```
In [ ]: # la première fois le chargement a vraiment lieu
        import this

In [ ]: # la deuxième fois il ne se passe plus rien
        import this
```

Les raisons de ce choix Le choix de ne charger le module qu'une seule fois est motivé par plusieurs considérations.

— D'une part, cela permet à deux modules de dépendre l'un de l'autre (ou plus généralement à avoir des cycles de dépendances), sans avoir à prendre de précaution particulière.

— D'autre part, naturellement, cette stratégie améliore considérablement les performances.

— Marginalement, `import` est une instruction comme une autre, et vous trouverez occasionnellement un avantage à l'utiliser à l'intérieur d'une fonction, **sans aucun surcoût** puisque vous ne payez le prix de l'import qu'au premier appel et non à chaque appel de la fonction.

```
def ma_fonction():
    import un_module_improbable
    ....
```

Cet usage n'est pas recommandé en général, mais de temps en temps peut s'avérer très pratique pour alléger les dépendances entre modules dans des contextes particuliers, comme du code multi-plateformes.

Les inconvénients de ce choix - la fonction `reload` L'inconvénient majeur de cette stratégie de chargement unique est perceptible dans l'interpréteur interactif pendant le développement. Nous avons vu comment IDLE traite le problème en remettant l'interpréteur dans un état vierge lorsqu'on utilise la touche F5. Mais dans l'interpréteur "de base", on n'a pas cette possibilité.

Pour cette raison, python fournit dans le module `importlib` une fonction `reload`, qui permet comme son nom l'indique de forcer le rechargement d'un module, comme ceci :

```
In [ ]: from importlib import reload
        reload(multiple_import)
```

Remarquez bien que `importlib.reload` est une fonction et non une instruction comme `import` - d'où la syntaxe avec les parenthèses qui n'est pas celle de `import`.

Notez également que la fonction `importlib.reload` a été introduite en python3.4, avant, il fallait utiliser la fonction `imp.reload` qui est dépréciée depuis python3.4 mais qui existe toujours. Évidemment, vous devez maintenant exlusivement utiliser la fonction `importlib.reload`.

NOTE spécifique à l'environnement des **notebooks** (en fait, à l'utilisation de ipython) :
À l'intérieur d'un notebook, vous pouvez faire comme ceci pour recharger le code importé automatiquement :

```
In [ ]: # charger le magic 'autoreload'
        %load_ext autoreload

In [ ]: # activer autoreload
        %autoreload 2
```

À partir de cet instant, et si le code d'un module importé est modifié par ailleurs (ce qui est difficile à simuler dans notre environnement), alors le module en question sera effectivement rechargé lors du prochain import. Voyez le lien ci-dessus pour plus de détails.

5.12.2 Complément - niveau avancé

Revenons à python standard. Pour ceux qui sont intéressés par les détails, signalons enfin les deux variables suivantes.

`sys.modules`

L'interpréteur utilise cette variable pour conserver la trace des modules actuellement chargés.

```
In [ ]: import sys
        'csv' in sys.modules

In [ ]: import csv
        'csv' in sys.modules

In [ ]: csv is sys.modules['csv']
```

La documentation sur `sys.modules` indique qu'il est possible de forcer le rechargement d'un module en l'enlevant de cette variable `sys.modules`.

```
In [ ]: del sys.modules['multiple_import']
        import multiple_import
```

`sys.builtin_module_names`

Signalons enfin la variable `sys.builtin_module_names` qui contient le nom des modules, comme par exemple le garbage collector `gc`, qui sont implémentés en C et font partie intégrante de l'interpréteur.

```
In [ ]: 'gc' in sys.builtin_module_names
```

Pour en savoir plus

Pour aller plus loin, vous pouvez lire la documentation sur l'instruction `import`

5.13 Où sont cherchés les modules ?

5.13.1 Complément - niveau basique

Pour les débutants en informatique, le plus simple est de se souvenir que si vous voulez uniquement charger vos propres modules ou packages, il suffit de les placer dans le répertoire où vous lancez la commande python. Si vous n'êtes pas sûr de cet emplacement vous pouvez le savoir en faisant :

```
In [ ]: from pathlib import Path
        Path.cwd()
```

5.13.2 Complément - niveau intermédiaire

Dans ce complément nous allons voir, de manière générale, comment sont localisés (sur le disque dur) les modules que vous chargez dans python grâce à l'instruction `import` ; nous verrons aussi où placer vos propres fichiers pour qu'ils soient accessibles à python.

Comme expliqué ici, lorsque vous importez le module `spam`, python cherche dans cet ordre : * un module *built-in* de nom `spam` - possiblement/probablement écrit en C, * ou sinon un fichier `spam.py` (ou `spam/__init.py__` s'il s'agit d'un package) ; pour le localiser on utilise la variable `sys.path` (c'est-à-dire l'attribut `path` dans le module `sys`), qui est une liste de répertoires, et qui est initialisée avec, dans cet ordre : * le répertoire où se trouve le point d'entrée ; * la variable d'environnement `PYTHONPATH` ; * un certain nombre d'emplacements définis au moment de la compilation de python.

Ainsi sans action particulière de l'utilisateur, python trouve l'intégralité de la librairie standard, ainsi que les modules et packages installés dans le même répertoire que le fichier passé à l'interpréteur.

La façon dont cela se présente dans l'interpréteur des notebooks peut vous induire en erreur. Aussi je vous engage à exécuter plutôt, et sur votre machine, le programme suivant :

```
#!/usr/bin/env python3

import sys

def show_argv_and_path():
    print(f"le point d'entrée du programme est {sys.argv[0]}")
    print(f"la variable sys.path contient")
    for i, path in enumerate(sys.path, 1):
        print(f"{i}-ème chemin dans sys.path {path}")

show_argv_and_path()
```

En admettant que vous rangez ce fichier dans le fichier `/le/repertoire/d/installation/run.py`, et que vous le lancez à partir de `/le/repertoire/ou/vous/etes`, avec une variable `PYTHONPATH` vide, vous devriez observer :

- que la variable `sys.argv[0]` contient le chemin complet `/le/repertoire/d/installation/run.py`,
- et que le premier terme dans `sys.path` contient `/le/repertoire/d/installation`.

La variable d'environnement PYTHONPATH est définie de façon à donner la possibilité d'étendre ces listes depuis l'extérieur, et sans recompiler l'interpréteur, ni modifier les sources.

Cette possibilité s'adresse donc à l'utilisateur final - ou à son administrateur système - plutôt qu'au programmeur.

En tant que programmeur par contre, vous avez la possibilité d'étendre `sys.path` avant de faire vos `import`.

Imaginons par exemple que vous avez écrit un petit outil utilitaire qui se compose d'un point d'entrée `main.py`, et de plusieurs modules `spam.py` et `eggs.py`. Vous n'avez pas le temps de packager proprement cet outil, vous voudriez pouvoir distribuer un *tar* avec ces trois fichiers python, qui puissent s'installer n'importe où (pourvu qu'ils soient tous les trois au même endroit), et que le point d'entrée trouve ses deux modules sans que l'utilisateur ait à s'en soucier.

Imaginons donc ces trois fichiers installés sur machine de l'utilisateur dans :

```
/usr/share/utilitaire/
                      main.py
                      spam.py
                      eggs.py
```

Si vous ne faites rien de particulier, c'est-à-dire que `main.py` contient juste

```
import spam, eggs
```

Alors le programme ne fonctionnera **que s'il est lancé depuis** `/usr/share/utilitaire`, ce qui n'est pas du tout pratique.

Pour contourner cela on peut écrire dans `main.py` quelque chose comme :

```
# on récupère le répertoire où est installé le point d'entrée
import os.path
directory_installation = os.path.dirname(__file__)

# et on l'ajoute au chemin de recherche des modules
import sys
sys.path.append(directory_installation)

# maintenant on peut importer spam et eggs de n'importe où
import spam, eggs
```

Distribuer sa propre librairie avec `setuptools`

Notez bien que l'exemple précédent est **uniquement donné à titre d'illustration** pour décortiquer la mécanique d'utilisation de `sys.path`.

Ce n'est pas une technique recommandée dans le cas général. On préfère en effet de beaucoup diffuser une application python, ou une librairie, sous forme de packaging en utilisant le module setuptools. Il s'agit d'un outil qui **ne fait pas partie de la librairie standard**, et qui supplante `distutils` qui lui, fait partie de la distribution standard mais qui est tombé en déshérence au fil du temps.

`setuptools` permet au programmeur d'écrire - dans un fichier qu'on appelle traditionnellement `setup.py` - le contenu de son application ; grâce à quoi on peut ensuite de manière unifiée : * installer l'application sur une machine à partir des sources ; * préparer un package de l'application ; * diffuser le package dans l'infrastructure PyPI ; * installer le package depuis PyPI en utilisant `pip3`.

Pour installer `setuptools`, comme d'habitude vous pouvez faire simplement :

```
pip3 install setuptools
```

5.14 La clause `import as`

5.14.1 Complément - niveau intermédiaire

Rappel

Jusqu'ici nous avons vu les formes d'importation suivantes :

Importer tout un module D'abord pour importer tout un module

```
import monmodule
```

Importer un symbole dans un module Dans la vidéo nous venons de voir qu'on peut aussi faire :

```
from monmodule import monsymbole
```

Pour mémoire, le langage permet de faire aussi des `import *`, qui est d'un usage déconseillé en dehors de l'interpréteur interactif, car cela crée évidemment un risque de collisions non controlées des espaces de nommage.

`import_module`

Comme vous pouvez le voir, avec `import` on ne peut importer qu'un nom fixe. On ne peut pas calculer le nom d'un module, et le charger ensuite :

```
In [ ]: # si on calcule un nom de module
        modulename = "ma" + "th"
```

on ne peut pas ensuite charger le module math avec import puisque

```
import modulename
```

cherche un module dont le nom est "modulename"
Sachez que vous pourriez utiliser dans ce cas la fonction `import_module` du module `importlib`, qui cette fois permet d'importer un module dont vous avez calculé le nom :

```
In [ ]: from importlib import import_module

In [ ]: loaded = import_module(modulename)
        type(loaded)
```

Nous avons maintenant bien chargé le module `math`, et on l'a rangé dans la variable `loaded`

```
In [ ]: # loaded référence le même objet module que si on avait fait
        # import math
        import math
        math is loaded
```

La fonction `import_module` n'est pas d'un usage très courant, dans la pratique on utilise une des formes de `import` que nous allons voir maintenant, mais `import_module` va me servir à bien illustrer ce que font, précisément, les différentes formes de `import`.

Reprenons

Maintenant que nous savons ce que fait `import_module`, on peut récrire les deux formes d'`import` de cette façon :

```
In [ ]: # un import simple
        import math

In [ ]: # peut se récrire
        math = import_module('math')
```

Et :

```
In [ ]: # et un import from
        from pathlib import Path

In [ ]: # est en gros équivalent à
        tmp = import_module('pathlib')
        Path = tmp.Path
        del tmp
```

`import as`

Tout un module Dans chacun de ces deux cas, on n'a pas le choix du nom de l'entité importée, et cela pose parfois problème.

Il peut arriver d'écrire un module sous un nom qui semble bien choisi, mais on se rend compte au bout d'un moment qu'il entre en conflit avec un autre symbole.

Par exemple, vous écrirez un module dans un fichier `globals.py` et vous l'importez dans votre code

```
import globals
```

Puis un moment après pour débugger vous voulez utiliser la fonction *builtin* `globals`. Sauf que, en vertu de la règle LEGB, le symbole `globals` se trouve maintenant désigner votre module, et non la fonction.

À ce stade évidemment vous pouvez (devriez) renommer votre module, mais cela peut prendre du temps parce qu'il y a de nombreuses dépendances. En attendant vous pouvez tirer profit de la clause `import as` dont la forme générale est :

```
import monmodule as autremodule
```

ce qui, toujours à la grosse louche, est équivalent à :

```
autremodule = import_module('monmodule')
```

Un symbole dans un module On peut aussi importer un symbole spécifique d'un module, sous un autre nom que celui qu'il a dans le module. Ainsi :

```
from monmodule import monsymbole as autresymbole
```

qui fait quelque chose comme :

```
temporaire = import_module('monmodule')
autresymbole = temporaire.monsymbole
del temporaire
```

Quelques exemples

J'ai écrit des modules jouet : * `un_deux` qui définit des fonctions `un` et `deux` ; * `un_deux_trois` qui définit des fonctions `un`, `deux` et `trois` ; * `un_deux_trois_quatre` qui définit, eh oui, des fonctions `un`, `deux`, `trois` et `quatre`.

Toutes ces fonctions se contentent d'écrire leur nom et leur module.

```
In [ ]: # changer le nom du module importé
        import un_deux as one_two
        one_two.un()

In [ ]: # changer le nom d'un symbole importé du module
        from un_deux_trois import un as one
        one()

In [ ]: # on peut mélanger tout ça
        from un_deux_trois_quatre import un as one, deux, trois as three

In [ ]: one()
        deux()
        three()
```

Pour en savoir plus

Vous pouvez vous reporter à la section sur l'instruction `import` dans la documentation python.

5.15 Récapitulatif sur `import`

5.15.1 Complément - niveau basique

Nous allons récapituler les différentes formes d'importation, et introduire la clause `import *` - et voir pourquoi il est déconseillé de l'utiliser.

Importer tout un module

L'`import` le plus simple consiste donc à uniquement mentionner le nom du module

```
In [ ]: import un_deux
```

Ce module se contente de définir deux fonctions de noms `un` et `deux`. Une fois l'import réalisé de cette façon, on peut accéder au contenu du module en utilisant un nom de variable complet :

```
In [ ]: # la fonction elle-même
        print(un_deux.un)

        un_deux.un()
```

Mais bien sûr on n'a pas de cette façon défini de nouvelle variable `un` ; la seule nouvelle variable dans la portée courante est donc `un_deux` :

```
In [ ]: # dans l'espace de nommage courant on peut accéder au module lui-même
        print(un_deux)
```

```
In [ ]: # mais pas à la variable `un`
        try:
            print(un)
        except NameError:
            print("La variable 'un' n'est pas définie")
```

Importer une variable spécifique d'un module

On peut également importer un ou plusieurs symboles spécifiques d'un module en faisant maintenant (avec un nouveau module du même tonneau) :

```
In [ ]: from un_deux_trois import un, deux
```

À présent nous avons deux nouvelles variables dans la portée locale :

```
In [ ]: un()
        deux()
```

Et cette fois, c'est le module lui-même qui n'est pas accessible :

```
In [ ]: try:
            print(un_deux_trois)
        except NameError:
            print("La variable 'un_deux_trois' n'est pas définie")
```

Il est important de voir que la variable locale ainsi créée, un peu comme dans le cas d'un appel de fonction, est une **nouvelle variable** qui est initialisée avec l'objet du module. Ainsi si on importe le module **et** une variable du module comme ceci :

```
In [ ]: import un_deux_trois
```

alors nous avons maintenant **deux variables différentes** qui désignent la fonction un dans le module :

```
In [ ]: print(un_deux_trois.un)
        print(un)
        print("ce sont deux façons d'accéder au même objet", un is un_deux_trois.un)
```

En on peut modifier l'une **sans affecter** l'autre :

```
In [ ]: # les deux variables sont différentes
        # un n'est pas un 'alias' vers un_deux_trois.un
        un = 1
        print(un_deux_trois.un)
        print(un)
```

5.15.2 Complément - niveau intermédiaire

`import .. as`

Que l'on importe avec la forme `import unmodule` ou avec la forme `from unmodule import unevariable`, on peut toujours ajouter une clause `as nouveaunom`, qui change le nom de la variable qui est ajoutée dans l'environnement courant.

Ainsi :

— `import foo` définit une variable `foo` qui désigne un module ;
— `import foo as bar` a le même effet, sauf que le module est accessible par la variable `bar` ;

Et :

— `from foo import var` définit une variable `var` qui désigne un attribut du module ;
— `from foo import var as newvar` définit une variable `newvar` qui désigne ce même attribut.

Ces deux formes sont pratiques pour éviter les conflits de nom.

```
In [ ]: # par exemple
        import un_deux as mod12
        mod12.un()
```

```
In [ ]: from un_deux import deux as m12deux
        m12deux()
```

```
import *
```

La dernière forme d'`import` consiste à importer toutes les variables d'un module comme ceci :

```
In [ ]: from un_deux_trois_quatre import *
```

Cette forme, pratique en apparence, va donc créer dans l'espace de nommage courant les variables

```
In [ ]: un()
        deux()
        trois()
        quatre()
```

Quand utiliser telle ou telle forme

Les deux premières formes - import d'un module ou de variables spécifiques - peuvent être utilisées indifféremment ; souvent lorsqu'une variable est utilisée très souvent dans le code on pourra préférer la deuxième forme pour raccourcir le code.

À cet égard, citons des variantes de ces deux formes qui permettent d'utiliser des noms plus courts. Vous trouverez par exemple très souvent

```
import numpy as np
```

qui permet d'importer le module numpy mais de l'utiliser sous un nom plus court - car avec `numpy` on ne cesse d'utiliser des symboles dans le module.

Avertissement : nous vous recommandons de **ne pas utiliser la dernière forme** `import *` - sauf dans l'interpréteur interactif - car cela peut gravement nuire à la lisibilité de votre code.

python est un langage à liaison statique ; cela signifie que lorsque vous concentrez votre attention sur un (votre) module, et que vous voyez une référence en lecture à une variable `spam` disons à la ligne 201, vous devez forcément trouver dans les deux cents premières lignes quelque chose comme une déclaration de `spam`, qui vous indique en gros d'où elle vient.

`import *` est une construction qui casse cette bonne propriété (pour être tout à fait exhaustif, cette bonne propriété n'est pas non plus remplie avec les fonctions *built-in* comme `len`, mais il faut vivre avec...)

Mais le point important est ceci : imaginez que dans un module vous faites plusieurs `import *` comme par exemple

```
from django.db import *
from django.conf.urls import *
```

Peu importe le contenu exact de ces deux modules, il nous suffit de savoir qu'un des deux modules expose la variable `patterns`.

Dans ce cas de figure vécu, le module utilise cette variable `patterns` sans avoir besoin de la déclarer explicitement, si bien qu'à la lecture on voit une utilisation de la variable `patterns`, mais on n'a plus aucune idée de quel module elle provient, sauf à aller lire le code correspondant...

5.15.3 Complément - niveau avancé

`import` de manière "programmative"

Étant donné la façon dont est conçue l'instruction `import`, on rencontre une limitation lorsqu'on veut, par exemple, **calculer le nom d'un module** avant de l'importer.

Si vous êtes dans ce genre de situation, reportez-vous au module `importlib` et notamment sa fonction `import_module` qui, cette fois, accepte en argument une chaîne.

Voici une illustration dans un cas simple. Nous allons importer le module `modtools` (qui fait partie de ce MOOC) de deux façons différentes et montrer que le résultat est le même :

```
In [ ]: # on importe la fonction 'import_module' du module 'importlib'
        from importlib import import_module

        # grâce à laquelle on peut importer à partir d'un string
        imported_modtools = import_module('mod' + 'tools')

        # on peut aussi importer modtools "normalement"
        import modtools

        # les deux objets sont identiques
        imported_modtools is modtools
```

5.16 La notion de package

5.16.1 Complément - niveau basique

Dans ce complément, nous approfondissons la notion de module, qui a été introduite dans les vidéos, et nous décrivons la notion de *package* qui permet de créer des bibliothèques plus structurées qu'avec un simple module.

Pour ce notebook nous aurons besoin de deux utilitaires pour voir le code correspondant aux modules et packages que nous manipulons :

```
In [ ]: from modtools import show_module, show_package
```

Rappel sur les modules

Nous avons vu dans la vidéo qu'on peut charger une bibliothèque, lorsqu'elle se présente sous la forme d'un seul fichier source, au travers d'un objet python de type **module**.

Chargeons un module "jouet" :

```
In [ ]: import module_simple
```

Voyons à quoi ressemble ce module :

```
In [ ]: show_module(module_simple)
```

On a bien compris maintenant que le module joue le rôle d'**espace de nom**, dans le sens où :

```
In [ ]: # on peut définir sans risque une variable globale 'spam'
        spam = 'eggs'
        # qui est indépendante de celle définie dans le module
        print("spam globale", spam)
        print("spam du module", module_simple.spam)
```

Pour résumer, un module est donc un objet python qui correspond à la fois à : * un (seul) **fichier** sur le disque ; * et un **espace de nom** pour les variables du programme.

La notion de package

Lorsqu'il s'agit d'implémenter une très grosse bibliothèque, il n'est pas concevable de tout concentrer en un seul fichier. C'est là qu'intervient la notion de **package**, qui est un peu aux **répertoires** ce que que le **module** est aux **fichiers**.

On importe un package exactement comme un module :

```
In [ ]: import package_jouet
```

```
In [ ]: package_jouet.module_jouet
```

Le package porte **le même nom** que le répertoire, c'est-à-dire que, de même que le module `module_jouet` correspond au fichier `module_jouet.py`, le package python `package_jouet` correspond au répertoire `package_jouet`.

Pour définir un package, il faut **obligatoirement** créer dans le répertoire (celui, donc, que l'on veut exposer à python), un fichier nommé `__init__.py`. Voilà comment a été implémenté le package que nous venons d'importer :

```
In [ ]: show_package(package_jouet)
```

Comme on le voit, importer un package revient essentiellement à charger le fichier `__init__.py` correspondant. Le package se présente aussi comme un espace de nom, à présent on a une troisième variable `spam` qui est encore différente des deux autres :

```
In [ ]: package_jouet.spam
```

L'avantage principal du package par rapport au module est qu'il peut contenir d'autres packages ou modules. Dans notre cas, `package_jouet` vient avec un module qu'on peut importer comme un attribut du package, c'est-à-dire comme ceci :

```
In [ ]: import package_jouet.module_jouet
```

À nouveau regardons comment cela est implémenté ; le fichier correspondant au module se trouve naturellement à l'intérieur du répertoire correspondant au package, c'était le but du jeu au départ :

```
In [ ]: show_module(package_jouet.module_jouet)
```

Vous remarquerez que le module `module_jouet` a été chargé au même moment que `package_jouet`. Ce comportement **n'est pas implicite**. C'est nous qui avons explicitement choisi d'importer le module dans le package (dans `__init__.py`).

Cette technique correpond à un usage assez fréquent, où on veut exposer directement dans l'espace de nom du package des symboles qui sont en réalité définis dans un module.

Avec le code ci-dessus, après avoir importé `package_jouet`, nous pouvons utiliser

```
In [ ]: package_jouet.jouet
```

alors qu'en fait il faudrait écrire en toute rigueur

```
In [ ]: package_jouet.module_jouet.jouet
```

Mais cela impose alors à l'utilisateur d'avoir une connaissance sur l'organisation interne de la bibliothèque, ce qui est considéré comme une mauvaise pratique.

D'abord, cela donne facilement des noms à rallonge et du coup nuit à la lisibilité, ce n'est pas pratique. Mais surtout, que se passerait-il alors si le développeur du package voulait renommer des modules à l'intérieur de la bibliothèque ? On ne veut pas que ce genre de décision ait un impact sur les utilisateurs.

À quoi sert `__init__.py` ?

Le code placé dans `__init__.py` est chargé d'initialiser la bibliothèque. Le fichier **peut être vide** mais **doit absolument exister**. Nous vous mettons en garde car c'est une erreur fréquente de l'oublier. Sans lui vous ne pourrez importer ni le package, ni les modules ou sous-packages qu'il contient.

C'est ce fichier qui est chargé par l'interpréteur python lorsque vous importez le package. Comme pour les modules, le fichier n'est chargé qu'une seule fois par l'interpréteur python, s'il rencontre plus tard à nouveau le même `import`, il l'ignore silencieusement.

5.16.2 Complément - niveau avancé

Variables spéciales

Comme on le voit dans les exemples, certaines variables *spéciales* peuvent être lues ou écrites dans les modules ou packages. Voici les plus utilisées :

`__name__`

```
In [ ]: print(package_jouet.__name__, package_jouet.module_jouet.__name__)
```

Remarquons à cet égard que le **point d'entrée** du programme (c'est-à-dire, on le rappelle, le fichier qui est passé directement à l'interpréteur python) est considéré comme un module dont l'attribut `__name__` vaut la chaîne `"__main__"`

C'est pourquoi (et c'est également expliqué ici) les scripts python se terminent généralement par une phrase du genre de

```
if __name__ == "__main__":
    <faire vraiment quelque chose>
    <comme par exemple tester le module>
```

Cet idiome très répandu permet d'attacher du code à un module lorsqu'on le passe directement à l'interpréteur python.

`__file__`

```
In [ ]: print(package_jouet.__file__)
        print(package_jouet.module_jouet.__file__)
```

`__all__` Il est possible de redéfinir dans un package la variable `__all__`, de façon à définir les symboles qui sont réellement concernés par un `import *`, comme c'est décrit ici.

Pour en savoir plus

Voir la section sur les modules dans la documentation python, et notamment la section sur les packages.

5.17 Décoder le module `this`

5.17.1 Exercice - niveau avancé

Le module `this` et le *zen de python*

Nous avons déjà eu l'occasion de parler du *zen de python* ; on peut lire ce texte en important le module `this` comme ceci

```
In [ ]: import this
```

Il suit du cours qu'une fois cet import effectué nous avons accès à une variable `this`, de type module :

```
In [ ]: this
```

But de l'exercice

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

Constatant que le texte du manifeste doit se trouver quelque part dans le module, le but de l'exercice est de deviner le contenu du module, et d'écrire une fonction `decode_zen`, qui retourne le texte du manifeste.

Indices

Cet exercice peut paraître un peu déconcertant ; voici quelques indices optionnels :

```
In [ ]: # on rappelle que dir() renvoie les noms des attributs
        # accessibles à partir de l'objet
        dir(this)
```

Vous pouvez ignorer `this.c` et `this.i`, les deux autres variables du module sont importantes pour nous.

```
In [ ]: # ici on calcule le résultat attendu
        resultat = exo_decode_zen.resultat(this)
```

Ceci devrait vous donner une idée de comment utiliser une des deux variables du module :

```
In [ ]: # ces deux quantités sont égales
        len(this.s) == len(resultat)
```

À quoi peut bien servir l'autre variable ?

```
In [ ]: # ce pourrait-il que d agisse comme un code simple ?
        this.d[this.s[0]] == resultat[0]
```

Le texte comporte certes des caractères alphabétiques

```
In [ ]: # si on ignore les accents,
        # il y a 26 caractères minuscule
        # et 26 caractères majuscule
        len(this.d)
```

mais pas seulement ; les autres sont préservés.

À vous de jouer

```
In [ ]: def decode_zen(this):
            "<votre code>"
```

Correction

```
In [ ]: exo_decode_zen.correction(decode_zen)
```

SEMAINE 6

CONCEPTION DES CLASSES

6.1 Introduction aux classes

6.1.1 Complément - niveau basique

On définit une classe lorsqu'on a besoin de créer un type spécifique au contexte de l'application. Il faut donc voir une classe au même niveau qu'un type *builtin* comme `list` ou `dict`.

Un exemple simpliste

Par exemple, imaginons qu'on a besoin de manipuler des matrices 2×2
$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$
Et en guise d'illustration, nous allons utiliser le déterminant ; c'est juste un prétexte pour implémenter une méthode sur cette classe, ne vous inquiétez pas si le terme ne vous dit rien, ou vous rappelle de mauvais souvenirs. Tout ce qu'on a besoin de savoir c'est que, sur une matrice de ce type, le déterminant vaut :
$ det (A) = a_{11}.a_{22} - a_{12}.a_{21}$
Dans la pratique, on utiliserait la classe `matrix` de `numpy` qui est une bibliothèque de calcul scientifique très populaire et largement utilisée. Mais comme premier exemple de classe, nous allons écrire **notre propre classe** `Matrix2` pour mettre en action les mécanismes de base des classes de python. Naturellement, il s'agit d'une implémentation jouet.

```
In [ ]: class Matrix2:
            "Une implémentation sommaire de matrice carrée 2x2"

            def __init__(self, a11, a12, a21, a22):
                "construit une matrice à partir des 4 coefficients"
                self.a11 = a11
                self.a12 = a12
                self.a21 = a21
                self.a22 = a22

            def determinant(self):
                "renvoie le déterminant de la matrice"
                return self.a11 * self.a22 - self.a12 * self.a21
```

La première version de `Matrix2`

Une classe peut avoir un *docstring* Pour commencer, vous remarquez qu'on peut attacher à cette classe un *docstring* comme pour les fonctions

```
In [ ]: help(Matrix2)
```

La classe définit donc deux méthodes, nommées `__init__` et `determinant`.

La méthode `__init__` La méthode `__init__`, comme toutes celles qui ont un nom en *__nom__*, est une **méthode spéciale**. En l'occurrence, il s'agit de ce qu'on appelle le **constructeur** de la classe, c'est-à-dire le code qui va être appelé lorsqu'on crée une instance. Voyons cela tout de suite sur un exemple.

```
In [ ]: matrice = Matrix2(1, 2, 2, 1)
        print(matrice)
```

Vous remarquez tout d'abord que `__init__` s'attend à recevoir *5 arguments*, mais que nous appelons `Matrix2` avec seulement *4 arguments*.

L'argument surnuméraire, le **premier** de ceux qui sont déclarés dans la méthode, correspond à l'**instance qui vient d'être créée** et qui est automatiquement passée par l'interpréteur python à la méthode `__init__`. En ce sens, le terme constructeur est impropre puisque la méthode `__init__` ne crée pas l'instance, elle ne fait que l'initialiser, mais c'est un abus de langage très répandu. Nous reviendrons sur le processus de création des objets lorsque nous parlerons des métaclasses en dernière semaine.

La **convention** est de nommer le premier argument de ce constructeur `self`, nous y reviendrons un peu plus loin.

On voit également que le constructeur se contente de mémoriser, à l'intérieur de l'instance, les arguments qu'on lui passe, sous la forme d'**attributs** de l'**instance** `self`.

C'est un cas extrêmement fréquent ; de manière générale, il est recommandé d'écrire des constructeurs passifs de ce genre ; dit autrement, on évite de faire trop de traitements dans le constructeur.

La méthode `determinant` La classe définit aussi la méthode `determinant`, qu'on utiliserait comme ceci :

```
In [ ]: matrice.determinant()
```

Vous voyez que la **syntaxe** pour appeler une méthode sur un objet est **identique** à celle que nous avons utilisée jusqu'ici avec **les types de base**. Nous verrons très bientôt comment on peut pousser beaucoup plus loin la similitude, pour pouvoir par exemple calculer la **somme** de deux objets de la classe `Matrix2` avec l'opérateur `+`, mais n'anticipons pas.

Vous voyez aussi que, ici encore, la méthode définie dans la classe attend ***1 argument*** `self`, alors qu'apparemment nous ne lui en passons **aucun**. Comme tout à l'heure avec le constructeur, le premier argument passé automatiquement par l'interpréteur python à `determinant` est l'objet `matrice` lui-même.

En fait on aurait pu aussi bien écrire, de manière parfaitement équivalente :

```
In [ ]: Matrix2.determinant(matrice)
```

qui n'est presque jamais utilisé en pratique, mais qui illustre bien ce qui se passe lorsqu'on invoque une méthode sur un objet. En réalité, lorsque l'on écrit `matrice.determinant()` l'interpréteur python va essentiellement convertir cette expression en `Matrix2.determinant(matrice)`.

6.1.2 Complément - niveau intermédiaire

À quoi ça sert ?

Ce cours n'est pas consacré à la Programmation Orientée Objet (OOP) en tant que telle. Voici toutefois quelques-uns des avantages qui sont généralement mis en avant :

— encapsulation ;
— résolution dynamique de méthode ;
— héritage.

Encapsulation L'idée de la notion d'encapsulation consiste à ce que : * une classe définit son **interface**, c'est-à-dire les méthodes par lesquelles on peut utiliser ce code, * mais reste tout à fait libre de modifier son **implémentation**, et tant que cela n'impacte pas l'interface, **aucun changement** n'est requis dans les **codes utilisateurs**.

Nous verrons plus bas une deuxième implémentation de `Matrix2` qui est plus générale que notre première version, mais qui utilise la même interface, donc qui fonctionne exactement de la même manière pour le code utilisateur.

La notion d'encapsulation peut paraître à première vue banale ; il ne faut pas s'y fier, c'est de cette manière qu'on peut efficacement découper un gros logiciel en petits morceaux indépendants, et réellement découplés les uns des autres, et ainsi casser, réduire la complexité.

La programmation objet est une des techniques permettant d'atteindre cette bonne propriété d'encapsulation. Il faut reconnaître que certains langages comme Java et C++ ont des mécanismes plus sophistiqués, mais aussi plus complexes, pour garantir une bonne étanchéité entre l'interface publique et les détails d'implémentation. Les choix faits en la matière en python reviennent, une fois encore, à privilégier la simplicité.

Aussi, il n'existe pas en python l'équivalent des notions d'interface `public`, `private` et `protected` qu'on trouve en C++ et en Java. Il existe tout au plus une convention, selon laquelle les attributs commençant par un underscore (le tiret bas `_`) sont privés et ne *devraient* pas être utilisés par un code tiers, mais le langage ne fait rien pour garantir le bon usage de cette convention.

Si vous désirez creuser ce point nous vous conseillons de lire : * *Reserved classes of identifiers* où l'on décrit également les noms privés à une classe (les noms de variables en `__nom`) ; * *Private Variables and Class-local References*, qui en donne une illustration.

Malgré cette simplicité revendiquée, les classes de python permettent d'implémenter en pratique une encapsulation tout à fait acceptable, on peut en juger rien que par le nombre de bibliothèques tierces existantes dans l'écosystème python.

Résolution dynamique de méthode Le deuxième atout de OOP, c'est le fait que l'envoi de méthode est résolu lors de l'exécution (*run-time*) et non pas lors de la compilation (*compile-time*). Ceci signifie que l'on peut écrire du code générique, qui pourra fonctionner avec des objets non connus *à priori*. Nous allons en voir un exemple tout de suite, en redéfinissant le comportement de `print` dans la deuxième implémentation de `Matrix2`.

Héritage L'héritage est le concept qui permet de : * dupliquer une classe presque à l'identique, mais en redéfinissant une ou quelques méthodes seulement (héritage simple) ; * composer plusieurs classes en une seule, pour réaliser en quelque sorte l'union des propriétés de ces classes (héritage multiple).

Illustration

Nous revenons sur l'héritage dans une prochaine vidéo. Dans l'immédiat, nous allons voir une seconde implémentation de la classe `Matrix2`, qui illustre l'encapsulation et l'envoi dyna-

mique de méthodes.

Pour une raison ou pour une autre, disons que l'on décide de remplacer les 4 attributs nommés `self.a11`, `self.a12`, etc., qui n'étaient pas très extensibles, par un seul attribut `a` qui regroupe tous les coefficients de la matrice dans un seul tuple.

```
In [ ]: class Matrix2:
            """Une deuxième implémentation, tout aussi
            sommaire, mais différente, de matrice carrée 2x2"""

            def __init__(self, a11, a12, a21, a22):
                "construit une matrice à partir des 4 coefficients"
                # on décide d'utiliser un tuple plutôt que de ranger
                # les coefficients individuellement
                self.a = (a11, a12, a21, a22)

            def determinant(self):
                "le déterminant de la matrice"
                return self.a[0] * self.a[3] - self.a[1] * self.a[2]

            def __repr__(self):
                "comment présenter une matrice dans un print()"
                return f"<<mat-2x2 {self.a}>>"
```

Grâce à l'**encapsulation**, on peut continuer à utiliser la classe exactement de la même manière :

```
In [ ]: matrice = Matrix2(1, 2, 2, 1)
        print("Determinant =", matrice.determinant())
```

Et en prime, grâce à la **résolution dynamique de méthode**, et parce que dans cette seconde implémentation on a défini une autre méthode spéciale `__repr__`, nous avons maintenant une impression beaucoup plus lisible de l'objet `matrice` :

```
In [ ]: print(matrice)
```

Ce format d'impression reste d'ailleurs valable dans l'impression d'objets plus compliqués, comme par exemple :

```
In [ ]: # on profite de ce nouveau format d'impression même si on met
        # par exemple un objet Matrix2 à l'intérieur d'une liste
        composite = [matrice, None, Matrix2(1, 0, 0, 1)]
        print(f"composite={composite}")
```

Cela est possible parce que le code de `print` envoie la méthode `__repr__` sur les objets qu'elle parcourt. Le langage fournit une façon de faire par défaut, comme on l'a vu plus haut avec la première implémentation de `Matrix2` ; et en définissant notre propre méthode `__repr__` nous pouvons surcharger ce comportement, et définir notre format d'impression.

Nous reviendrons sur les notions de surcharge et d'héritage dans les prochaines séquences vidéos.

La convention d'utiliser `self`

Avant de conclure, revenons rapidement sur le nom `self` qui est utilisé comme nom pour le premier argument des méthodes habituelles (nous verrons en semaine 9 d'autres sortes de méthodes, les méthodes statiques et de classe, qui ne reçoivent pas l'instance comme premier argument).

Comme nous l'avons dit plus haut, le premier argument d'une méthode s'appelle `self` **par convention**. Cette pratique est particulièrement bien suivie, mais ce n'est qu'une convention, en ce sens qu'on aurait pu utiliser n'importe quel identificateur ; pour le langage `self` n'a aucun sens particulier, ce n'est pas un mot clé ni une variable *built-in*.

Ceci est à mettre en contraste avec le choix fait dans d'autres langages, comme par exemple en C++ où l'instance est référencée par le mot-clé `this`, qui n'est pas mentionné dans la signature de la méthode. En python, selon le manifeste, *explicit is better than implicit*, c'est pourquoi on mentionne l'instance dans la signature, sous le nom `self`.

6.2 Enregistrements et instances

6.2.1 Complément - niveau basique

Un enregistrement implémenté comme une instance de classe

Nous reprenons ici la discussion commencée en semaine 3, où nous avions vu comment implémenter un enregistrement comme un dictionnaire. Un enregistrement est l'équivalent, selon les langages, de *struct* ou *record*.

Notre exemple était celui des personnes, et nous avions alors écrit quelque chose comme :

```
In [ ]: pierre = {'nom': 'pierre', 'age': 25, 'email': 'pierre@foo.com'}
        print(pierre)
```

Cette fois-ci nous allons implémenter la même abstraction, mais avec une classe `Personne` comme ceci :

```
In [ ]: class Personne:
            """Une personne possède un nom, un âge et une adresse e-mail"""

            def __init__(self, nom, age, email):
                self.nom = nom
                self.age = age
                self.email = email

            def __repr__(self):
                # comme nous avons la chance de disposer de python-3.6
                # utilisons un f-string
                return f"<<{self.nom}, {self.age} ans, email:{self.email}>>"
```

Le code de cette classe devrait être limpide à présent ; voyons comment on l'utiliserait - en guise rappel sur le passage d'arguments aux fonctions :

```
In [ ]: personnes = [

            # on se fie à l'ordre des arguments dans le créateur
            Personne('pierre', 25, 'pierre@foo.com'),
```

```
        # ou bien on peut être explicite
        Personne(nom='paul', age=18, email='paul@bar.com'),

        # ou bien on mélange
        Personne('jacques', 52, email='jacques@cool.com'),
    ]
    for personne in personnes:
        print(personne)
```

Un dictionnaire pour indexer les enregistrements

Nous pouvons appliquer exactement la même technique d'indexation qu'avec les dictionnaires :

```
In [ ]: # on crée un index pour pouvoir rechercher efficacement
        # une personne par son nom
        index_par_nom = {personne.nom: personne for personne in personnes}
```

De façon à pouvoir facilement localiser une personne :

```
In [ ]: pierre = index_par_nom['pierre']
        print(pierre)
```

Encapsulation

Pour marquer l'anniversaire d'une personne, nous pourrions faire :

```
In [ ]: pierre.age += 1
        pierre
```

À ce stade, surtout si vous venez de C++ ou de Java, vous devriez vous dire que ça ne va pas du tout !

En effet, on a parlé dans le complément précédent des mérites de l'encapsulation, et vous vous dites que là, la classe n'est pas du tout encapsulée car le code utilisateur a besoin de connaître l'implémentation.

En réalité, avec les classes python on a la possibilité, grâce aux *properties*, de conserver ce style de programmation qui a l'avantage d'être très simple, tout en préservant une bonne encapsulation, comme on va le voir dans le prochain complément.

6.2.2 Complément - niveau intermédiaire

Illustrons maintenant qu'en python on peut ajouter des méthodes à une classe *à la volée* - c'est-à-dire en dehors de l'instuction `class`.

Pour cela on tire simplement profit du fait que **les méthodes sont implémentées comme des attributs de l'objet classe**.

Ainsi, on peut étendre l'objet `classe` lui-même dynamiquement :

```
In [ ]: # pour une implémentation réelle voyez la bibliothèque smtplib
        # https://docs.python.org/3/library/smtplib.html

        def sendmail(self, subject, body):
            "Envoie un mail à la personne"
            print(f"To: {self.email}")
```

```
        print(f"Subject: {subject}")
        print(f"Body: {body}")

    Personne.sendmail = sendmail
```

Ce code commence par définir une fonction en utilisant `def` et la signature de la méthode. La fonction accepte un premier argument `self` ; exactement comme si on avait défini la méthode dans l'instruction `class`.

Ensuite, il suffit d'affecter la fonction ainsi définie à **l'attribut** `sendmail` de l'objet classe.

Vous voyez que c'est très simple, et à présent la classe a connaissance de cette méthode exactement comme si on l'avait définie dans la clause `class`, comme le montre l'aide :

```
In [ ]: help(Personne)
```

Et on peut à présent utiliser cette méthode :

```
In [ ]: pierre.sendmail("Coucou", "Salut ça va ?")
```

6.3 Les *property*

Note : nous reviendrons largement sur cette notion de property lorsque nous parlerons des *property et descripteurs* en semaine 9. Cependant, cette notion est suffisamment importante pour que nous vous proposions un complément dès maintenant dessus.

6.3.1 Complément - niveau intermédiaire

Comme on l'a vu dans le complément précédent, il est fréquent en python qu'une classe expose dans sa documentation un ou plusieurs attributs ; c'est une pratique qui, en apparence seulement, paraît casser l'idée d'une bonne encapsulation.

En réalité, grâce au mécanisme de *property*, il n'en est rien. Nous allons voir dans ce complément comment une classe peut en quelque sorte intercepter les accès à ses attributs, et par là fournir une encapsulation forte.

Pour être concret, on va parler d'une classe `Temperature`. Au lieu de proposer, comme ce serait l'usage dans d'autres langages, une interface avec `get_kelvin()` et `set_kelvin()`, on va se contenter d'exposer l'attribut `kelvin`, et malgré cela on va pouvoir faire diverses vérifications et autres.

Implémentation naïve Je vais commencer par une implémentation naïve, qui ne tire pas profit des *properties* :

```
In [ ]: # dans sa version la plus épurée, une classe
        # température pourrait ressembler à ça :

        class Temperature1:
            def __init__(self, kelvin):
                self.kelvin = kelvin

            def __repr__(self):
                return f"{self.kelvin}K"
```

```
In [ ]: # créons une instance
        t1 = Temperature1(20)
        t1
```

```
In [ ]: # et pour accéder à la valeur numérique je peux faire
        t1.kelvin
```

Avec cette implémentation il est très facile de créer une température négative, qui n'a bien sûr pas de sens physique, ce n'est pas bon.

Interface *getter/setter* Si vous avez été déjà exposés à des langages orientés objet comme C++, Java ou autre, vous avez peut-être l'habitude d'accéder aux données internes des instances par des **méthodes** de type *getter* ***ou*** *setter*, de façon à contrôler les accès et, dans une optique d'encapsulation, de préserver des invariants, comme ici le fait que la température doit être positive.

C'est-à-dire que vous vous dites peut-être, ça ne devrait pas être fait comme ça, on devrait plutôt proposer une interface pour accéder à l'implémentation interne ; quelque chose comme :

```
In [ ]: class Temperature2:
            def __init__(self, kelvin):
                # au lieu d'écrire l'attribut il est plus sûr
                # d'utiliser le setter
                self.set_kelvin(kelvin)

            def set_kelvin(self, kelvin):
                # je m'assure que _kelvin est toujours positif
                # et j'utilise un nom d'attribut avec un _ pour signifier
                # que l'attribut est privé et qu'il ne faut pas y toucher directement
                # on pourrait aussi bien sûr lever une exception
                # mais ce n'est pas mon sujet ici
                self._kelvin = max(0, kelvin)

            def get_kelvin(self):
                return self._kelvin

            def __repr__(self):
                return f"{self._kelvin}K"
```

Bon c'est vrai que d'un coté, c'est mieux parce que je garantis un invariant, la température est toujours positive :

```
In [ ]: t2 = Temperature2(-30)
        t2
```

Mais par contre, d'un autre coté, c'est très lourd, parce que chaque fois que je veux utiliser mon objet, je dois faire pour y accéder :

```
In [ ]: t2.get_kelvin()
```

et aussi, si j'avais déjà du code qui utilisait `t.kelvin` il va falloir le modifier entièrement.

Implémentation pythonique La façon de s'en sortir ici consiste à définir une property. Comme on va le voir ce mécanisme permet d'écrire du code qui fait référence à l'attribut `kelvin` de l'instance, mais qui passe tout de même par une couche de logique.

Ça ressemblerait à ceci :

```
In [ ]: class Temperature3:
            def __init__(self, kelvin):
                self.kelvin = kelvin

            # je définis bel et bien mes accesseurs de type getter et setter
            # mais _get_kelvin commence avec un _
            # car il n'est pas censé être appelé par l'extérieur
            def _get_kelvin(self):
                return self._kelvin

            # idem
            def _set_kelvin(self, kelvin):
                self._kelvin = max(0, kelvin)

            # une fois que j'ai ces deux éléments je peux créer une property
            kelvin = property(_get_kelvin, _set_kelvin)

            # et toujours la façon d'imprimer
            def __repr__(self):
                return f"{self._kelvin}K"
```

```
In [ ]: t3 = Temperature3(200)
        t3
```

```
In [ ]: # par contre ici on va le mettre à zéro
        # à nouveau, une exception serait préférable sans doute
        t3.kelvin = -30
        t3
```

Comme vous pouvez le voir, cette technique a plusieurs avantages : * on a ce qu'on cherchait, c'est-à-dire une façon d'ajouter une couche de logique lors des accès en lecture et en écriture à l'intérieur de l'objet, * mais **sans toutefois** demander à l'utilisateur de passer son temps à envoyer des méthodes `get_` et `set()` sur l'objet, ce qui a tendance à alourdir considérablement le code.

C'est pour cette raison que vous ne rencontrerez presque jamais en python une bibliothèque qui offre une interface à base de méthodes `get_something` et `set_something`, mais au contraire les API vous exposeront directement des attributs que vous devez utiliser directement.

6.3.2 Complément - niveau avancé

À titre d'exemple d'utlisation, voici une dernière implémentation de `Temperature` qui donne l'illusion d'avoir 3 attributs (`kelvin`, `celsius` et `fahrenheit`), alors qu'en réalité le seul attribut de donnée est `_kelvin`.

```
In [ ]: class Temperature:

            ## les constantes de conversion
            # kelvin / celsius
            K = 273.16
            # fahrenheit / celsius
            RF = 5 / 9
```

```
    KF = (K / RF) - 32

    def __init__(self, kelvin=None, celsius=None, fahrenheit=None):
        """
        Création à partir de n'importe quelle unité
        Il faut préciser exactement une des trois unités
        """
        # on passe par les properties pour initialiser
        if kelvin is not None:
            self.kelvin = kelvin
        elif celsius is not None:
            self.celsius = celsius
        elif fahrenheit is not None:
            self.fahrenheit = fahrenheit
        else:
            self.kelvin = 0
            raise ValueError("need to specify at least one unit")

    # pour le confort
    def __repr__(self):
        return f"<{self.kelvin:g}K == {self.celsius:g}°C " \
               f"== {self.fahrenheit:g}°F>"

    def __str__(self):
        return f"{self.kelvin:g}K"


    # l'attribut 'kelvin' n'a pas de conversion à faire,
    # mais il vérifie que la valeur est positive
    def _get_kelvin(self):
        return self._kelvin

    def _set_kelvin(self, kelvin):
        if kelvin < 0:
            raise ValueError(f"Kelvin {kelvin} must be positive")
        self._kelvin = kelvin

    # la property qui définit l'attribut `kelvin`
    kelvin = property(_get_kelvin, _set_kelvin)


    # les deux autres properties font la conversion, puis
    # sous-traitent à la property kelvin pour le contrôle de borne
    def _set_celsius(self, celsius):
        # using .kelvin instead of ._kelvin to enforce
        self.kelvin = celsius + self.K

    def _get_celsius(self):
        return self._kelvin - self.K

    celsius = property(_get_celsius, _set_celsius)
```

```
    def _set_fahrenheit(self, fahrenheit):
        # using .kelvin instead of ._kelvin to enforce
        self.kelvin = (fahrenheit + self.KF) * self.RF

    def _get_fahrenheit(self):
        return self._kelvin / self.RF - self.KF

    fahrenheit = property(_get_fahrenheit, _set_fahrenheit)
```

Et voici ce qu'on peut en faire :

```
In [ ]: t = Temperature(celsius=0)
        t
```

```
In [ ]: t.fahrenheit
```

```
In [ ]: t.celsius += 100
        print(t1)
```

```
In [ ]: try:
            t = Temperature(fahrenheit = -1000)
        except Exception as e:
            print(f"OOPS, {type(e)}, {e}")
```

Pour en savoir plus Voir aussi la documentation officielle.

Vous pouvez notamment aussi, en option, ajouter un *deleter* pour intercepter les instructions du type :

```
In [ ]: # comme on n'a pas défini de deleter, on ne peut pas faire ceci
        try:
            del t.kelvin
        except Exception as e:
            print(f"OOPS {type(e)} {e}")
```

6.4 Un exemple de classes de la bibliothèque standard

Notez que ce complément, bien qu'un peu digressif par rapport au sujet principal qui est les classes et instances, a pour objectif de vous montrer l'intérêt de la programmation objet avec un module de la bibliothèque standard.

6.4.1 Complément - niveau basique

Le module `time`

Pour les accès à l'horloge, python fournit un module `time` - très ancien ; il s'agit d'une interface de très bas niveau avec l'OS, qui s'utilise comme ceci :

```
In [ ]: import time

        # on obtient l'heure courante sous la forme d'un flottant
        # qui représente le nombre de secondes depuis le 1er Janvier 1970
        t_now = time.time()
        t_now
```

```
In [ ]: # et pour calculer l'heure qu'il sera dans trois heures on fait
        t_later = t_now + 3 * 3600
```

Nous sommes donc ici clairement dans une approche non orientée objet ; on manipule des types de base, ici le type flottant :

```
In [ ]: type(t_later)
```

Et comme on le voit, les calculs se font sous une forme pas très lisible. Pour rendre ce nombre de secondes plus lisible, on utilise des conversions, pas vraiment explicites non plus ; voici par exemple un appel à `gmtime` qui convertit le flottant obtenu par la méthode `time()` en heure UTC (`gm` est pour Greenwich Meridian) :

```
In [ ]: struct_later = time.gmtime(t_later)
        print(struct_later)
```

Et on met en forme ce résultat en utilisant des méthodes comme, par exemple, `strftime()` pour afficher l'heure UTC dans 3 heures :

```
In [ ]: print(f'heure UTC dans trois heures {time.strftime("%Y-%m-%d at %H:%M", struct_later)}')
```

Le module datetime

Voyons à présent, par comparaison, comment ce genre de calculs se présente lorsqu'on utilise la programmation par objets.

Le module `datetime` expose un certain nombre de classes, que nous illustrons brièvement avec les classes `datetime` (qui modélise la date et l'heure d'un instant) et `timedelta` (qui modélise une durée).

La première remarque qu'on peut faire, c'est qu'avec le module `time` on manipulait un flottant pour représenter ces deux sortes d'objets (instant et durée) ; avec deux classes différentes notre code va être plus clair quant à ce qui est réellement représenté.

Le code ci-dessus s'écrirait alors, en utilisant le module `datetime` :

```
In [ ]: from datetime import datetime, timedelta

        dt_now = datetime.now()
        dt_later = dt_now + timedelta(hours=3)
```

Vous remarquez que c'est déjà un peu plus expressif.

Voyez aussi qu'on a déjà moins besoin de s'escrimer pour en avoir un aperçu lisible :

```
In [ ]: # on peut imprimer simplement un objet date_time
        print(f'maintenant {dt_now}')

In [ ]: # et si on veut un autre format, on peut toujours appeler strftime
        print(f'dans trois heures {dt_later.strftime("%Y-%m-%d at %H:%M")}')

In [ ]: # mais ce n'est même pas nécessaire, on peut passer le format directement
        print(f'dans trois heures {dt_later:%Y-%m-%d at %H:%M}')
```

Je vous renvoie à la documentation du module, et notamment ici, pour le détail des options de formatage disponibles.

Conclusion

Une partie des inconvénients du module `time` vient certainement du parti-pris de l'efficacité. De plus, c'est un module très ancien, mais auquel on ne peut guère toucher pour des raisons de compatibilité ascendante.

Par contre, le module `datetime`, tout en vous procurant un premier exemple de classes exposées par la bibliothèque standard, vous montre certains des avantages de la programmation orientée objet en général, et des classes de python en particulier.

Si vous devez manipuler des dates ou des heures, le module `datetime` constitue très certainement un bon candidat ; voyez la documentation complète du module pour plus de précisions sur ses possibilités.

6.4.2 Complément - niveau intermédiaire

Fuseaux horaires et temps local

Le temps nous manque pour traiter ce sujet dans toute sa profondeur.

En substance, c'est un sujet assez voisin de celui des accents, en ce sens que lors d'échanges d'informations de type *timestamp* entre deux ordinateurs, il faut échanger d'une part une valeur (l'heure et la date), et d'autre part le référentiel (s'agit-il de temps UTC, ou bien de l'heure dans un fuseau horaire, et si oui lequel).

La complexité est tout de même moindre que dans le cas des accents ; on s'en sort en général en convenant d'échanger systématiquement des heures UTC. Par contre, il existe une réelle diversité quant au format utilisé pour échanger ce type d'information, et cela reste une source d'erreurs assez fréquente.

6.4.3 Complément - niveau avancé

Classes et *marshalling*

Ceci nous procure une transition pour un sujet beaucoup plus général.

Nous avons évoqué en semaine 4 les formats comme JSON pour échanger les données entre applications, au travers de fichiers ou d'un réseau.

On a vu, par exemple, que JSON est un format "proche des langages" en ce sens qu'il est capable d'échanger des objets de base comme des listes ou des dictionnaires entre plusieurs langages comme, par exemple, JavaScript, python ou ruby. En XML, on a davantage de flexibilité puisqu'on peut définir une syntaxe sur les données échangées.

Mais il faut être bien lucide sur le fait que, aussi bien pour JSON que pour XML, il n'est **pas possible** d'échanger entre applications des **objets** en tant que tel. Ce que nous voulons dire, c'est que ces technologies de *marshalling* prennent bien en charge le *contenu* en termes de données, mais pas les informations de type, et *a fortiori* pas non plus le code qui appartient à la classe.

Il est important d'être conscient de cette limitation lorsqu'on fait des choix de conception, notamment lorsqu'on est amené à choisir entre classe et dictionnaire pour l'implémentation de telle ou telle abstraction.

Voyons cela sur un exemple inspiré de notre fichier de données liées au trafic maritime. En version simplifiée, un bateau est décrit par trois valeurs, son identité (`id`), son nom et son pays d'attachement.

Nous allons voir comment on peut échanger ces informations entre, disons, deux programmes dont l'un est en python, via un support réseau ou disque.

Si on choisit de simplement manipuler un dictionnaire standard :

```
In [ ]: bateau1 = {'name' : "Toccata", 'id' : 1000, 'country' : "France"}
```

alors on peut utiliser tels quels les mécanismes d'encodage et décodage de, disons, JSON. En effet c'est exactement ce genre d'informations que sait gérer la couche JSON.

Si au contraire on choisit de manipuler les données sous forme d'une classe on pourrait avoir envie d'écrire quelque chose comme ceci :

```
In [ ]: class Bateau:
            def __init__(self, id, name, country):
                self.id = id
                self.name = name
                self.country = country

        bateau2 = Bateau(1000, "Toccata", "FRA")
```

Maintenant, si vous avez besoin d'échanger cet objet avec le reste du monde, en utilisant par exemple JSON, tout ce que vous allez pouvoir faire passer par ce médium, c'est la valeur des trois champs, dans un dictionnaire. Vous pouvez facilement obtenir le dictionnaire en question pour le passer à la couche d'encodage :

```
In [ ]: vars(bateau2)
```

Mais à l'autre bout de la communication il va vous falloir : * déterminer d'une manière ou d'une autre que les données échangées sont en rapport avec la classe `Bateau` ; * construire vous même un objet de cette classe, par exemple avec un code comme :

```
In [ ]: # du coté du récepteur de la donnée
        class Bateau:
            def __init__(self, *args):
                if len(args) == 1 and isinstance(args[0], dict):
                    self.__dict__ = args[0]
                elif len(args) == 3:
                    id, name, country = args
                    self.id = id
                    self.name = name
                    self.country = country

        bateau3 = Bateau({'id': 1000, 'name': 'Leon', 'country': 'France'})
        bateau4 = Bateau(1001, 'Maluba', 'SUI' )
```

Conclusion

Pour reformuler ce dernier point, il n'y a pas en python l'équivalent de jmi (Java Metadata Interface) intégré à la distribution standard.

De plus on peut écrire du code en dehors des classes, et on n'est pas forcément obligé d'écrire une classe pour tout - à l'inverse ici encore de Java. Chaque situation doit être jugée dans son contexte naturellement, mais, de manière générale, la classe n'est pas la solution universelle ; il peut y avoir des mérites dans le fait de manipuler certaines données sous une forme allégée comme un type natif.

6.4.4 Complément - niveau intermédiaire

Souvenez-vous de ce qu'on avait dit en semaine 3 séquence 4, concernant les clés dans un dictionnaire ou les éléments dans un ensemble. Nous avions vu alors que, pour les types *built-in*, les clés devaient être des objets immuables et même globalement immuables.

Nous allons voir dans ce complément quelles sont les règles qui s'appliquent aux instances de classe, et notamment comment on peut manipuler des ensembles d'instances d'une manière qui fasse du sens.

Une instance de classe est presque toujours un objet mutable (voir à ce sujet un prochain complément sur les `namedtuples`).

Et pourtant, le langage vous permet d'insérer une instance dans un ensemble - ou de l'utiliser comme clé dans un dictionnaire.

Nous allons voir ce mécanisme en action, et mettre en évidence ses limites.

hachage par défaut : basé sur `id()`

```
In [ ]: # une classe Point qui ne redéfinit pas __eq__ ni __hash__
        class Point1:
            def __init__(self, x, y):
                self.x = x
                self.y = y

            def __repr__(self):
                return f"Pt[{self.x}, {self.y}]"
```

```
In [ ]: # deux instances
        p1 = Point1(2, 3)
        p2 = Point1(3, 4)

        # bien qu'ils soient mutables, on peut les mettre dans un ensemble
        s = {p1, p2}
```

Mais par contre soyez attentifs, car il faut savoir que pour la classe `Point1`, où nous n'avons rien redéfini, la fonction de hachage sur une instance de `Point1` ne dépend que de la valeur de `id()` sur cet objet.

Ce qui, dit autrement, signifie que deux objets qui sont distincts au sens de `id()` sont considérés comme différents, et donc peuvent coexister dans un ensemble, ou dans un dictionnaire, ce qui n'est pas forcément ce qu'on veut :

```
In [ ]: # un point similaire à p1
        p0 = Point1(2, 3)
        # nos deux objets se ressemblent
        p0, p1
```

```
In [ ]: # mais peuvent coexister dans un ensemble
        { p0, p1 }
```

`__hash__` et `__eq__`

Le protocole hashable permet de pallier cette déficience ; pour cela il nous faut définir deux méthodes :

— `__eq__` qui, sans grande surprise, va servir à évaluer `p == q` ;
— `__hash__` qui va retourner la clé de hachage sur un objet.

La subtilité étant bien entendu que ces deux méthodes doivent être cohérentes, si deux objets sont égaux, il faut que leurs hashs soient égaux ; de bon sens, si l'égalité se base sur nos deux attributs `x` et `y`, il faudra bien entendu que la fonction de hachage utilise elle aussi ces deux attributs. Voir la documentation de `__hash__`.

Voyons cela sur une sous-classe de `Point1`, dans laquelle nous définissons ces deux méthodes :

```
In [ ]: class Point2(Point1):

            # l'égalité va se baser naturellement sur x et y
            def __eq__(self, other):
                return self.x == other.x and self.y == other.y

            # du coup la fonction de hachage
            # dépend aussi de x et de y
            def __hash__(self):
                return (11 * self.x + self.y) // 16
```

On peut vérifier que cette fois les choses fonctionnent correctement :

```
In [ ]: q0 = Point2(2, 3)
        q1 = Point2(2, 3)
```

nos deux objets sont distincts pour is() mais égaux pour == :

```
In [ ]: print(f"is → {q0 is q1} \n== → {q0 == q1}")
```

et un ensemble contenant les deux points n'en contient qu'un :

```
In [ ]: {q0, q1}
```

```
In [ ]: # et bien sûr c'est pareil pour un dictionnaire
        d = {}
        d[q0] = 1
        # les deux clés q0 et q1 sont les mêmes pour le dictionnaire
        # du coupe ici on écrase la (seule) valeur dans d
        d[q1] = 10000
        d
```

6.5 Surcharge d'opérateurs (1)

6.5.1 Complément - niveau intermédiaire

Ce complément vise à illustrer certaines des possibilités de surcharge d'opérateurs, ou plus généralement les mécanismes disponibles pour étendre le langage et donner un sens à des fragments de code comme : * `objet1 + objet2` * `item in objet` * `objet[key]` * `objet.key` * `for i in objet:` * `if objet:` * `objet(arg1, arg2)` (et non pas `classe(arg1, arg2)`) * etc..

que jusqu'ici, sauf pour la boucle `for` et pour le hachage, on n'a expliqué que pour des objets de type prédéfini.

Le mécanisme général pour cela consiste à définir des **méthodes spéciales**, avec un nom en `__nom__`. Il existe un total de près de 80 méthodes dans ce système de surcharges, aussi il n'est pas question ici d'être exhaustif. Vous trouverez dans ce document une liste complète de ces possibilités.

Il nous faut également signaler que les mécanismes mis en jeu ici sont **de difficultés assez variables**. Dans le cas le plus simple il suffit de définir une méthode sur la classe pour obtenir le résultat (par exemple, définir `__call__` pour rendre un objet callable). Mais parfois on parle d'un ensemble de méthodes qui doivent être cohérentes, voyez par exemple les descriptors qui

mettent en jeu les méthodes `__get__`, `__set__` et `__delete__`, et qui peuvent sembler particulièrement cryptiques. On aura d'ailleurs l'occasion d'approfondir les descriptors en semaine 9 avec les sujets avancés.

Nous vous conseillons de commencer par des choses simples, et surtout de n'utiliser ces techniques que lorsqu'elles apportent vraiment quelque chose. Le constructeur et l'affichage sont pratiquement toujours définis, mais pour tout le reste il convient d'utiliser ces traits avec le plus grand discernement. Dans tous les cas écrivez votre code avec la documentation sous les yeux, c'est plus prudent :)

Nous avons essayé de présenter cette sélection par difficulté croissante. Par ailleurs, et pour alléger la présentation, cet exposé a été coupé en trois notebooks différents.

Rappels

Pour rappel, on a vu dans la vidéo : * la méthode `__init__` pour définir un **constructeur** ; * la méthode `__str__` pour définir comment une instance s'imprime avec `print`.

Affichage : `__repr__` et `__str__`

Nous commençons par signaler la méthode `__repr__` qui est assez voisine de `__str__`, et qui donc doit retourner un objet de type chaîne de caractères, sauf que : * `__str__` est utilisée par `print` (affichage orienté utilisateur du programme, priorité au confort visuel) ; * alors que `__repr__` est utilisée par la fonction `repr()` (affichage orienté programmeur, aussi peu ambigu que possible) ; * enfin il faut savoir que `__repr__` est utilisée **aussi** par `print` si `__str__` n'est pas définie.

Pour cette dernière raison, on trouve dans la nature `__repr__` plutôt plus souvent que `__str__` ; voyez ce lien pour davantage de détails.

Quand est utilisée `repr()` ? La fonction `repr()` est utilisée massivement dans les informations de debugging comme les traces de pile lorsqu'une exception est levée. Elle est aussi utilisée lorsque vous affichez un objet sans passer par `print`, c'est-à-dire par exemple :

```
In [ ]: class Foo:
            def __repr__(self):
                return 'custom repr'

        foo = Foo()
        # lorsque vous affichez un objet comme ceci
        foo
        # en fait vous utilisez repr()
```

Deux exemples Voici deux exemples simples de classes ; dans le premier on n'a défini que `__repr__`, dans le second on a redéfini les deux méthodes :

```
In [ ]: # une classe qui ne définit que __repr__
        class Point:
            "première version de Point - on ne définit que __repr__"
            def __init__(self, x, y):
                self.x = x
                self.y = y
            def __repr__(self):
```

```
                return f"Point({self.x},{self.y})"

        point = Point (0,100)

        print("avec print", point)

        # si vous affichez un objet sans passer par print
        # vous utilisez repr()
        point
```

```
In [ ]: # la même chose mais où on redéfinit __str__ et __repr__
        class Point2:
            "seconde version de Point - on définit __repr__ et __str__"
            def __init__(self, x, y):
                self.x = x
                self.y = y
            def __repr__(self):
                return f"Point2({self.x},{self.y})"
            def __str__(self):
                return f"({self.x},{self.y})"

        point2 = Point2 (0,100)

        print("avec print", point2)

        # les f-strings (ou format) utilisent aussi __str__
        print(f"avec format {point2}")

        # et si enfin vous affichez un objet sans passer par print
        # vous utilisez repr()
        point2
```

`__bool__`

Vous vous souvenez que la condition d'un test dans un `if` peut ne pas retourner un booléen (nous avons vu cela en Semaine 4, Séquence "Test if/elif/else et opérateurs booléens"). Nous avions noté que pour les types prédéfinis, sont considérés comme *faux* les objets : `None`, la liste vide, un tuple vide, etc.

Avec `__bool__` on peut redéfinir le comportement des objets d'une classe vis-à-vis des conditions - ou si l'on préfère, quel doit être le résultat de `bool(instance)`.

Attention pour éviter les comportements imprévus, comme on est en train de redéfinir le comportement des conditions, il **faut** renvoyer un **booléen** (ou à la rigueur 0 ou 1), on ne peut pas dans ce contexte retourner d'autres types d'objet.

Nous allons **illustrer** cette méthode dans un petit moment avec une nouvelle implémentation de la classe `Matrix2`.

Remarquez enfin qu'en l'absence de méthode `__bool__`, on cherche aussi la méthode `__len__` pour déterminer le résultat du test ; une instance de longueur nulle est alors considéré comme `False`, en cohérence avec ce qui se passe avec les types *built-in* `list`, `dict`, `tuple`, etc.

Ce genre de *protocole*, qui cherche d'abord une méthode (`__bool__`), puis une autre (`__len__`) en cas d'absence de la première, est relativement fréquent dans la mécanique de surcharge des opérateurs ; c'est entre autres pourquoi la documentation est indispensable lorsqu'on surcharge les opérateurs.

`__add__` et apparentés (`__mul__`, `__sub__`, `__div__`, `__and__`, etc.)

On peut également redéfinir les opérateurs arithmétiques et logiques. Dans l'exemple qui suit, nous allons l'illustrer sur l'addition de matrices. On rappelle pour mémoire que :

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} \\ a_{21}+b_{21} & a_{22}+b_{22} \end{pmatrix}$$

Une nouvelle version de la classe `Matrix2`

Voici (encore) une nouvelle implémentation de la classe de matrices 2x2, qui illustre cette fois : * la possibilité d'ajouter deux matrices ; * la possibilité de faire un test sur une matrice - le test sera faux si la matrice a tous ses coefficients nuls ; * et, bien que ce ne soit pas le sujet immédiat, cette implémentation illustre aussi la possibilité de construire la matrice à partir : * soit des 4 coefficients, comme par exemple : `Matrix2(a, b, c, d)` * soit d'une séquence, comme par exemple : `Matrix2(range(4))`

Cette dernière possibilité va nous permettre de simplifier le code de l'addition, comme on va le voir.

```
In [ ]: # notre classe Matrix2 avec encore une autre implémentation
        class Matrix2:

            def __init__(self, *args):
                """
                le constructeur accepte
                (*) soit les 4 coefficients individuellement
                (*) soit une liste - ou + généralement une séquence - des mêmes
                """
                # on veut pouvoir créer l'objet à partir des 4 coefficients
                # souvenez-vous qu'avec la forme *args, args est toujours un tuple
                if len(args) == 4:
                    self.coefs = args
                # ou bien d'une séquence de 4 coefficients
                elif len(args) == 1:
                    self.coefs = tuple(*args)

            def __repr__(self):
                "l'affichage"
                return "[" + ", ".join([str(c) for c in self.coefs]) + "]"

            def __add__(self, other):
                """
                l'addition de deux matrices retourne un nouvel objet
                la possibilité de créer une matrice à partir
                d'une liste rend ce code beaucoup plus facile a écrire
                """
                return Matrix2([a + b for a, b in zip(self.coefs, other.coefs)])
```

```
        def __bool__(self):
            """
            on considère que la matrice est non nulle
            si un au moins de ses coefficients est non nul
            """
            # ATTENTION le retour doit être un booléen
            # ou à la rigueur 0 ou 1
            for c in self.coefs:
                if c:
                    return True
            return False
```

On peut à présent créer deux objets, les ajouter, et vérifier que la matrice nulle se comporte bien comme attendu :

```
In [ ]: zero     = Matrix2 ([0,0,0,0])

        matrice1 = Matrix2 (1,2,3,4)
        matrice2 = Matrix2 (list(range(10,50,10)))

        print('avant matrice1', matrice1)
        print('avant matrice2', matrice2)

        print('somme', matrice1 + matrice2)

        print('après matrice1', matrice1)
        print('après matrice2', matrice2)

        if matrice1:
            print(matrice1,"n'est pas nulle")
        if not zero:
            print(zero,"est nulle")
```

Voici en vrac quelques commentaires sur cet exemple.

Utiliser un tuple Avant de parler de la surcharge des opérateurs *per se*, vous remarquerez que l'on range les coefficients dans un **tuple**, de façon à ce que notre objet `Matrix2` soit indépendant de l'objet qu'on a utilisé pour le créer (et qui peut être ensuite modifié par l'appelant).

Créer un nouvel objet Vous remarquez que l'addition `__add__` renvoie un **nouvel objet**, au lieu de modifier `self` en place. C'est la bonne façon de procéder tout simplement parce que lorsqu'on écrit :

```
print('somme', matrice1 + matrice2)
```

on ne s'attend pas du tout à ce que `matrice1` soit modifiée après cet appel.

Du code qui ne dépend que des 4 opérations Le fait d'avoir défini l'addition nous permet par exemple de bénéficier de la fonction *builtin* `sum`. En effet le code de `sum` fait lui-même des additions, il n'y a donc aucune raison de ne pas pouvoir l'exécuter avec en entrée un liste de matrices puisque maintenant on sait les additionner, (mais on a dû toutefois passer à `sum` comme élément neutre `zero`) :

```
In [ ]: sum([matrice1, matrice2, matrice1] , zero)
```

C'est un effet de bord du typage dynamique. On ne vérifie pas *a priori* que tous les arguments passés à `sum` savent faire une addition ; *a contrario*, si ils savent s'additionner on peut exécuter le code de `sum`.

De manière plus générale, si vous écrivez par exemple un morceau de code qui travaille sur les éléments d'un anneau (au sens anneau des entiers $\mathbb{Z}$) - imaginez un code qui factorise des polynômes - vous pouvez espérer utiliser ce code avec n'importe quel anneau, c'est à dire avec une classe qui implémente les 4 opérations (pourvu bien sûr que cet ensemble soit effectivement un anneau).

On peut aussi redéfinir un ordre La place nous manque pour illustrer la possibilité, avec les opérateurs `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, et `__ge__`, de redéfinir un ordre sur les instances d'une classe.

Signalons à cet égard qu'il existe un mécanisme "intelligent" qui permet de définir un ordre à partir d'un sous-ensemble seulement de ces méthodes, l'idée étant que si vous savez faire > et =, vous savez sûrement faire tout le reste. Ce mécanisme est documenté ici ; il repose sur **un décorateur** (`@total_ordering`), un mécanisme que nous étudierons en semaine 9, mais que vous pouvez utiliser dès à présent.

De manière analogue à `sum` qui fonctionne sur une liste de matrices, si on avait défini un ordre sur les matrices, on aurait pu alors utiliser les *built-in* `min` et `max` pour calculer une borne supérieure ou inférieure dans une séquence de matrices.

6.5.2 Complément - niveau avancé

Le produit avec un scalaire On implémenterait la multiplication de deux matrices d'une façon identique (quoique plus fastidieuse naturellement).

La multiplication d'une matrice par un scalaire (un réel ou complexe pour fixer les idées), comme ici :

```
matrice2 = reel * matrice1
```

peut être également réalisée par surcharge de l'opérateur `__rmul__`.

Il s'agit d'une astuce, destinée précisément à ce genre de situations, où on veut étendre la classe de l'opérande de **droite**, sachant que dans ce cas précis l'opérande de gauche est un type de base, qu'on ne peut pas étendre (les classes *built-in* sont non mutables, pour garantir la stabilité de l'interpréteur).

Voici donc comment on s'y prendrait. Pour éviter de reproduire tout le code de la classe, on va l'étendre à la volée.

```
In [ ]: # remarquez que les opérandes sont apparemment inversés
        # dans le sens où pour evaluer
        #     reel * matrice
        # on écrit une méthode qui prend en argument
        #   la matrice, puis le réel
        # mais n'oubliez pas qu'on est en fait en train
        # d'écrire une méthode sur la classe `Matrix2`
        def multiplication_scalaire(self, alpha):
            return Matrix2([alpha * coef for coef in self.coefs])

        # on ajoute la méthode spéciale __rmul__
        Matrix2.__rmul__ = multiplication_scalaire
```

```
In [ ]: matrice1
```

```
In [ ]: 12 * matrice1
```

6.6 Méthodes spéciales (2/3)

6.6.1 Complément - niveau avancé

Nous poursuivons dans ce complément la sélection de méthodes spéciales entreprise en première partie.

`__contains__`, `__len__`, `__getitem__` et apparentés

La méthode `__contains__` permet de donner un sens à :

```
item in objet
```

Sans grande surprise, elle prend en argument un objet et un item, et doit renvoyer un booléen. Nous l'illustrons ci-dessous avec la classe `DualQueue`.

La méthode `__len__` est utilisée par la fonction *built-in* `len` pour retourner la longueur d'un objet.

La classe `DualQueue` Nous allons illustrer ceci avec un exemple de classe, un peu artificiel, qui implémente une queue de type FIFO. Les objets sont d'abord admis dans la file d'entrée (`add_input`), puis déplacés dans la file de sortie (`move_input_to_output`), et enfin sortis (`emit_output`).

Clairement, cet exemple est à but uniquement pédagogique ; on veut montrer comment une implémentation qui repose sur deux listes séparées peut donner l'illusion d'une continuité, et se présenter comme un container unique. De plus cette implémentation ne fait aucun contrôle pour ne pas obscurcir le code.

```
In [ ]: class DualQueue:
            """Une double file d'attente FIFO"""

            def __init__(self):
                "constructeur, sans argument"
                self.inputs = []
                self.outputs = []

            def __repr__ (self):
                "affichage"
                return f"<DualQueue, inputs={self.inputs}, outputs={self.outputs}>"

            # la partie qui nous intéresse ici
            def __contains__(self, item):
                "appartenance d'un objet à la queue"
                return item in self.inputs or item in self.outputs

            def __len__(self):
                "longueur de la queue"
```

```
        return len(self.inputs) + len(self.outputs)

    # l'interface publique de la classe
    # le plus simple possible et sans aucun contrôle
    def add_input(self, item):
        "faire entrer un objet dans la queue d'entrée"
        self.inputs.insert(0, item)

    def move_input_to_output (self):
        "l'objet le plus ancien de la queue d'entrée est promu dans la queue de sorti
        self.outputs.insert(0, self.inputs.pop())

    def emit_output (self):
        "l'objet le plus ancien de la queue de sortie est émis"
        return self.outputs.pop()
```

```
In [ ]: # on construit une instance pour nos essais
        queue = DualQueue()
        queue.add_input('zero')
        queue.add_input('un')
        queue.move_input_to_output()
        queue.move_input_to_output()
        queue.add_input('deux')
        queue.add_input('trois')

        print(queue)
```

Longueur et appartenance Avec cette première version de la classe `DualQueue` on peut utiliser `len` et le test d'appartenance :

```
In [ ]: print(f'len() = {len(queue)}')

        print(f"deux appartient-il ? {'deux' in queue}")
        print(f"1 appartient-il ? {1 in queue}")
```

Accès séquentiel (accès par un index entier) Lorsqu'on a la notion de longueur de l'objet avec `__len__`, il peut être opportun - quoique cela n'est pas imposé par le langage, comme on vient de le voir - de proposer également un accès indexé par un entier pour pouvoir faire :

```
queue[1]
```

Pour ne pas répéter tout le code de la classe, nous allons étendre `DualQueue` ; pour cela nous définissons une fonction, que nous affectons ensuite à `DualQueue.__getitem__`, comme nous avons déjà eu l'occasion de le faire :

```
In [ ]: # une première version de DualQueue.__getitem__
        # pour uniquement l'accès par index

        # on définit une fonction
        def dual_queue_getitem (self, index):
            "redéfinit l'accès [] séquentiel"
```

```
            # on vérifie que l'index a un sens
            if not (0 <= index < len(self)):
                raise IndexError(f"Mauvais indice {index} pour DualQueue")
            # on décide que l'index 0 correspond à l'élément le plus ancien
            # ce qui oblige à une petite gymnastique
            li = len(self.inputs)
            lo = len(self.outputs)
            if index < lo:
                return self.outputs[lo - index - 1]
            else:
                return self.inputs[li - (index-lo) - 1]

        # et on affecte cette fonction à l'intérieur de la classe
        DualQueue.__getitem__ = dual_queue_getitem
```

À présent, on peut **accéder** aux objets de la queue **séquentiellement** :

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

ce qui lève la même exception qu'avec une vraie liste si on utilise un mauvais index :

```
In [ ]: try:
            print(queue[5])
        except IndexError as e:
            print('ERREUR',e)
```

Amélioration : accès par slice Si on veut aussi supporter l'accès par slice comme ceci :

```
queue[1:3]
```

il nous faut modifier la méthode `__getitem__`.

Le second argument de `__getitem__` correspond naturellement au contenu des crochets `[]`, on utilise donc `isinstance` pour écrire un code qui s'adapte au type d'indexation, comme ceci :

```
In [ ]: # une deuxième version de DualQueue.__getitem__
        # pour l'accès par index et/ou par slice

        def dual_queue_getitem (self, key):
            "redéfinit l'accès par [] pour entiers, slices, et autres"

            # l'accès par slice queue[1:3]
            # nous donne pour key un objet de type slice
            if isinstance(key, slice):
                # key.indices donne les indices qui vont bien
                return [self[index] for index in range(*key.indices(len(self)))]

            # queue[3] nous donne pour key un entier
            elif isinstance(key, int):
                index = key
                # on vérifie que l'index a un sens
                if index < 0 or index >= len(self):
```

```
                raise IndexError(f"Mauvais indice {index} pour DualQueue")
            # on décide que l'index 0 correspond à l'élément le plus ancien
            # ce qui oblige à une petite gymnastique
            li = len(self.inputs)
            lo = len(self.outputs)
            if index < lo:
                return self.outputs[lo-index-1]
            else:
                return self.inputs[li-(index-lo)-1]
        # queue ['foo'] n'a pas de sens pour nous
        else:
            raise KeyError(f"[] avec type non reconnu {key}")

    # et on affecte cette fonction à l'intérieur de la classe
    DualQueue.__getitem__ = dual_queue_getitem
```

Maintenant on peut accéder par slice :

```
In [ ]: queue[1:3]
```

Et on reçoit bien une exception si on essaie d'accéder par clé :

```
In [ ]: try:
            queue['key']
        except KeyError as e:
            print(f"OOPS: {type(e).__name__}: {e}")
```

L'objet est itérable (même sans avoir `__iter__`) Avec seulement `__getitem__`, on peut **faire une boucle** sur l'objet queue. On l'a mentionné rapidement dans la séquence sur les itérateurs, mais la **méthode `__iter__` n'est pas la seule façon** de rendre un objet itérable :

```
In [ ]: # grâce à __getitem__ on a rendu les
        # objets de type DualQueue itérables
        for item in queue:
           print(item)
```

On peut faire un test sur l'objet De manière similaire, même sans la méthode `__bool__`, cette classe sait **faire des tests de manière correcte** grâce uniquement à la méthode `__len__` :

```
In [ ]: # un test fait directement sur la queue
        if queue:
            print(f"La queue {queue} est considérée comme True")

In [ ]: # le même test sur une queue vide
        empty = DualQueue()

        # maintenant le test est négatif (notez bien le *not* ici)
        if not empty:
            print(f"La queue {empty} est considérée comme False")
```

`__call__` et les *callables*

Le langage introduit de manière similaire la notion de ***callable*** - littéralement, qui peut être appelé. L'idée est très simple, on cherche à donner un sens à un fragment de code du genre de :

```
# on crée une instance
objet = Classe(arguments)
```

et c'est l'objet (Attention : **l'objet, pas la classe**) qu'on utilise comme une fonction

```
objet(arg1, arg2)
```

Le protocole ici est très simple ; cette dernière ligne a un sens en python dès lors que : * `objet` possède une méthode `__call__` ; * et que celle-ci peut être envoyée à `objet` avec les arguments `arg1, arg2` ; * et c'est ce résultat qui sera alors retourné par `objet(arg1, arg2)` :

```
objet(arg1, arg2) ⟺ objet.__call__(arg1, arg2)
```

Voyons cela sur un exemple :

```
In [ ]: class PlusClosure:
            """Une classe callable qui permet de faire un peu comme la
            fonction built-in sum mais en ajoutant une valeur initiale"""
            def __init__(self, initial):
                self.initial = initial
            def __call__(self, *args):
                return self.initial + sum(args)

        # on crée une instance avec une valeur initiale 2 pour la somme
        plus2 = PlusClosure (2)

In [ ]: # on peut maintenant utiliser cet objet
        # comme une fonction qui fait sum(*arg)+2
        plus2()

In [ ]: plus2(1)

In [ ]: plus2(1, 2)
```

Pour ceux qui connaissent, nous avons choisi à dessein un exemple qui s'apparente à une clôture. Nous reviendrons sur cette notion de *callable* lorsque nous verrons les décorateurs en semaine 9.

6.7 Méthodes spéciales (3/3)

6.7.1 Complément - niveau avancé

Ce complément termine la série sur les méthodes spéciales.

`__getattr__` et apparentés

Dans cette dernière partie nous allons voir comment avec la méthode `__getattr__`, on peut redéfinir la façon que le langage a d'évaluer :

```
objet.attribut
```

Avertissement : on a vu dans la séquence consacrée à l'héritage que, pour l'essentiel, le mécanisme d'héritage repose **précisément** sur la façon d'évaluer les attributs d'un objet, aussi nous vous recommandons d'utiliser ce trait avec précaution, car il vous donne la possibilité de "faire muter le langage" comme on dit.

Remarque : on verra en toute dernière semaine que `__getattr__` est *une* façon d'agir sur la façon dont le langage opère les accès aux attributs. Sachez qu'en réalité, le protocole d'accès aux attributs peut être modifié beaucoup plus profondément si nécessaire.

Un exemple : la classe `RPCProxy` Pour illustrer `__getattr__`, nous allons considérer le problème suivant. Une application utilise un service distant, avec laquelle elle interagit au travers d'une API.

C'est une situation très fréquente : lorsqu'on utilise un service météo, ou de géolocalisation, ou de réservation, le prestataire vous propose une **API** (Application Programming Interface) qui se présente bien souvent comme une **liste de fonctions**, que votre fonction peut appeler à distance au travers d'un mécanisme de **RPC** (Remote Procedure Call).

Imaginez pour fixer les idées que vous utilisez un service de réservation de ressources dans un Cloud, qui vous permet d'appeler les fonctions suivantes : * `GetNodes`(...) pour obtenir des informations sur les noeuds disponibles ; * `BookNode`(...) pour réserver un noeud ; * `ReleaseNode`(...) pour abandonner un noeud.

Naturellement ceci est une API extrêmement simplifiée. Le point que nous voulons illustrer ici est que le dialogue avec le service distant : * requiert ses propres données - comme l'URL où on peut joindre le service, et les identifiants à utiliser pour s'authentifier ; * et possède sa propre logique - dans le cas d'une authentification par session par exemple, il faut s'authentifier une première fois avec un login/password, pour obtenir une session qu'on peut utiliser dans les appels suivants.

Pour ces raisons il est naturel de concevoir une classe `RPCProxy` dans laquelle on va rassembler à la fois ces données et cette logique, pour soulager toute l'application de ces détails, comme on l'a illustré ci-dessous :

Pour implémenter la plomberie liée à RPC, à l'encodage et décodage des données, et qui sera interne à la classe `RPCProxy`, on pourra en vraie grandeur utiliser des outils comme : * `xmlrpc.client` qui fait partie de la bibliothèque standard ; * ou, pour JSON, une des nombreuses implémentations qu'un moteur de recherche vous exposera si vous cherchez `python rpc json`, comme par exemple `json-rpc`.

Cela n'est toutefois pas notre sujet ici, et nous nous contenterons, dans notre code simplifié, d'imprimer un message.

Une approche naïve Se pose donc la question de savoir quelle interface la classe `RPCProxy` doit offrir au reste du monde. Dans une première version naïve on pourrait écrire quelque chose comme :

```
In [ ]: # la version naïve de la classe RPCProxy

        class RPCProxy:

            def __init__(self, url, login, password):
```

```
            self.url = url
            self.login = login
            self.password = password

        def _forward_call(self, functionname, *args):
            """
            helper method that marshalls and forwards
            the function and arguments to the remote end
            """
            print(f"""Envoi à {self.url}
    de la fonction {functionname} -- args= {args}""")
            return "retour de la fonction " + functionname

        def GetNodes (self, *args):
            return self._forward_call ('GetNodes', *args)
        def BookNode (self, *args):
            return self._forward_call ('BookNode', *args)
        def ReleaseNode (self, *args):
            return self._forward_call ('ReleaseNode', *args)
```

Ainsi l'application utilise la classe de cette façon :

```
In [ ]: # création d'une instance de RPCProxy

    rpc_proxy = RPCProxy(url='http://cloud.provider.com/JSONAPI',
                         login='dupont',
                         password='***')

    # cette partie du code, en tant qu'utilisateur de l'API,
    # est supposée connaître les détails
    # des arguments à passer
    # et de comment utiliser les valeurs de retour
    nodes_list = rpc_proxy.GetNodes (
        [ ('phy_mem', '>=', '32G') ] )

    # reserver un noeud
    node_lease = rpc_proxy.BookNode (
        { 'id' : 1002, 'phy_mem' : '32G' } )
```

Discussion Quelques commentaires en vrac au sujet de cette approche :

— l'interface est correcte ; l'objet `rcp_proxy` se comporte bien comme un proxy, on a donné au programmeur l'illusion complète qu'il utilise une classe locale (sauf pour les performances bien entendu...) ;
— la séparation des rôles est raisonnable également, la classe RPCProxy n'a pas à connaître le détail de la signature de chaque méthode, charge à l'appelant d'utiliser l'API correctement ;
— par contre ce qui cloche, c'est que l'implémentation de la classe RPCProxy dépend de la liste des fonctions exposées par l'API ; imaginez une API avec 100 ou 200 méthodes, cela donne une dépendance assez forte et surtout inutile ;
— enfin, nous avons escamoté la nécessité de faire de RPCProxy un singleton, mais c'est une toute autre histoire.

Une approche plus subtile Pour obtenir une implémentation qui conserve toutes les qualités de la version naïve, mais sans la nécessité de définir une à une toutes les fonctions de l'API, on peut tirer profit de `__getattr__`, comme dans cette deuxième version :

```
In [ ]: # une deuxième implémentation de RPCProxy

        class RPCProxy:

            def __init__(self, url, login, password):
                self.url = url
                self.login = login
                self.password = password

            def __getattr__(self, function):
                """
                Crée à la volée une méthode sur RPCProxy qui correspond
                à la fonction distante 'function'
                """
                def forwarder(*args):
                    print(f"Envoi à {self.url}\nde la fonction {function} -- args= {args}")
                    return "retour de la fonction " + function
                return forwarder
```

Qui est cette fois **totalement découplée** des détails de l'API, et qu'on peut utiliser exactement comme tout à l'heure :

```
In [ ]: # création d'une instance de RPCProxy

        rpc_proxy = RPCProxy (url='http://cloud.provider.com/JSONAPI',
                              login='dupont',
                              password='***')

        # cette partie du code, en tant qu'utilisateur de l'API,
        # est supposée connaître les détails
        # des arguments à passer
        # et de comment utiliser les valeurs de retour
        nodes_list = rpc_proxy.GetNodes (
            [ ('phy_mem', '>=', '32G') ] )

        # reserver un noeud
        node_lease = rpc_proxy.BookNode (
            { 'id' : 1002, 'phy_mem' : '32G' } )
```

6.8 Héritage

6.8.1 Complément - niveau basique

La notion d'héritage, qui fait partie intégrante de la Programmation Orientée Objet, permet principalement de maximiser la **réutilisabilité**.

Nous avons vu dans la vidéo les mécanismes d'héritage *in abstracto*. Pour résumer très brièvement, on recherche un attribut (pour notre propos, disons une méthode) à partir d'une

instance en cherchant : * d'abord dans l'instance elle-même ; * puis dans la classe de l'instance ; * puis dans les super-classes de la classe.

L'objet de ce complément est de vous donner, d'un point de vue plus appliqué, des idées de ce que l'on peut faire ou non avec ce mécanisme. Le sujet étant assez rabâché par ailleurs, nous nous concentrerons sur deux points :

— les pratiques de base utilisées pour la conception de classes, et notamment pour bien distinguer **héritage** et **composition** ;
— nous verrons ensuite, sur des **exemples extraits de code réel**, comment ces mécanismes permettent en effet de contribuer à la réutilisabilité du code.

Plusieurs formes d'héritage

Une méthode héritée peut être rangée dans une de ces trois catégories : * *implicite* : si la classe fille ne définit pas du tout la méthode ; * *redéfinie* : si on récrit la méthode entièrement ; * *modifiée* : si on récrit la méthode dans la classe fille, mais en utilisant le code de la classe mère.

Commençons par illustrer tout ceci sur un petit exemple :

```
In [ ]: # Une classe mère
        class Fleur:
            def implicite(self):
                print('Fleur.implicite')
            def redefinie(self):
                print('Fleur.redéfinie')
            def modifiee(self):
                print('Fleur.modifiée')

        # Une classe fille
        class Rose(Fleur):
            # on ne définit pas implicite
            # on rédéfinit complement redefinie
            def redefinie(self):
                print('Rose.redefinie')
            # on change un peu le comportement de modifiee
            def modifiee(self):
                Fleur.modifiee(self)
                print('Rose.modifiee apres Fleur')
```

On peut à présent créer une instance de Rose et appeler sur cette instance les trois méthodes.

```
In [ ]: # fille est une instance de Rose
        fille = Rose()

        fille.implicite()
```

```
In [ ]: fille.redefinie()
```

S'agissant des deux premières méthodes, le comportement qu'on observe est simplement la conséquence de l'algorithme de recherche d'attributs : `implicite` est trouvée dans la classe Fleur et `redefinie` est trouvée dans la classe Rose.

```
In [ ]: fille.modifiee()
```

Pour la troisième méthode, attardons nous un peu car on voit ici comment *combiner* facilement le code de la classe mère avec du code spécifique à la classe fille, en appelant explicitement la méthode de la classe mère lorsqu'on fait :

```
Fleur.modifiee(self)
```

La fonction *builtin* `super()` Signalons à ce sujet, pour être exhaustif, l'existence de la fonction *built-in* `super()` qui permet de réaliser la même chose sans nommer explicitement la classe mère, comme on le fait ici :

```
In [ ]: # Une version allégée de la classe fille, qui utilise super()
        class Rose(Fleur):
            def modifiee(self):
                super().modifiee()
                print('Rose.modifiee apres Fleur')

In [ ]: fille = Rose()

        fille.modifiee()
```

On peut envisager d'utiliser `super()` dans du code très abstrait où on ne sait pas déterminer statiquement le nom de la classe dont il est question. Mais, c'est une question de goût évidemment, je recommande personnellement la première forme, où on qualifie la méthode avec le nom de la classe mère qu'on souhaite utiliser. En effet, assez souvent :

— on sait déterminer le nom de la classe dont il est question ;
— ou alors on veut mélanger plusieurs méthodes héritées (via l'héritage multiple, dont on va parler dans un prochain complément) et dans ce cas `super()` ne peut rien pour nous.

Héritage *vs* Composition

Dans le domaine de la conception orientée objet, on fait la différence entre deux constructions, l'héritage et la composition, qui à une analyse superficielle peuvent paraître procurer des résultats similaires, mais qu'il est important de bien distinguer.

Voyons d'abord en quoi consiste la composition et pourquoi le résultat est voisin :

```
In [ ]: # Une classe avec qui on n'aura pas de relation d'héritage
        class Tige:
            def implicite(self):
                print('Tige.implicite')
            def redefinie(self):
                print('Tige.redefinie')
            def modifiee(self):
                print('Tige.modifiee')

        # on n'hérite pas
        # mais on fait ce qu'on appelle une composition
        # avec la classe Tige
        class Rose:
            # mais pour chaque objet de la classe Rose
            # on va créer un objet de la classe Tige
            # et le conserver dans un champ
            def __init__(self):
```

```
            self.externe = Tige()

        # le reste est presque comme tout à l'heure
        # sauf qu'il faut definir implicite
        def implicite(self):
            self.externe.implicite()

        # on redefinit complement redefinie
        def redefinie(self):
            print('Rose.redefinie')

        def modifiee(self):
            self.externe.modifiee()
            print('Rose.modifiee apres Tige')
```

```
In [ ]: # on obtient ici exactement le même comportement pour les trois sortes de méthodes
        fille = Rose()

        fille.implicite()
        fille.redefinie()
        fille.modifiee()
```

Comment choisir ? Alors, quand faut-il utiliser l'héritage et quand faut-il utiliser la composition ?
On arrive ici à la limite de notre cours, il s'agit plus de conception que de codage à proprement parler, mais pour donner une réponse très courte à cette question :

— on utilise l'héritage lorsqu'un objet de la sous-classe **est aussi un** objet de la super-classe (une rose est aussi une fleur) ;
— on utilise la composition lorsqu'un objet de la sous-classe **a une relation avec** un objet de la super-classe (une rose possède une tige, mais c'est un autre objet).

6.8.2 Complément - niveau intermédiaire

Des exemples de code

Sans transition, dans cette section un peu plus prospective, nous vous avons signalé quelques morceaux de code de la bibliothèque standard qui utilisent l'héritage. Sans aller nécessairement jusqu'à la lecture de ces codes, il nous a semblé intéressant de commenter spécifiquement l'usage qui est fait de l'héritage dans ces bibliothèques.

Le module `calendar` On trouve dans la bibliothèque standard le module `calendar`. Ce module expose deux classes `TextCalendar` et `HTMLCalendar`. Sans entrer du tout dans le détail, ces deux classes permettent d'imprimer dans des formats différents le même type d'informations.

Le point ici est que les concepteurs ont choisi un graphe d'héritage comme ceci :

```
Calendar
|-- TextCalendar
|-- HTMLCalendar
```

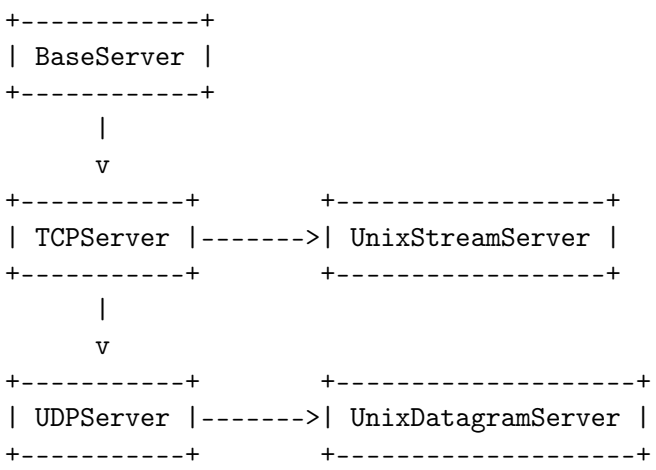
qui permet de grouper le code concernant la logique dans la classe `Calendar`, puis dans les deux sous-classes d'implémenter le type de sortie qui va bien.

C'est l'utilisateur qui choisit la classe qui lui convient le mieux, et de cette manière, le maximum de code est partagé entre les deux classes ; et de plus si vous avez besoin d'une sortie au format, disons PDF, vous pouvez envisager d'hériter de `Calendar` et de n'implémenter que la partie spécifique au format PDF.

C'est un peu le niveau élémentaire de l'héritage.

Le module `SocketServer` Toujours dans la bibliothèque standard, le module `SocketServer` fait un usage beaucoup plus sophistiqué de l'héritage.

Le module propose une hiérarchie de classes comme ceci :



Ici encore notre propos n'est pas d'entrer dans les détails, mais d'observer le fait que les différents *niveaux d'intelligence* sont ajoutés les uns aux les autres au fur et à mesure que l'on descend le graphe d'héritage.

Cette hiérarchie est par ailleurs étendue par le module `http.server` et notamment au travers de la classe `HTTPServer` qui hérite de `TCPServer`.

Pour revenir au module `SocketServer`, j'attire votre attention dans la page d'exemples sur cet exemple en particuler, où on crée une classe de serveurs multi-threads (la classe `ThreadedTCPServer`) par simple héritage multiple entre `ThreadingMixIn` et `TCPServer`. La notion de `Mixin` est décrite dans cette page wikipédia dans laquelle vous pouvez accéder directement à la section consacrée à python.

6.9 Hériter des types *built-in* ?

6.9.1 Complément - niveau avancé

Vous vous demandez peut-être s'il est possible d'hériter des types *built-in*.

La réponse est oui et nous allons voir un exemple qui est parfois très utile en pratique, c'est le type - ou plus exactement la famille de types - `namedtuple`

La notion de *record*

On se place dans un contexte voisin de celui de *record* qu'on a déjà rencontré souvent ; pour ce notebook nous allons à nouveau prendre le cas du point à deux coordonnées x et y. Nous avons déjà vu que pour implémenter un point on peut utiliser :

un dictionnaire

```
In [ ]: p1 = {'x': 1, 'y': 2}
        # ou de manière équivalente
        p1 = dict(x=1, y=2)
```

ou une classe

```
In [ ]: class Point:
            def __init__(self, x, y):
                self.x = x
                self.y = y

        p2 = Point(1, 2)
```

Nous allons voir une troisième façon de s'y prendre, qui présente deux caractéristiques :

— les objets seront non-mutables (en fait ce sont des tuples) ;
— et accessoirement on pourra accéder aux différents champs par leur nom aussi bien que par un index.

Pous faire ça il nous faut donc créer une sous-classe de `tuple` ; pour nous simplifier la vie, le module `collections` nous offre un utilitaire :

`namedtuple`

```
In [ ]: from collections import namedtuple
```

Techniquement, il s'agit d'une fonction :

```
In [ ]: type(namedtuple)
```

qui **renvoie une classe** - oui les classes sont des objets comme les autres ; par exemple pour créer une classe `TuplePoint`, on ferait :

```
In [ ]: # on passe à namedtuple
        #  - le nom du type qu'on veut créer
        #  - la liste ordonnée des composants (champs)
        TuplePoint = namedtuple('TuplePoint', ['x', 'y'])
```

Et maintenant si je crée un objet :

```
In [ ]: p3 = TuplePoint(1, 2)
```

```
In [ ]: # cet objet est un tuple
        isinstance(p3, tuple)
```

```
In [ ]: # auquel je peux accéder par index
        # comme un tuple
        p3[0]
```

```
In [ ]: # mais aussi par nom via un attribut
        p3.x
```

```
In [ ]: # et comme c'est un tuple il est immuable
        try:
            p3.x = 10
        except Exception as e:
            print(f"OOPS {type(e)} {e}")
```

À quoi ça sert

J'admets que ce n'est pas d'un usage fréquent, mais on en a déjà rencontré un exemple dans le notebook sur le module `pathlib`. En effet le type de retour de la méthode `Path.stat` est un `namedtuple` :

```
In [ ]: from pathlib import Path
        dot_stat = Path('.').stat()

In [ ]: dot_stat

In [ ]: isinstance(dot_stat, tuple)
```

Nom

Quand on crée une classe avec l'instruction `class`, on ne mentionne le nom de la classe qu'une seule fois. Ici vous avez remarqué qu'il faut en pratique le donner deux fois. Pour être précis, le paramètre qu'on a passé à `namedtuple` sert à ranger le nom dans l'attribut `__name__` de la classe créée :

```
In [ ]: Foo = namedtuple('Bar', ['spam', 'eggs'])

In [ ]: # Foo est le nom de la variable classe
        foo = Foo(1, 2)

In [ ]: # mais cette classe a son attribut __name__ mal positionné
        Foo.__name__
```

Il est donc préférable d'utiliser deux fois le même nom..

Mémoire

À titre de comparaison voici la place prise par chacun de ces objets ; le `namedtuple` ne semble pas de ce point de vue spécialement attractif par rapport à une instance :

```
In [ ]: import sys

        # p1 = dict / p2 = instance / p3 = namedtuple

        for p in p1, p2, p3:
            print(sys.getsizeof(p))
```

Pour en savoir plus

Si vous êtes intéressés de savoir comment on peut bien arriver à rendre les objets d'une classe immuable, vous pouvez commencer par regarder le code utilisé par `namedtuple` pour créer son résultat, en l'invoquant avec le mode bavard.

Vous y remarquerez notamment :

— une redéfinition de la méthode spéciale `__new__`,
— et aussi un usage des `property` que l'on a rencontrés en début de semaine.

Vous pouvez vous reporter à la documentation officielle.

```
In [ ]: # le code utilisé pour implémenter un namedtuple
        Point = namedtuple('Point', ['x', 'y'], verbose=True)
```

6.10 Énumérations

6.10.1 Complément - niveau basique

On trouve dans d'autres langages la notion de type énumérés.

L'usage habituel, c'est typiquement un code d'erreur qui peut prendre certaines valeurs précises. pensez par exemple aux codes prévus par le protocole HTTP. Le protocole prévoit un code de retour qui peut prendre un ensemble fini de valeurs, comme par exemple 200, 301, 302, 404, 500, mais pas 90 ni 110.

On veut pouvoir utiliser des noms parlants dans les programmes qui gèrent ce type de valeurs, c'est une application typique des types énumérés.

La bibliothèque standard offre depuis python-3.4 un module qui s'appelle sans grande surprise `enum`, et qui expose entre autres une classe `Enum`. On l'utiliserait comme ceci, dans un cas d'usage plus simple :

```
In [ ]: from enum import Enum

In [ ]: class Flavour(Enum):
            CHOCOLATE = 1
            VANILLA = 2
            PEAR = 3

In [ ]: vanilla = Flavour.VANILLA
```

Un premier avantage est que les représentations textuelles sont plus parlantes :

```
In [ ]: str(vanilla)

In [ ]: repr(vanilla)
```

Vous pouvez aussi retrouver une valeur par son nom :

```
In [ ]: chocolate = Flavour['CHOCOLATE']
        chocolate

In [ ]: Flavour.CHOCOLATE
```

Et réciproquement :

```
In [ ]: chocolate.name
```

`IntEnum`

En fait, le plus souvent on préfère utiliser `IntEnum`, une sous-classe de `Enum` qui permet également de faire des comparaisons. Pour reprendre le cas des codes d'erreur HTTP :

```
In [ ]: from enum import IntEnum

        class HttpError(IntEnum):

            OK = 200
            REDIRECT = 301
            REDIRECT_TMP = 302
            NOT_FOUND = 404
```

```
    INTERNAL_ERROR = 500

    # avec un IntEnum on peut faire des comparaisons
    def is_redirect(self):
        return 300 <= self.value <= 399
```

```
In [ ]: code = HttpError.REDIRECT_TMP
```

```
In [ ]: code.is_redirect()
```

Pour en savoir plus

Consultez la documentation officielle du module `enum`.

6.11 Héritage, typage

6.11.1 Complément - niveau avancé

Dans ce complément, nous allons revenir sur la notion de *duck typing*, et attirer votre attention sur cette différence assez essentielle entre python et les langages statiquement typés. On s'adresse ici principalement à ceux d'entre vous qui sont habitués à C++ et/ou Java.

Type concret et type abstrait

Revenons sur la notion de type et remarquons que les types peuvent jouer plusieurs rôles, comme on l'a évoqué rapidement en première semaine ; et pour reprendre des notions standard en langages de programmation nous allons distinguer deux types.

0. **type concret :** d'une part, la notion de type a bien entendu à voir avec l'implémentation ; par exemple, un compilateur C a besoin de savoir très précisément quel espace allouer à une variable, et l'interpréteur python sous-traite à la classe le soin d'initialiser un objet ;
1. **type abstrait :** d'autre part, les types sont cruciaux dans les systèmes de vérification statique, au sens large, dont le but est de trouver un maximum de défauts à la seule lecture du code (par opposition aux techniques qui nécessitent de le faire tourner).

Duck typing

En python, ces deux aspects du typage sont relativement décorrélés.

Pour la deuxième dimension du typage, le système de types abstraits de python est connu sous le nom de *duck typing*, une appellation qui fait référence à cette phrase :

```
When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call
```

L'exemple des itérables

Pour prendre l'exemple sans doute le plus représentatif, la notion d'*itérable* est un type abstrait, en ce sens que pour que le fragment :

```
for item in container:
    do_something(item)
```

ait un sens, il faut et il suffit que `container` soit un itérable. Et vous connaissez maintenant plein d'exemples très différents d'objets itérables, a minima parmi les *built-in* `str`, `list`, `tuple`, `range`...

Dans un langage typé statiquement, pour pouvoir donner un type à cette construction, on serait **obligé** de définir un type - qu'on appellerait logiquement une classe abstraite - dont ces trois types seraient des descendants.

En python, et c'est le point que nous voulons souligner dans ce complément, il n'existe pas dans le système python d'objet de type `type` qui matérialise l'ensemble des `iterables`. Si on regarde les superclasses de nos types concrets itérables, on voit que leur seul ancêtre commun est la classe `object` :

```
In [ ]: str.__bases__

In [ ]: list.__bases__

In [ ]: tuple.__bases__

In [ ]: range.__bases__
```

Un autre exemple

Pour prendre un exemple plus simple, si je considère :

```
def foo(graphic):
    ...
    graphic.draw()
```

pour que l'expression `graphic.draw()` ait un sens, il faut et il suffit que l'objet `graphic` ait une méthode `draw`.

À nouveau, dans un langage typé statiquement, on serait amené à définir une classe abstraite `Graphic`. En python ce n'est **pas requis** ; vous pouvez utiliser ce code tel quel avec deux classes `Rectangle` et `Texte` qui n'ont pas de rapport entre elles - autres que, à nouveau, d'avoir `object` comme ancêtre commun - pourvu qu'elles aient toutes les deux une méthode `draw`.

Héritage et type abstrait

Pour résumer, en python comme dans les langages typés statiquement, on a bien entendu la bonne propriété que si, par exemple, la classe `Spam` est itérable, alors la classe `Eggs` qui hérite de `Spam` est itérable.

Mais dans l'autre sens, si `Foo` et `Bar` sont itérables, il n'y a pas forcément une superclasse commune qui représente l'ensemble des objets itérables.

`isinstance` sur stéroïdes

D'un autre côté, c'est très utile d'exposer au programmeur un moyen de vérifier si un objet a un *type* donné - dans un sens volontairement vague ici.

On a déjà parlé en Semaine 4 de l'intérêt qu'il peut y avoir à tester le type d'un argument avec `isinstance` dans une fonction, pour parvenir à faire l'équivalent de la surcharge en C++ (la surcharge en C++, c'est quand vous définissez plusieurs fonctions qui ont le même nom mais des types d'arguments différents).

C'est pourquoi, quand on a cherché à exposer au programmeur des propriétés comme "cet objet est-il iterable ?", on a choisi d'étendre *isinstance* au travers de cette initiative. C'est ainsi qu'on peut faire par exemple :

```
In [ ]: from collections.abc import Iterable

In [ ]: isinstance('ab', Iterable)

In [ ]: isinstance([1, 2], Iterable)

In [ ]: # comme on l'a vu, un objet qui a des méthodes
        # __iter__() et __next__()
        # est considéré comme un itérable
        class Foo:
            def __iter__(self):
                return self
            def __next__(self):
                # ceci naturellement est bidon
                return

In [ ]: foo = Foo()
        isinstance(foo, Iterable)
```

L'implémentation du module `abc` donne l'**illusion** que `Iterable` est un objet dans la hiérarchie de classes, et que tous ces *classes* `str`, `list`, et `Foo` lui sont asujetties, mais ce n'est pas le cas en réalité ; comme on l'a vu plus tôt, ces trois types ne sont pas comparables dans la hiérarchie de classes, ils n'ont pas de plus petit (ou plus grand) élément à part `object`.

Je signale pour finir, à propos de `isinstance` et du module `collections`, que la définition du symbole `Hashable` est à mon avis beaucoup moins convaincante que `Iterable` ; si vous vous souvenez qu'en Semaine 3, Séquence "les dictionnaires", on avait vu que les clés doivent être globalement immuables. C'est une caractéristique qui est assez difficile à écrire, et en tous cas ceci de mon point de vue ne remplit pas la fonction :

```
In [ ]: from collections import Hashable

In [ ]: # un tuple qui contient une liste ne convient
        # pas comme clé dans un dictionnaire
        # et pourtant
        isinstance (([1], [2]), Hashable)
```

python et les classes abstraites

Les points à retenir de ce complément un peu digressif sont :

— en python, on hérite des **implémentations** et pas des **spécifications** ;
— et le langage n'est pas taillé pour tirer profit de **classes abstraites** - même si rien ne vous interdit d'écrire, pour des raisons documentaires, une classe qui résume l'interface qui est attendue par tel ou tel système de plugin.

Venant de C++ ou de Java, cela peut prendre du temps d'arriver à se débarrasser de l'espèce de réflexe qui fait qu'on pense d'abord classe abstraite, puis implémentations.

Pour aller plus loin

La documentation du module `collections.abc` contient la liste de tous les symboles exposés par ce module, dont par exemple en vrac :

— `Iterable`
— `Iterator`

— `Hashable`
— `Generator`
— `Coroutine` (rendez-vous semaine 8)

et de nombreux autres.

Avertissement

Prenez garde enfin que ces symboles n'ont - à ce stade du moins - pas de relation forte avec ceux du module `typing` dont on a parlé lorsqu'on a vu les *type hints*.

6.12 Héritage multiple

6.12.1 Complément - niveau intermédiaire

La classe `object`

Le symbole `object` est une variable prédéfinie (qui donc fait partie du module `builtins`) :

```
In [ ]: object

In [ ]: import builtins

        builtins.object is object
```

La classe `object` est une classe spéciale ; toutes les classes en python héritent de la classe `object`, même lorsqu'aucun héritage n'est spécifié :

```
In [ ]: class Foo:
            pass

        Foo.__bases__
```

L'attribut spécial `__bases__`, comme on le devine, nous permet d'accéder aux superclasses directes, ici de la classe `Foo`.

En python moderne, on n'a **jamais besoin de mentionner** `object` dans le code. La raison de sa présence dans les symboles prédéfinis est liée à l'histoire de python, et à la distinction que faisait python2 entre classes *old-style* et classes *new-style*. Nous le mentionnons seulement car on rencontre encore parfois du code qui fait quelque chose comme :

```
In [ ]: class Bar(object):
            pass
```

qui est un reste de python2, et que python3 accepte uniquement au titre de la compatibilité.

6.12.2 Complément - niveau avancé

Rappels

L'héritage en python consiste principalement en l'algorithme de recherche d'un attribut d'une instance ; celui-ci regarde :

0. d'abord dans l'instance ;
1. ensuite dans la classe ;
2. ensuite dans les super-classes.

Ordre sur les super-classes

Le problème revient donc, pour le dernier point, à définir un **ordre** sur l'ensemble des **super-classes**. On parle bien, naturellement, de **toutes** les super-classes, pas seulement celles dont on hérite directement - en termes savants on dirait qu'on s'intéresse à la fermeture transitive de la relation d'héritage.

L'algorithme utilisé pour cela depuis la version 2.3 est connu sous le nom de **linéarisation C3**. Cet algorithme n'est pas propre à python, comme vous pourrez le lire dans les références citées dans la dernière section.

Nous ne décrirons pas ici l'algorithme lui-même dans le détail ; par contre nous allons :

— dans un premier temps résumer **les raisons** qui ont guidé ce choix, en décrivant les bonnes propriétés que l'on attend, ainsi que les **limitations** qui en découlent ;
— puis voir l'ordre obtenu sur quelques **exemples** concrets de hiérarchies de classes.

Vous trouverez dans les références (voir ci-dessous la dernière section, "Pour en savoir plus") des liens vers des documents plus techniques si vous souhaitez creuser le sujet.

Les bonnes propriétés attendues

Il y a un certain nombre de bonnes propriétés que l'on attend de cet algorithme.

Priorité au spécifique Lorsqu'une classe A hérite d'une classe B, on s'attend à ce que les méthodes définies sur A, qui sont en principe plus spécifiques, soient utilisées de préférence à celles définies sur B.

Priorité à gauche Lorsqu'on utilise l'héritage multiple, on mentionne les classes mères dans un certain ordre, qui n'est pas anodin. Les classes mentionnées en premier sont bien entendu celles desquelles on veut hériter en priorité.

6.13 La Method Resolution Order (MRO)

De manière un peu plus formelle Pour reformuler les deux points ci-dessus, on s'intéresse à la `mro` d'une classe O, et on veut avoir les deux bonnes propriétés suivantes :

— si O hérite (pas forcément directement) de A qui elle même hérite de B, alors A est avant B dans la `mro` de O ;
— si O hérite (pas forcément directement) de A, qui elle hérite de B, puis (pas forcément immédiatement) de C, alors dans la `mro` A est avant B qui est avant C.

Limitations : toutes les hiérarchies ne peuvent pas être traitées L'algorithme C3 permet de calculer un ordre sur $\mathcal{S}$ qui respecte toutes ces contraintes, lorsqu'il en existe un.

En effet, dans certains cas on ne peut pas trouver un tel ordre, on le verra plus bas, mais dans la pratique, il est assez rare de tomber sur de tels cas pathologiques ; et lorsque cela se produit c'est en général le signe d'erreurs de conception plus profondes.

Un exemple très simple

On se donne la hiérarchie suivante :

```
In [ ]: class LeftTop(object):
            def attribut(self):
                return "attribut(LeftTop)"
```

```
class LeftMiddle(LeftTop):
    pass

class Left(LeftMiddle):
    pass

class Middle(object):
    pass

class Right(object):
    def attribut(self):
        return "attribut(Right)"

class Class(Left, Middle, Right):
    pass

instance = Class()
```

qui donne en version dessinée, avec deux points rouges pour représenter les deux définitions de la méthode `attribut` :

Les deux règles, telles que nous les avons énoncées en premier lieu (priorité à gauche, priorité au spécifique) sont un peu contradictoires ici. En fait, c'est la méthode de `LeftTop` qui est héritée dans `Class`, comme on le voit ici :

```
In [ ]: instance.attribut() == 'attribut(LeftTop)'
```

Exercice : Remarquez qu'ici `Right` a elle-même un héritage très simple. À titre d'exercice, modifiez le code ci-dessus pour faire que `Right` hérite de la classe `LeftMiddle` ; de quelle classe d'après vous est-ce que `Class` hérite `attribut` dans cette configuration ?

Si cela ne vous convient pas C'est une évidence, mais cela va peut-être mieux en le rappelant : si la méthode que vous obtenez "gratuitement" avec l'héritage n'est pas celle qui vous convient, vous avez naturellement toujours la possibilité de la redéfinir, et ainsi d'en **choisir** une autre. Dans notre exemple si on préfère la méthode implémentée dans `Right`, on définira plutôt la classe `Class` comme ceci :

```
In [ ]: class Class(Left, Middle, Right):
            # en redéfinissant explicitement la méthode
            # attribut ici on court-circuite la mro
            # et on peut appeler explicitement une autre
            # version de attribut()
            def attribut(*args, **kwds):
                return Right.attribut(*args, **kwds)

        instance2 = Class()
        instance2.attribut()
```

Ou encore bien entendu, si dans votre contexte vous devez appelez **les deux** méthodes dont vous pourriez hériter et les combiner, vous pouvez le faire aussi, par exemple comme ceci :

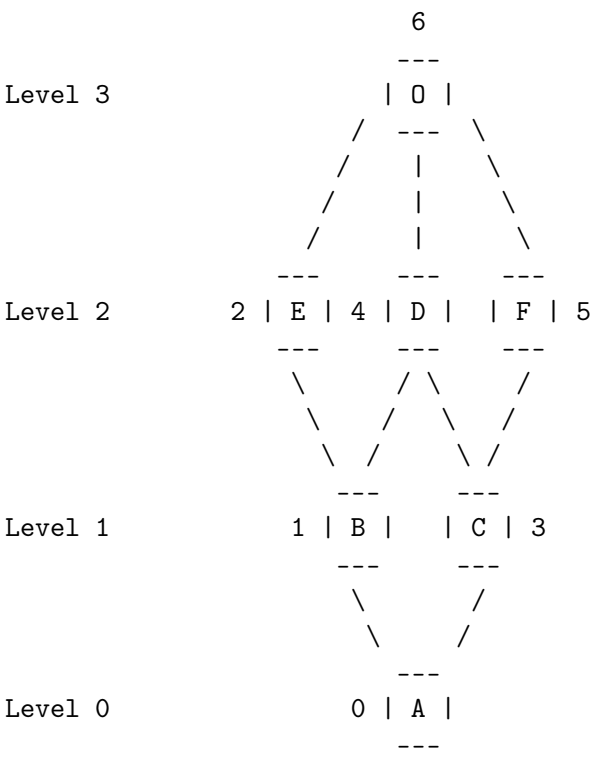
```
In [ ]: class Class(Left, Middle, Right):
            # pour faire un composite des deux méthodes
            # trouvées dans les classes mères
            def attribut(*args, **kwds):
                return LeftTop.attribut(*args, **kwds) + " ** " + Right.attribut(*args, **kwd

        instance3 = Class()
        instance3.attribut()
```

Un exemple un peu plus compliqué

Voici un exemple, assez parlant, tiré de la deuxième référence (voir ci-dessous la dernière section, "Pour en savoir plus").

```
In [ ]: O = object
        class F(O): pass
        class E(O): pass
        class D(O): pass
        class C(D, F): pass
        class B(E, D): pass
        class A(B, C): pass
```

Cette hiérarchie nous donne, en partant de A, l'ordre suivant :



Que l'on peut calculer, sous l'interpréteur python, avec la méthode `mro` sur la classe de départ :

```
In [ ]: A.mro()
```

Un exemple qui ne peut pas être traité

Voici enfin un exemple de hiérarchie pour laquelle on ne **peut pas trouver d'ordre** qui respecte les bonnes propriétés que l'on a vues tout à l'heure, et qui pour cette raison sera **rejetée par l'interpréteur python**. D'abord en version dessinée :

```
In [ ]: # puis en version code
        class X: pass
        class Y: pass
        class XY(X, Y): pass
        class YX(Y, X): pass

        # on essaie de créer une sous-classe de XY et YX
        try:
            class Class(XY, YX): pass
        # mais ce n'est pas possible
        except Exception as e:
            print(f"OOPS, {type(e)}, {e}")
```

Pour en savoir plus

0. Un blog de Guido Van Rossum qui retrace l'historique des différents essais qui ont été faits avant de converger sur le modèle actuel.
1. Un article technique qui décrit le fonctionnement de l'algorithme de calcul de la MRO, et donne des exemples.
2. L'article de wikipedia sur l'algorithme C3.

6.14 Les attributs

6.14.1 Compléments - niveau basique

La notation . et les attributs

La notation `module.variable` que nous avons vue dans la vidéo est un cas particulier de la notion d'attribut, qui permet d'étendre un objet, ou si on préfère de lui accrocher des annotations.

Nous avons déjà rencontré ceci de nombreuses fois à présent, c'est exactement le même mécanisme d'attribut qui est utilisé pour les méthodes ; pour le système d'attribut il n'y a pas de différence entre `module.variable`, `module.fonction`, `objet.methode`, etc.

Nous verrons très bientôt que ce mécanisme est massivement utilisé également dans les instances de classe.

Les fonctions de gestion des attributs

Pour accéder programmativement aux attributs d'un objet, on dispose des 3 fonctions *built-in* `getattr`, `setattr`, et `hasattr`, que nous allons illustrer tout de suite.

Lire un attribut

```
In [ ]: import math
        # nous savons lire un attribut comme ceci
        # qui lit l'attribut de nom 'pi' dans le module math
        math.pi
```

La fonction *built-in* `getattr` permet de lire un attribut programmativement :

```
In [ ]: # si on part d'une chaîne qui désigne le nom de l'attribut
        # la formule équivalente est alors
        getattr(math, 'pi')

In [ ]: # on peut utiliser les attributs avec la plupart des objets
        # ici nous allons le faire sur une fonction
        def foo():
            "une fonction vide"
            pass

        # on a déjà vu certains attributs des fonctions
        print(f"nom={foo.__name__}, docstring=`{foo.__doc__}`")

In [ ]: # on peut préciser une valeur par défaut pour le cas où l'attribut
        # n'existe pas
        getattr(foo, "attribut_inexistant", 'valeur_par_defaut')
```

Écrire un attribut

```
In [ ]: # on peut ajouter un attribut arbitraire (toujours sur l'objet fonction)
        foo.hauteur = 100

        foo.hauteur
```

Comme pour la lecture on peut écrire un attribut programmativement avec la fonction *built-in* `setattr` :

```
In [ ]: # écrire un attribut avec setattr
        setattr(foo, "largeur", 200)

        # on peut bien sûr le lire indifféremment
        # directement comme ici, ou avec getattr
        foo.largeur
```

Liste des attributs La fonction *built-in* `hasattr` permet de savoir si un objet possède ou pas un attribut :

```
In [ ]: # pour savoir si un attribut existe
        hasattr(math, 'pi')
```

Ce qui peut aussi être retrouvé autrement, avec la fonction *built-in* `vars` :

```
In [ ]: vars(foo)
```

Sur quels objets

Il n'est pas possible d'ajouter des attributs sur les types de base, car ce sont des classes immuables :

```
In [ ]: for builtin_type in (int, str, float, complex, tuple, dict, set, frozenset):
            obj = builtin_type()
            try:
                obj.foo = 'bar'
            except AttributeError as e:
                print(f"{builtin_type.__name__:>10} → exception {type(e)} - {e}")
```

C'est par contre possible sur virtuellement tout le reste, et notamment là où c'est très utile, c'est-à-dire pour ce qui nous concerne sur les : * modules * packages * fonctions * classes * instances

6.15 Espaces de nommage

6.15.1 Complément - niveau basique

Nous venons de voir les règles pour l'affectation (ou l'assignation) et le référencement des variables et des attributs ; en particulier, on doit faire une distinction entre les attributs et les variables.

— Les attributs sont résolus de manière **dynamique**, c'est-à-dire au moment de l'exécution (*run-time*) ;
— alors que la liaison des variables est par contre **statique** (*compile-time*) et **lexicale**, en ce sens qu'elle se base uniquement sur les imbrications de code.

Vous voyez donc que la différence entre attributs et variables est fondamentale. Dans ce complément, nous allons reprendre et résumer les différentes règles qui régissent l'affectation et le référencement des attributs et des variables.

Attributs Un attribut est un symbole `x` utilisé dans la notation `obj.x` où `obj` est l'objet qui définit l'espace de nommage sur lequel `x` existe.

L'**affectation** (explicite ou implicite) d'un attribut `x` sur un objet `obj` va créer (ou altérer) un symbole `x` **directement** dans l'espace de nommage de `obj`, symbole qui va référencer l'objet affecté, typiquement l'objet à droite du signe `=`

```
In [ ]: class MaClasse:
            pass

        # affectation explicite
        MaClasse.x = 10

        # le symbole x est défini dans l'espace de nommage de MaClasse
        'x' in MaClasse.__dict__
```

Le **référencement** (la lecture) d'un attribut va chercher cet attribut **le long de l'arbre d'héritage** en commençant par l'instance, puis la classe qui a créé l'instance, puis les super-classes et suivant la MRO (voir le complément sur l'héritage multiple).

Variables Une variable est un symbole qui n'est pas précédé de la notation `obj.` et l'affectation d'une variable rend cette variable locale au bloc de code dans lequel elle est définie, un bloc de code pouvant être :

— une fonction, dans ce cas la variable est locale à la fonction ;
— une classe, dans ce cas la variable est locale à la classe ;

— un module, dans ce cas la variable est locale au module, on dit également que la variable est globale.

Une variable référencée est toujours cherchée suivant la règle LEGB :

— localement au bloc de code dans lequel elle est référencée ;
— puis dans les blocs de code des **fonctions ou méthodes** englobantes, s'il y en a, de la plus proche à la plus eloignée ;
— puis dans le bloc de code du module.

Si la variable n'est toujours pas trouvée, elle est cherchée dans le module `builtins` et si elle n'est toujours pas trouvée, une exception est levée.

Par exemple :

```
In [ ]: var = 'dans le module'

        class A:
            var = 'dans la classe A'
            def f(self):
                var = 'dans la fonction f'
                class B:
                    print(var)
                B()
        A().f()
```

En résumé Dans la vidéo et dans ce complément basique, on a couvert tous les cas standards, et même si python est un langage plutôt mieux fait, avec moins de cas particuliers que d'autres langages, il a également ses cas étranges entre raisons historiques et bugs qui ne seront jamais corrigés (parce que ça casserait plus de choses que ça n'en réparerait). Pour éviter de tomber dans ces cas spéciaux, c'est simple, vous n'avez qu'à suivre ces règles :

— ne jamais affecter dans un bloc de code local une variable de même nom qu'une variable globale ;
— éviter d'utiliser les directives `global` et `nonlocal`, et les réserver pour du code avancé comme les décorateurs et les métaclasses ;
— et lorsque vous devez vraiment les utiliser, toujours mettre les directives `global` et `nonlocal` comme premières instructions du bloc de code où elle s'appliquent.

Si vous ne suivez pas ces règles, vous risquez de tomber dans un cas particulier que nous détaillons ci-dessous dans la partie avancée.

6.15.2 Complément - niveau avancé

La documentation officielle est fausse Oui, vous avez bien lu, la documentation officielle est fausse sur un point subtil. Regardons le modèle d'exécution, on trouve la phrase suivante "If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block." qui est fausse, il faut lire "If a name binding operation occurs anywhere within a code block **of a function**, all uses of the name within the block are treated as references to the current block."

En effet, les classes se comportent différemment des fonctions :

```
In [ ]: x = "x du module"
        class A():
            print("dans classe A: " + x)
            x = "x dans A"
```

```
        print("dans classe A: " + x)
        del x
        print("dans classe A: " + x)
```

Alors pourquoi si c'est une mauvaise idée de mélanger variables globales et locales de même nom dans une fonction, c'est possible dans une classe ?

Cela vient de la manière dont sont implémentés les espaces de nommage. Normalement, un objet a pour espace de nommage un dictionnaire qui s'appelle `__dict__`. D'un côté un dictionnaire est un objet python qui offre beaucoup de flexibilité, mais d'un autre côté, il induit un petit surcoût pour chaque recherche d'éléments. Comme les fonctions sont des objets qui par définition peuvent être appelés très souvent, il a été décidé de mettre toutes les variables locales à la fonction dans un objet écrit en C qui n'est pas dynamique (on ne peut pas ajouter des éléments à l'exécution), mais qui est un peu plus rapide qu'un dictionnaire lors de l'accès aux variables. Mais pour faire cela, il faut déterminer la portée de la variable dans la phase de précompilation. Donc si le précompilateur trouve une affectation (explicite ou implicite) dans une fonction, il considère la variable comme locale pour tout le bloc de code. Donc si on référence une variable définie comme étant locale avant une affectation dans la fonction, on ne va pas la chercher globalement, on a une erreur `UnboundLocalError`.

Cette optimisation n'a pas été faite pour les classes, parce que dans l'évaluation du compromis souplesse contre efficacité pour les classes, c'est la souplesse, donc le dictionnaire qui a gagné.

6.15.3 Complément - niveau avancé

Implémenter un itérateur de permutations

Dans ce complément nous allons nous amuser à implémenter une fonctionnalité qui est déjà disponible dans le module `itertools`.

C'est quoi déjà les permutations ? En guise de rappel, l'ensemble des permutations d'un ensemble fini correspond à toutes les façons d'ordonner ses éléments ; si l'ensemble est de cardinal n, il possède $n!$ permutations : on a n façons de choisir le premier élément, $n-1$ façons de choisir le second, etc.

Un itérateur sur les permutations est disponible au travers du module standard `itertools`. Cependant il nous a semblé intéressant de vous montrer comment nous pourrions écrire nous-mêmes cette fonctionnalité, de manière relativement simple.

Pour illustrer le concept, voici à quoi ressemblent les 6 permutations d'un ensemble à trois éléments :

```
In [ ]: from itertools import permutations

In [ ]: set = {1, 2, 3}

        for p in permutations(set):
            print(p)
```

Une implémentation Voici une implémentation possible pour un itérateur de permutations :

```
In [ ]: class Permutations:
            """
            Un itérateur qui énumère les permutations de n
```

```
sous la forme d'une liste d'indices commençant à 0
"""
def __init__(self, n):
    # le constructeur bien sûr ne fait (presque) rien
    self.n = n
    # au fur et à mesure des itérations
    # le compteur va aller de 0 à n-1
    # puis retour à 0 et comme ça en boucle sans fin
    self.counter = 0
    # on se contente d'allouer un iterateur de rang n-1
    # si bien qu'une fois qu'on a fini de construire
    # l'objet d'ordre n on a n objets Permutations en tout
    if n >= 2:
        self.subiterator = Permutations(n-1)

# pour satisfaire le protocole d'itération
def __iter__(self):
    return self

# c'est ici bien sûr que se fait tout le travail
def __next__(self):
    # pour n == 1
    # le travail est très simple
    if self.n == 1:
        # on doit renvoyer une fois la liste [0]
        # car les indices commencent à 0
        if self.counter == 0:
            self.counter += 1
            return [0]
        # et ensuite c'est terminé
        else:
            raise StopIteration

    # pour n >= 2
    # lorsque counter est nul,
    # on traite la permutation d'ordre n-1 suivante
    # si next() lève StopIteration on n'a qu'à laisser passer
    # car en effet c'est qu'on a terminé
    if self.counter == 0:
        self.subsequence = next(self.subiterator)
    #
    # on insère alors n-1 (car les indices commencent à 0)
    # successivement dans la sous-sequence
    #
    # naivement on écrirait
    # result = self.subsequence[0:self.counter] \
    #     + [self.n - 1] \
    #     + self.subsequence[self.counter:self.n-1]
    # mais c'est mettre le nombre le plus élevé en premier
    # et donc à itérer les permutations dans le mauvais ordre,
    # en commençant par la fin
```

```
            #
            # donc on fait plutôt une symétrie
            # pour insérer en commençant par la fin
            cutter = self.n-1 - self.counter
            result = self.subsequence[0:cutter] + [self.n - 1] \
                     + self.subsequence[cutter:self.n-1]
            #
            # on n'oublie pas de maintenir le compteur et de
            # le remettre à zéro tous les n tours
            self.counter = (self.counter+1) % self.n
            return result

    # la longeur de cet itérateur est connue
    def __len__(self):
        import math
        return math.factorial(self.n)
```

Ce qu'on a essayé d'expliquer dans les commentaires, c'est qu'on procède en fin de compte par récurrence. Un objet `Permutations` de rang `n` possède un sous-itérateur de rang `n-1` qu'on crée dans le constructeur. Ensuite l'objet de rang `n` va faire successivement (c'est-à-dire à chaque appel de `next()`) :

— appel *0* :
— demander à son sous-itérateur une permutation de rang `n-1` (en lui envoyant `next`),
— la stocker dans l'objet de rang `n`, ce sera utilisé par les *n* premier appels,
— et construire une liste de taille `n` en insérant `n-1` à la fin de la séquence de taille `n-1`,
— appel *1* :
— insérer `n-1` dans la même séquence de rang `n-1` mais cette fois 1 cran avant la fin,
— ...
— appel *n-1* :
— insérer `n-1` au début de la séquence de rang `n-1`,
— appel *n* :
— refaire `next()` sur le sous-itérateur pour traiter une nouvelle sous-séquence,
— la stocker dans l'objet de rang `n`, comme à l'appel *0*, pour ce bloc de n appels,
— et construire la permutation en insérant *n-1* à la fin, comme à l'appel 0,
— ...

On voit donc le caractère cyclique d'ordre *n* qui est matérialisé par `counter`, que l'on incrémente à chaque boucle mais modulo *n* - notez d'ailleurs que pour ce genre de comportement on dispose aussi de `itertools.cycle` comme on le verra dans une deuxième version, mais pour l'instant j'ai préféré ne pas l'utiliser pour ne pas tout embrouiller ;)

La terminaison se gère très simplement, car une fois que l'on a traité toutes les séquences d'ordre *n-1* eh bien on a fini, on n'a même pas besoin de lever StopIteration explicitement, sauf bien sûr dans le cas *n=1*.

Le seul point un peu délicat, si on veut avoir les permutations dans le "bon" ordre, consiste à commencer à insérer `n-1` par la droite (la fin de la sous-séquence).

Discussion Il existe certainement des tas d'autres façons de faire bien entendu. Le point important ici, et qui donne toute sa puissance à la notion d'itérateur, c'est **qu'à aucun moment on ne construit** une liste ou une séquence quelconque de** n! termes**.

C'est une erreur fréquente chez les débutants que de calculer une telle liste dans le constructeur, mais procéder de cette façon c'est aller exactement à l'opposé de ce pourquoi les itérateurs ont été conçus ; au contraire, on veut éviter à tout prix le coût d'une telle construction.

On peut le voir sur un code qui n'utiliserait que les 20 premières valeurs de l'itérateur, vous constatez que ce code est immédiat :

```
In [ ]: def show_first_items(iterable, nb_items):
            """
            montre les <nb_items> premiers items de iterable
            """
            print(f"Il y a {len(iterable)} items dans l'itérable")
            for i, item in enumerate(iterable):
                print(item)
                if i >= nb_items:
                    print('....')
                    break
```

```
In [ ]: show_first_items(Permutations(12), 20)
```

Ce tableau vous montre par ailleurs sous un autre angle comment fonctionne l'algorithme, si vous observez le `11` qui balaie en diagonale les 12 premières lignes, puis les 12 suivantes, etc..

Ultimes améliorations Dernières remarques, sur des améliorations possibles - mais tout à fait optionnelles :

— le lecteur attentif aura remarqué qu'au lieu d'un entier `counter` on aurait pu profitablement utiliser une instance de `itertools.cycle`, ce qui aurait eu l'avantage d'être plus clair sur le propos de ce compteur ;
— aussi dans le même mouvement, au lieu de se livrer à la gymnastique qui calcule `cutter` à partir de `counter`, on pourrait dès le départ créer dans le cycle les bonnes valeurs en commençant à `n-1`.

C'est ce qu'on a fait dans cette deuxième version ; après avoir enlevé la loghorrée de commentaires ça redevient presque lisible ;)

```
In [ ]: import itertools

        class Permutations2:
            """
            Un itérateur qui énumère les permutations de n
            sous la forme d'une liste d'indices commençant à 0
            """
            def __init__(self, n):
                self.n = n
                # on commence à insérer à la fin
                self.cycle = itertools.cycle(list(range(n))[::-1])
                if n >= 2:
                    self.subiterator = Permutations2(n-1)
                # pour savoir quand terminer le cas n==1
```

```
            if n == 1:
                self.done = False

        def __iter__(self):
            return self

        def __next__(self):
            cutter = next(self.cycle)

            # quand n==1 on a toujours la même valeur 0
            if self.n == 1:
                if not self.done:
                    self.done = True
                    return [0]
                else:
                    raise StopIteration

            # au début de chaque séquence de n appels
            # il faut appeler une nouvelle sous-séquence
            if cutter == self.n-1:
                self.subsequence = next(self.subiterator)
            # dans laquelle on insére n-1
            return self.subsequence[0:cutter] + [self.n-1] \
                     + self.subsequence[cutter:self.n-1]

        # la longeur de cet itérateur est connue
        def __len__(self):
            import math
            return math.factorial(self.n)
```

```
In [ ]: show_first_items(Permutations2(5), 20)
```

6.16 *Context managers* et exceptions

6.16.1 Complément - niveau intermédiaire

On a vu jusqu'ici dans la vidéo comment écrire un context manager, mais on n'a pas envisagé le cas où une exception serait levée pendant la durée de vie du context manager.

Et c'est très important, car si je me contente de faire :

```
In [ ]: import time

        class Timer1:
            def __enter__(self):
                print("Entering Timer1")
                self.start = time.time()
                return self

            def __exit__(self, *args):
                print(f"Total duration {time.time()-self.start:2f}")
                return True
```

Alors dans les cas nominaux, tout se passe comme attendu :

```
In [ ]: with Timer1():
            n = 0
            for i in range(2*10**6):
                n += i**2
```

Mais par contre, dans le cas où j'exécute du code qui lève une exception, ça ne va plus du tout :

```
In [ ]: with Timer1():
            n = 0
            for i in range(2*10**6):
                n += i**2 / 0
```

À la toute première itération de la boucle, on fait une division par 0, qui lève l'exception `ZeroDivisionError`, mais tel qu'est conçue notre classe de context manager, cette exception **est étouffée** et n'est pas correctement propagée à l'extérieur.

Il est important, lorsqu'on conçoit un context manager, de bien **propager** les exceptions qui ne sont pas liées au fonctionnement attendu du context manager. Par exemple un objet de type fichier va en effet attraper par exemple les exceptions liées à la fin du fichier, mais doit par contre laisser passer une exception comme `ZeroDivisionError`.

Les paramètres de `__exit__`

Comme vous pouvez le retrouver ici, la méthode `__exit__` reçoit trois arguments :

```
def __exit__(self, exc_type, exc_value, traceback):
```

lorsqu'on sort du bloc `with` sans qu'une exception soit levée, ces trois arguments valent `None`. Par contre si une exception est levée, ils permettent d'accéder au type, à la valeur de l'exception, et à l'état de la pile lorsque l'exception est levée.

```
In [ ]: # une deuxième version de Timer
        # qui propage correctement les exceptions

        class Timer2:
            def __enter__(self):
                print("Entering Timer1")
                self.start = time.time()
                # rappel : le retour de __enter__ est ce qui est passé
                # à la clause `as` du `with`
                return self

            def __exit__(self, exc_type, exc_value, traceback):
                if exc_type is None:
                    print(f"Total duration {time.time()-self.start:2f}")
                    # ceci indique que tout s'est bien passé
                    return True
                else:
                    print(f"OOPS : on propage l'exception {exc_type} - {exc_value}")
                    # c'est ici que je propage l'exception au dehors du with
                    raise exc_type(exc_value)
                return True
```

```
In [ ]: try:
            with Timer2():
                n = 0
                for i in range(2*10**6):
                    n += i**2 / 0
        except Exception as e:
            print(f"L'exception a bien été propagée, {type(e)} - {e}")
```

Pour en savoir plus

Je vous signale enfin la bibliothèque `contextlib` qui offre quelques utilitaires pour se définir un contextmanager.

Notamment, un peu comme on peut implémenter un itérateur comme un générateur qui fait (n'importe quel nombre de) `yield`, on peut également implémenter un context manager simple sous la forme d'une fonction qui fait un `yield`.

6.17 Exercice sur l'utilisation des classes

Introduction

Objectifs de l'exercice Maintenant que vous avez un bagage qui couvre toutes les bases du langage, cette semaine nous ne ferons qu'un seul exercice de taille un peu plus réaliste. Vous devez écrire quelques classes, que vous intégrez ensuite dans un code écrit pas nos soins.

L'exercice comporte donc à la fois une part lecture et une part écriture.

Par ailleurs, cette fois-ci l'exercice n'est plus à faire dans un notebook ; vous êtes donc également incités à améliorer autant que possible l'environnement de travail sur votre propre ordinateur.

Objectifs de l'application Dans le prolongement des exercices de la semaine 3 sur les données maritimes, l'application dont il est question ici fait principalement ceci :

— en entrée :
— agréger des données obtenues auprès de marinetraffic ;
— et produire en sortie :
— un fichier texte qui liste par ordre alphabétique les bateaux concernés, et le nombre de positions trouvées pour chacun ;
— et un fichier KML, pour exposer les trajectoires trouvées à Google Earth, Google Maps ou autre outil similaire.

Les données générées dans ces deux fichiers sont triées dans l'ordre alphabétique, de façon à permettre une comparaison des résultats sous forme textuelle. Plus précisément, on trie les bateaux selon le critère suivant :

— ordre alphabétique sur le nom des bateaux ;
— et ordre sur les `id` en cas d'ex-aequo (il y a des bateaux homonymes dans cet échantillon réel).

Voici à quoi ressemble le fichier KML obtenu avec les données que nous fournissons, une fois ouvert sous Google Earth :

Choix d'implémentation En particulier, dans cet exercice nous allons voir comment on peut gérer des données sous forme d'instances de classes plutôt que de types de base. Cela résonne avec la discussion commencée en Semaine 3, Séquence "Les dictionnaires", dans le complément "record-et-dictionnaire".

Dans les exercices de cette semaine-là nous avions uniquement utilisé des types "standard" comme listes, tuples et dictionnaires pour modéliser les données, cette semaine nous allons faire le choix inverse et utiliser plus souvent des (instances de) classes.

Principe de l'exercice On a écrit une application complète, constituée de 4 modules ; on vous donne le code de trois de ces modules et vous devez écrire le module manquant.

Correction Tout d'abord nous fournissons un jeu de données d'entrées. De plus, l'application vient avec son propre système de vérification, qui est très rustique. Il consiste à comparer, une fois les sorties produites, leur contenu avec les sorties de référence, qui ont été obtenues avec notre version de l'application.

Du coup, le fait de disposer de Google Earth sur votre ordinateur n'est pas strictement nécessaire, on ne s'en sert pas à proprement parler pour l'exercice.

Mise en place

Partez d'un répertoire vierge Pour commencer, créez-vous un répertoire pour travailler à cet exercice.

Les données Commencez par y installer les données que nous publions dans les formats suivants :

— au format tar
— au format tar compressé
— au format zip

Une fois installées, ces données doivent se trouver dans un sous-répertoire `json/` qui contient 133 fichiers `*.json` :

— `json/2013-10-01-00-00--t=10--ext.json`
— `...`
— `json/2013-10-01-23-50--t=10.json`

Comme vous pouvez le deviner, il s'agit de données sur le mouvement des bateaux dans la zone en date du 10 Octobre 2013 ; et comme vous le devinez également, on a quelques exemplaires de données étendues, mais dans la plupart des cas il s'agit de données abrégées.

Les résultats de référence De même il vous faut installer les résultats de référence que vous trouvez ici :

— au format tar
— au format tar compressé (tgz)
— au format zip

Quel que soit le format choisi, une fois installé ceci doit vous donner trois fichiers :

— `ALL_SHIPS.kml.ref`
— `ALL_SHIPS.txt.ref`
— `ALL_SHIPS-v.txt.ref`

Le code Vous pouvez à présent aller chercher les 3 modules suivants :

— merger.py
— compare.py
— kml.py

et les sauver dans le même répertoire.

Vous remarquerez que le code est cette fois entièrement rédigé en anglais, ce que nous vous conseillons de faire aussi souvent que possible.

Votre but dans cet exercice est d'écrire le module manquant `shipdict.py` qui permettra à l'application de fonctionner comme attendu.

Fonctionnement de l'application

Comment est structurée l'application Le point d'entrée s'appelle `merger.py`
Il utilise trois modules annexes, qui sont :

— `shipdict.py`, qui implémente les classes
— `Position` qui contient une latitude, une longitude, et un timestamp
— `Ship` qui modélise un bateau à partir de son `id` et optionnellement `name` et `country`
— `ShipDict`, qui maintient un index des bateaux (essentiellement un dictionnaire)
— `compare.py` qui implémente
— la classe `Compare` qui se charge de comparer les fichiers résultat avec leur version de référence
— `kml.py` qui implémente
— la classe `KML` dans laquelle sont concentrées les fonctions liées à la génération de KML ; c'est notamment en fonction de nos objectifs pédagogiques que ce choix a été fait.

Lancement Lorsque le programme est complet et qu'il fonctionne correctement, on le lance comme ceci :

```
$ python3 merger.py json/*
Opening ALL_SHIPS.txt for listing all named ships
Opening ALL_SHIPS.kml for ship ALL_SHIPS
Comparing ALL_SHIPS.txt and ALL_SHIPS.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

qui comme on le voit produit :

— `ALL_SHIPS.txt` qui résume, par ordre alphabétique les bateaux qu'on a trouvés et le nombre de positions pour chacun, et
— `ALL_SHIPS.kml` qui est le fichier au format `KML` qui contient toutes les trajectoires.

Mode bavard (verbose) On peut également lancer l'application avec l'option `--verbose` ou simplement `-v` sur la ligne de commande, ce qui donne un résultat plus détaillé. Le code KML généré reste inchangé, mais la sortie sur le terminal et le fichier de résumé sont plus étoffés :

```
$ python merger.py --verbose json/*.json
Opening json/2013-10-01-00-00--t=10--ext.json for parsing JSON
Opening json/2013-10-01-00-10--t=10.json for parsing JSON
...
Opening json/2013-10-01-23-40--t=10.json for parsing JSON
Opening json/2013-10-01-23-50--t=10.json for parsing JSON
Opening ALL_SHIPS-v.txt for listing all named ships
Opening ALL_SHIPS.kml for ship ALL_SHIPS
Comparing ALL_SHIPS-v.txt and ALL_SHIPS-v.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

À noter que dans le mode bavard toutes les positions sont listées dans le résumé au format texte, ce qui le rend beaucoup plus bavard comme vous pouvez le voir en inspectant la taille des deux fichiers de référence :

```
$ ls -l ALL_SHIPS*txt.ref v2.0
-rw-r--r--  1 parmentelat  staff  1438681 Dec  4 16:20 ALL_SHIPS-v.txt.ref
-rw-r--r--  1 parmentelat  staff    15331 Dec  4 16:20 ALL_SHIPS.txt.ref
-rw-r--r--  1 parmentelat  staff        0 Dec  4 16:21 v2.0
```

merger.py --help

```
$ merger.py --help
usage: merger.py [-h] [-v] [-s SHIP_NAME] [-z] [inputs [inputs ...]]

positional arguments:
  inputs

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Verbose mode
  -s SHIP_NAME, --ship SHIP_NAME
                        Restrict to ships by that name
  -z, --gzip            Store kml output in gzip (KMZ) format
```

Un mot sur les données **Attention** que le contenu détaillé des champs `extended` et `abbreviated` peut être légèrement différent de ce qu'on avait pour les exercices de la semaine 3, dans lequel certaines simplifications ont été apportées.

Voici ce avec quoi on travaille cette fois-ci :

```
>>> extended[0]
[228317000, 48.76829, -4.334262, 75, 333, u'2013-09-30T21:54:00', u'MA GONDOLE', 30, 0, u'FGS
```

c'est-à-dire :

```
[ id, latitude, longitude, _, _, timestamp, name, _, _, _, country, ...]
```

et en ce qui concerne les données abrégées :

```
>>> abbreviated[0]
[232005670, 49.39331, -5.939922, 33, 269, 3, u'2013-10-01T06:08:00']
```

c'est-à-dire :

```
[ id, latitude, longitude, _, _, _, timestamp]
```

Il y a unicité des `id` bien entendu (deux relevés qui portent le même `id` concernent le même bateau).

Note historique Dans une première version de cet exercice, on avait laissé des doublons, c'est-à-dire des bateaux différents mais de même nom. Afin de rendre l'exercice plus facile à corriger (notamment parce que la comparaison des résultats repose sur l'ordre alphabétique), dans la présente version ces doublons ont été enlevés. Sachez toutefois que cette unicité est artificielle, aussi efforcez-vous de ne pas écrire de code qui reposerait sur cette hypothèse.

6.17.1 Niveaux pour l'exercice

Quelque soit le niveau auquel vous choisissez de faire l'exercice, nous vous conseillons de commencer par lire intégralement les 3 modules qui sont à votre disposition, dans l'ordre :

- `merger.py` qui est le chef d'orchestre de toute cette affaire ;
- `compare.py` qui est très simple ;
- `kml.py` qui ne présente pas grand intérêt en soi si ce n'est pour l'utilisation de la classe `string.Template` qui peut être utile dans d'autres contextes également.

En **niveau avancé**, l'énoncé pourrait s'arrêter là ; vous lisez le code qui est fourni et vous en déduisez ce qui manque pour faire fonctionner le tout. En cas de difficulté liée aux arrondis avec le mode bavard, vous pouvez toutefois vous inspirer du code qui est donné dans la toute dernière section de cet énoncé (section "Un dernier indice"), pour traduire un flottant en représentation textuelle.

Vous pouvez considérer que vous avez achevé l'exercice lorsque les deux appels suivants affichent les deux dernières lignes avec OK :

```
$ python merger.py json/*.json
...
Comparing ALL_SHIPS.txt and ALL_SHIPS.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

```
$ python merger.py -v json/*.json
...
Comparing ALL_SHIPS-v.txt and ALL_SHIPS-v.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

Le cas où on lance `merger.py` avec l'option bavarde est facultatif.

En **niveau intermédiaire**, nous vous donnons ci-dessous un extrait de ce que donne `help` sur les classes manquantes de manière à vous donner une indication de ce que vous devez écrire.

Classe `Position`

```
Help on class Position in module shipdict:

class Position(__builtin__.object)
 |  a position atom with timestamp attached
 |
 |  Methods defined here:
 |
 |  __init__(self, latitude, longitude, timestamp)
 |      constructor
 |
 |  __repr__(self)
 |      only used when merger.py is run in verbose mode
 |
```

Notes * certaines autres classes comme `KML` sont également suscpetibles d'accéder aux champs internes d'une instance de la classe `Position` en faisant simplement `position.latitude` * La classe `Position` redéfinit `__repr__`, ceci est utilisé uniquement dans la sortie en mode bavard.

Classe `Ship`

```
Help on class Ship in module shipdict:

class Ship(__builtin__.object)
 |  a ship object, that requires a ship id,
 |  and optionnally a ship name and country
 |  which can also be set later on
 |
 |  this object also manages a list of known positions
 |
 |  Methods defined here:
 |
 |  __init__(self, id, name=None, country=None)
 |      constructor
 |
 |  add_position(self, position)
 |      insert a position relating to this ship
 |      positions are not kept in order so you need
 |      to call `sort_positions` once you're done
 |
 |  sort_positions(self)
 |      sort list of positions by chronological order
```

Classe `Shipdict`

```
Help on class ShipDict in module shipdict:

class ShipDict(__builtin__.dict)
 |  a repository for storing all ships that we know about
 |  indexed by their id
 |
```

```
 |  Method resolution order:
 |      ShipDict
 |      __builtin__.dict
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      constructor
 |
 |  __repr__(self)
 |
 |  add_abbreviated(self, chunk)
 |      adds an abbreviated data chunk to the repository
 |
 |  add_chunk(self, chunk)
 |      chunk is a plain list coming from the JSON data
 |      and be either extended or abbreviated
 |
 |      based on the result of is_abbreviated(),
 |      gets sent to add_extended or add_abbreviated
 |
 |  add_extended(self, chunk)
 |      adds an extended data chunk to the repository
 |
 |  all_ships(self)
 |      returns a list of all ships known to us
 |
 |  clean_unnamed(self)
 |      Because we enter abbreviated and extended data
 |      in no particular order, and for any time period,
 |      we might have ship instances with no name attached
 |      This method removes such entries from the dict
 |
 |  is_abbreviated(self, chunk)
 |      depending on the size of the incoming data chunk,
 |      guess if it is an abbreviated or extended data
 |
 |  ships_by_name(self, name)
 |      returns a list of all known ships with name <name>
 |
 |  sort(self)
 |      makes sure all the ships have their positions
 |      sorted in chronological order
```

Un dernier indice Pour éviter de la confusion, voici le code que nous utilisons pour transformer un flottant en coordonnées lisibles dans le résumé généré en mode bavard.

```
def d_m_s(f):
    """
    make a float readable; e.g. transform 2.5 into 2.30'00''
```

```
we avoid using ° to keep things simple
input is assumed positive
"""
d = int (f)
m = int((f-d)*60)
s = int( (f-d)*3600 - 60*m)
return f"{d:02d}.{m:02d}'{s:02d}'"
```

SEMAINE 7

L'ÉCOSYSTÈME DATA SCIENCE PYTHON

7.1 Installations supplémentaires

7.1.1 Complément - niveau basique

Les outils que nous voyons cette semaine, bien que jouant un rôle majeur dans le succès de l'écosystème Python, **ne font pas** partie de la **distribution standard**. Cela signifie qu'il vous faut éventuellement procéder à des installations complémentaires sur votre ordinateur (évidemment vous pouvez utiliser les notebooks sans installation de votre part).

Comment savoir ?

Pour savoir si votre installation est idoine, vous devez pouvoir faire ceci :

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt

In [ ]: import pandas as pd
```

Avec (ana)conda

Si vous avez installé votre Python avec conda, selon toute probabilité toutes ces bibliothèques sont déjà accessibles pour vous, vous n'avez rien à faire de particulier pour pouvoir faire tourner les exemples du cours sur votre ordinateur.

Distribution standard

Si vous avez installé Python à partir d'une distribution standard, vous pouvez utiliser `pip` comme ceci ; naturellement ceci doit être fait **dans un terminal** et non pas dans l'interpréteur python, ni dans IDLE :

```
$ pip3 install numpy matplotlib pandas
```

Debian/Ubuntu

Si vous utilisez Debian ou Ubuntu, et que vous avez déjà installé Python avec `apt-get`, la méthode préconisée sera :

```
$ apt-get install python3-numpy python3-matplotlib python3-pandas
```

Fedora

De manière similaire sur Fedora ou RHEL :

```
$ dnf install python3-numpy python3-matplotlib python3-pandas
```

SEMAINE 8

PROGRAMMATION ASYNCHRONE - ASYNCIO

8.1 Essayez vous-même

8.1.1 Complément - niveau avancé

Pour des raisons techniques, il ne nous est pas possible de mettre en ligne un notebook qui vous permette de reproduire les exemples de la vidéo.

C'est pourquoi, si vous êtes intéressés à reproduire vous-même les expériences de la vidéo - à savoir, aller chercher plusieurs URLs de manière séquentielle ou en parallèle - vous pouvez télécharger le code fourni à ce lien.

Il s'agit d'un simple script, qui reprend les 3 approches de la vidéo :

— accès en séquence
— accès asynchrones avec `fetch`
— accès asynchrones avec `fetch2` (qui pour rappel provoque un tick à chaque ligne qui revient d'un des serveurs web).

À part pour l'appel à `sys.stdout.flush()`, ce code est rigoureusement identique à celui utilisé dans la vidéo. On doit faire ici cet appel à `flush()`, dans le mode avec `fetch2`, car sinon les sorties de notre script sont bufferisées, et apparaissent toutes ensemble à la fin du programme, c'est beaucoup moins drôle.

Voici son mode d'emploi :

```
$ python3 async_http.py --help
usage: async_http.py [-h] [-s] [-d] [urls [urls ...]]

positional arguments:
  urls              URL's to be fetched

optional arguments:
  -h, --help        show this help message and exit
  -s, --sequential  run sequentially
  -d, --details     show details of lines as they show up (using fetch2)
```

Et voici les chiffres que j'obtiens lorsque je l'utilise dans une configuration réseau plus stable que dans la vidéo (qui donne pour cette raison montre un résultat un peu décevant) :

```
$ python3 async_http.py -s
Running sequential mode on 4 URLs
http://www.irs.gov/pub/irs-pdf/f1040.pdf returned 179940 chars
http://www.irs.gov/pub/irs-pdf/f1040ez.pdf returned 113242 chars
http://www.irs.gov/pub/irs-pdf/f1040es.pdf returned 395201 chars
http://www.irs.gov/pub/irs-pdf/f1040sb.pdf returned 73189 chars
duration = 9.80829906463623s
```

```
$ python3 async_http.py
Running simple mode (fetch) on 4 URLs
fetching http://www.irs.gov/pub/irs-pdf/f1040.pdf
fetching http://www.irs.gov/pub/irs-pdf/f1040sb.pdf
fetching http://www.irs.gov/pub/irs-pdf/f1040es.pdf
fetching http://www.irs.gov/pub/irs-pdf/f1040ez.pdf
http://www.irs.gov/pub/irs-pdf/f1040.pdf response status 200
http://www.irs.gov/pub/irs-pdf/f1040ez.pdf response status 200
http://www.irs.gov/pub/irs-pdf/f1040sb.pdf response status 200
http://www.irs.gov/pub/irs-pdf/f1040es.pdf response status 200
http://www.irs.gov/pub/irs-pdf/f1040sb.pdf returned 75864 bytes
http://www.irs.gov/pub/irs-pdf/f1040.pdf returned 186928 bytes
http://www.irs.gov/pub/irs-pdf/f1040ez.pdf returned 117807 bytes
http://www.irs.gov/pub/irs-pdf/f1040es.pdf returned 409193 bytes
duration = 2.211031913757324s
```

N'hésitez pas à utiliser ceci comme base pour expérimenter.

Nous verrons en fin de semaine un autre exemple qui cette fois illustrera l'interaction avec les sous-processus.

8.2 `asyncio` - un exemple un peu plus réaliste

8.2.1 Complément - niveau avancé

Pour des raisons techniques, il n'est pas possible de mettre en ligne un notebook pour les activités liées au réseau, qui sont pourtant clairement dans le coeur de cible de la librairie - souvenez-vous que ce paradigme de programmation a été développé au départ par les projets comme tornado, qui se préoccupe de services Web.

Aussi, pour illustrer les possibilités offertes par `asyncio` sur un exemple un peu plus significatif que ceux qui utilisent `asyncio.sleep`, nous allons écrire le début d'une petite architecture de jeu.

Il s'agit pour nous principalement d'illustrer les capacités de `asyncio` en termes de gestion de sous-processus, car c'est quelque chose qu'on peut déployer dans le contexte des notebooks.

Nous allons procéder en deux temps. Dans ce premier notebook nous allons écrire un petit programme python qui s'appelle `players.py`. C'est une brique de base dans notre architecture, dans le second notebook on écrira un programme qui lance (sous la forme de sous-processes) plusieurs instances de `players.py`.

Le programme `players.py`

Mais dans l'immédiat, voyons ce que fait `players.py`. On veut modéliser le comportement de plusieurs joueurs.

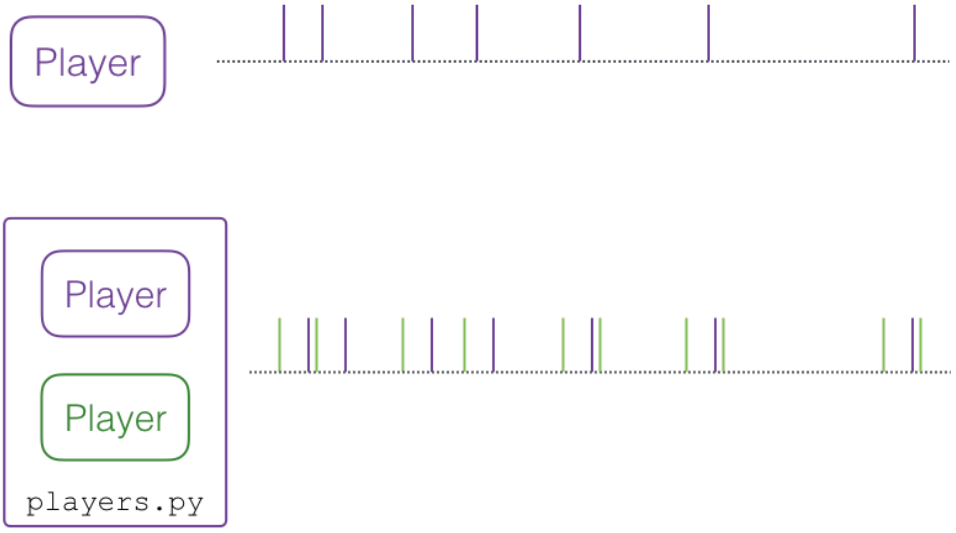
Chaque joueur a un comportement hyper basique, il émet simplement à des intervalles aléatoires un événement du type

> je suis le joueur John et je vais dans la direction Nord

Chaque joueur a un nom, et une fréquence moyenne, et un nombre de cycles.

Par ailleurs pour être un peu vraisemblable, il y a quatre directions `N`, `S`, `E` et `W`, mais que l'on n'utilisera pas vraiment dans la suite.

Voyez ici le code de `players.py`



Comme vous le voyez, dans ce premier exemple nous n'utilisons à nouveau que `asyncio.sleep` pour modéliser chaque joueur, dont la logique peut être illustrée simplement comme ceci (où la ligne horizontale représente le temps) :

Pour éviter de nous noyer dans des configurations compliquées, on a embarqué dans `players` plusieurs configurations prédéfinies, mais dans tous les cas chacune de ces configurations crée deux joueurs.

La logique des deux joueurs est simplement juxtaposée, ou si on préfère superposée, par `asyncio.gather`, ce qui fait que la sortie de `players.py` ressemble à ceci :

```
In [ ]: # je peux lancer un sous-processus
        # depuis le notebook
        !data/players.py

In [ ]: # ou une autre configuration
        !data/players.py 2
```

Nous allons voir dans le notebook suivant comment on peut orchestrer plusieurs instances du programme `players.py`, et prolonger cette logique de juxtaposition / mélange des sorties, mais cette fois au niveau de plusieurs processus.

8.3 Gestion de sous-process

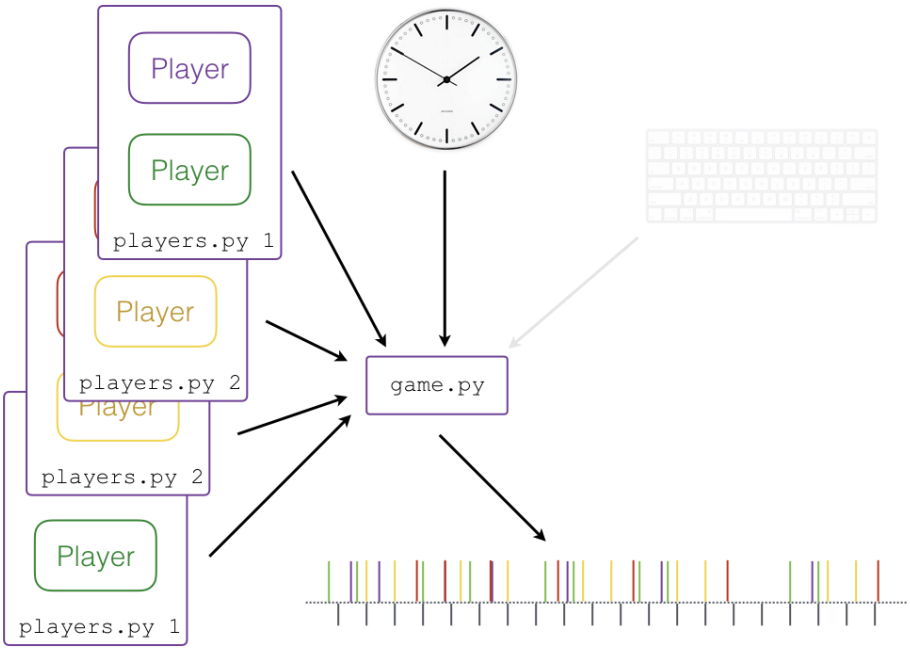
8.3.1 Complément - niveau (très) avancé

Dans ce second notebook, nous allons étudier un deuxième programme python, que j'appelle `game.py` (en fait c'est le présent notebook).

Fonctions de `game.py`

Son travail va consister à faire plusieurs choses en même temps ; pour rester le plus simple possible, on va se contenter des trois fonctions suivantes :

— *scheduler* (chef d'orchestre) : on veut lancer à des moments préprogrammés des instances (sous-processes) de `players.py` ;



— *multiplexer* (agrégateur) : on veut lire et imprimer au fur et à mesure les messages émis par les sous-processes ;

— horloge : on veut également afficher chaque seconde le temps écoulé depuis le début.

En pratique, le programme `game.py` serait plutôt le serveur du jeu qui reçoit les mouvements de tous les joueurs, et diffuse ensuite en retour, en mode broadcast, un état du jeu à tous les participants.

Mais dans notre version hyper simpliste, ça donne un comportement que j'ai essayé d'illustrer comme ceci :

Remarque concernant les notebooks et le clavier Lorsqu'on exécute du code python dans un notebook, les entrées clavier sont en fait interceptées par le browser web ; du coup on ne peut pas facilement (du tout ?) faire tourner dans un notebook un programme asynchrone qui réagirait aussi aux événements de type entrée clavier.

C'est pour cette raison que le clavier apparaît sur ma figure en filigrane. Si vous allez jusqu'à exécuter ce notebook localement sur votre machine (voir plus bas), vous pourrez utiliser le clavier pour ajouter à la volée des éléments dans le scénario - en entrant des numéros de 1 à 4 au moment voulu.

Terminaison Pour rester simple et en l'absence de clavier, j'ai choisi de terminer le programme lorsque le dernier sous-processus se termine.

Le programme `game.py`

C'est ce notebook qui va jouer pour nous le rôle du programme `game.py`.

```
In [ ]: import asyncio
        import sys
```

```
In [ ]: # cette constante est utile pour déclarer qu'on a l'intention
        # de lire les sorties (stout et stderr)
        # de nos sous-process par l'intermédiaire de pipes
        from subprocess import PIPE
```

Commençons par la classe `Scheduler` ; c'est celle qui va se charger de lancer les sous-processes selon un scénario. Pour ne pas se compliquer la vie on choisit de représenter un scénario (un script) comme une liste de tuples de la forme

`script = [ (secondes, predef), ...]`

qui signifie de lancer, un délai de `secondes` secondes après le début du programme, le programme `players.py` dans la configuration `predef` - de 1 à 4 donc.

```
In [ ]: class Scheduler:

            def __init__(self, script):

                # on trie le script par ordre chonologique
                self.script = list(script)
                self.script.sort(key = lambda time_predef : time_predef[0])

                # juste pour donner un numéro à chaque process
                self.counter = 1
                # combien de process sont actifs
                self.running = 0

            async def run(self):
                """
                fait tout le travail, c'est-à-dire
                * lance tous les sous-process à l'heure indiquée
                * et aussi en préambule, pour le mode avec clavier,
                  arme une callback sur l'entrée standard
                """
                # pour le mode avec clavier (pas fonctionnel dans le notebook)
                # on arme une callback sur stdin
                asyncio.get_event_loop().add_reader(
                    # il nous faut un file descriptor, pas un objet python
                    sys.stdin.fileno(),
                    # la callback
                    Scheduler.read_keyboard_line,
                    # les arguments de la callback
                    # cette fois c'est un objet python
                    self, sys.stdin
                )
                # le scénario prédéfini
                epoch = 0
                for tick, predef in self.script:
                    # attendre le bon moment
                    await asyncio.sleep(tick - epoch)
                    # pour le prochain
                    epoch = tick
                    asyncio.ensure_future(self.fork_players(predef))
```

```
async def fork_players(self, predef):
    """
    lance maintenant une instance de players avec cette config

    puis
    écoute à la fois sdtout et stderr, et les imprime
    (bon c'est vrai que players n'écrit rien sur stderr)
    attend la fin du sous-process (avec wait())
    et retourne son code de retour (exitcode) du sous-process

    par commodité on décide d'arrêter la boucle principale
    lorsqu'il n'y a plus aucun process actif
    """

    # la commande à lancer pour forker une instance de players.py
    command = f"python3 -u data/players.py {predef}".split()
    # pour afficher un nom un peu plus parlant
    worker = f"ps#{self.counter} (predef {predef})"
    # housekeeping
    self.counter += 1
    self.running += 1
    # c'est là que ça se passe : on forke
    print(8 * '>', f"worker {worker}")
    process = await asyncio.create_subprocess_exec(
        *command,
        stdout=PIPE, stderr=PIPE,
    )
    # et on lit et écrit les canaux du sous-process
    stdout, stderr = await asyncio.gather(
        self.read_and_display(process.stdout, worker),
        self.read_and_display(process.stderr, worker))
    # qu'il ne faut pas oublier d'attendre pour que l'OS sache
    # qu'il peut nettoyer
    retcod = await process.wait()
    # le process est terminé
    self.running -= 1
    print(8 * '<', f"worker {worker} - exit code {retcod}"
          f" - {self.running} still running")
    # si c'était le dernier on sort de la boucle principale
    if self.running == 0:
        print("no process left - bye")
        asyncio.get_event_loop().stop()
    # sinon on retourne le code de retour
    return retcod

async def read_and_display(self, stream, worker):
    """
    une coroutine pour afficher les sorties d'un canal
    stdout ou stderr d'un sous-process
    retourne lorsque le process est terminé
    """
```

```
            while True:
                bytes = await stream.readline()
                # l'OS nous signale qu'on en a terminé
                # avec ce process en renvoyant ici un objet bytes vide
                if not bytes:
                    break
                # ici on se contente d'imprimer, du coup
                # il faut convertir en str (bien qu'ici
                # players n'écrit que de l'ASCII)
                line = bytes.decode().strip()
                print(8 * ' ', f"got `{line}` from {worker}")

        # ceci est seulement fonctionnel si vous exécutez
        # le programme localement sur votre ordinateur
        # car depuis un notebook le clavier est intercepté
        # par le serveur web
        def read_keyboard_line(self, stdin):
            """
            ceci est une callback; eh oui :)
            c'est pourquoi d'ailleurs ce n'est pas une coroutine
            cependant on est sûr qu'elle n'est appelée
            que lorsqu'il y a réellement quelque chose à lire
            """
            line = stdin.readline().strip()
            # ici je triche complètement
            # lorsqu'on est dans un notebook, pour bien faire
            # on ne devrait pas regarder stdin du tout
            # mais pour garder le code le plus simple possible
            # je choisis d'ignorer les lignes vides ici
            # comme ça mon code marche dans les deux cas
            if not line:
                return
            # on traduit la ligne tapée au clavier
            # en un entier entre 1 et 4
            try:
                predef = int(line)
                if not (1 <= predef <= 4):
                    raise ValueError('entre 1 et 4')
            except Exception as e:
                print(f"{line} doit être entre 1 et 4 {type(e)} - {e}")
                return
            asyncio.ensure_future(self.fork_players(predef))
```

À ce stade on a déjà le cœur de la logique du *scheduler*, et aussi du multiplexer. Il ne nous manque plus que l'horloge :

```
In [ ]: class Clock:

        def __init__(self):
            self.clock_seconds = 0

        async def run(self):
```

```
        while True:
            print(f"clock = {self.clock_seconds:04d}s")
            await asyncio.sleep(1)
            self.clock_seconds += 1
```

Et enfin pour mettre tous ces morceaux en route il nous faut une boucle d'événements :

```
In [ ]: class Game:

    def __init__(self, script):
        self.script = script

    def mainloop(self):
        loop = asyncio.get_event_loop()

        clock = Clock()
        asyncio.ensure_future(clock.run())

        scheduler = Scheduler(self.script)
        asyncio.ensure_future(scheduler.run())
        loop.run_forever()
```

Et maintenant je peux lancer une session simple ; pour ne pas être noyé par les sorties on va se contenter de lancer :

— 0.5 seconde après le début une instance de `players.py 1`
— 1 seconde après le début une instance de `players.py 2`

```
In [ ]: game = Game( [(0.5, 1), (1., 2)])
        game.mainloop()
```

Conclusion

Notre but avec cet exemple est de vous montrer, après les exemples des vidéos qui reposent en grande majorité sur `asyncio.sleep`, que la boucle d'événements de `asyncio` permet d'avoir accès, de manière simple et efficace, à des événements de niveau OS. Dans un complément précédent nous avions aperçu la gestion de requêtes HTTP ; ici nous avons illustré la gestion de sous-process.

Actuellement on peut trouver des bibliothèques au dessus de `asyncio` pour manipuler de cette façon quasiment tous les protocoles réseau, et autres accès à des bases de données.

Exécution en local

Si vous voulez exécuter ce code localement sur votre machine :

Tout d'abord sachez que je n'ai pas du tout essayé ceci sur un OS Windows - et d'ailleurs ça m'intéresserait assez de savoir si ça fonctionne ou pas.

Cela étant dit, il vous suffit alors de télécharger le présent notebook au format python. Vous aurez aussi besoin :

— du code de `players.py`, évidemment ;
— et de modifier le fichier téléchargé pour lancer `players.py` au lieu de `data/players.py`, qui ne fait de sens probablement que sur le serveur de notebooks.

Comme on l'a indiqué plus haut, si vous l'exécutez en local vous pourrez cette fois interagir aussi via la clavier, et ajouter à la volée des sous-process qui n'étaient pas prévus initialement dans le scénario.

8.4 Pour aller plus loin

Je vous signale enfin, si vous êtes intéressés à creuser encore davantage, ce tutorial intéressant qui implémente un jeu complet.

Naturellement ce tutorial est lui basé sur du code réseau et non, comme nous y sommes contraints, sur une architecture de type sous-process ; le jeu en question est même en ligne ici...

SEMAINE 9

SUJETS AVANCÉS

9.1 Décorateurs

9.1.1 Complément - niveau (très) avancé

Le mécanisme des décorateurs - qui rappelle un peu, pour ceux qui connaissent, les macros Lisp - est un mécanisme très puissant. Sa portée va bien au delà de simplement rajouter du code avant et après une fonction, comme dans le cas de `NbAppels` que nous avons vu dans la vidéo.

Par exemple, les notions de méthodes de classe (`@classmethod`) et de méthodes statiques (`@staticmethod`) sont implémentées comme des décorateurs. Pour une liste plus représentative de ce qu'il est possible de faire avec les décorateurs, je vous invite à parcourir même rapidement ce recueil de décorateurs qui propose du code (à titre indicatif, car rien de ceci ne fait partie de la librairie standard) pour des thèmes qui sont propices à la décoration de code.

Nous allons voir en détails quelques-uns de ces exemples.

Un décorateur implémenté comme une classe

Dans la vidéo on a vu `NbAppels` pour compter le nombre de fois qu'on appelle une fonction. Pour mémoire on avait écrit :

```
In [ ]: # un rappel du code montré dans la vidéo
        class NbAppels(object):
            def __init__(self, f):
                self.f = f
                self.appels = 0
            def __call__(self, *args):
                self.appels += 1
                print(f"{self.appels}-ème appel à {self.f.__name__}")
                return self.f(*args)
```

```
In [ ]: # nous utilisons ici une implémentation en log(n)
        # de la fonction de fibonacci

        @NbAppels
        def fibo_aux(n):
            "Fibonacci en log(n)"
            if n < 1:
                return 0, 1
            u, v = fibo_aux(n//2)
            u, v = u * (2 * v - u), u*u + v*v
```

```
            if n % 2 == 1:
                return v, u + v
            else:
                return u, v

        def fibo_log(n):
            return fibo_aux(n)[0]
```

```
In [ ]: # pour se convaincre que nous sommes bien en log2(n)
        from math import log
```

```
In [ ]: n1 = 100

        log(n1)/log(2)
```

```
In [ ]: fibo_log(n1)
```

```
In [ ]: # on multiplie par 2**4 = 16,
        # donc on doit voir 4 appels de plus
        n2 = 1600

        log(n2)/log(2)
```

```
In [ ]: fibo_log(n2)
```

`memoize` implémenté comme une fonction

Ici nous allons implémenter `memoize`, un décorateur qui permet de mémoriser les résultats d'une fonction, et de les cacher pour ne pas avoir à les recalculer la fois suivante.

Alors que `NbAppels` était **implémenté comme une classe**, pour varier un peu, nous allons implémenter cette fois `memoize` **comme une vraie fonction**, pour vous montrer les deux alternatives que l'on a quand on veut implémenter un décorateur : une vraie fonction ou une classe de callables.

Le code du décorateur

```
In [ ]: # une première implémentation de memoize

        # un décorateur de fonction
        # implémenté comme une fonction
        def memoize(a_decorer):
            """
            Un décorateur pour conserver les résultats
            précédents et éviter de les recalculer
            """
            def decoree(*args):
                # si on a déjà calculé le résultat
                # on le renvoie
                try:
                    return decoree.cache[args]
                # si les arguments ne sont pas hashables,
                # par exemple s'ils contiennent une liste
                # on ne peut pas cacher et on reçoit TypeError
```

```
            except TypeError:
                return a_decorer(*args)
            # les arguments sont hashables mais on
            # n'a pas encore calculé cette valeur
            except KeyError:
                # on fait vraiment le calcul
                result = a_decorer(*args)
                # on le range dans le cache
                decoree.cache[args] = result
                # on le retourne
                return result
        # on initialise l'attribut 'cache'
        decoree.cache = {}
        return decoree
```

Comment l'utiliser Avant de rentrer dans le détail du code, voyons comment cela s'utiliserait ; il n'y a pas de changement de ce point de vue par rapport à l'option développée dans la vidéo :

```
In [ ]: # créer une fonction décorée
        @memoize
        def fibo_cache(n):
            """
            Un fibonacci hyper-lent (exponentiel) se transforme
            en temps linéaire une fois que les résultats sont cachés
            """
            return n if n <= 1 else fibo_cache(n-1) + fibo_cache(n-2)
```

Bien que l'implémentation utilise un algorithme épouvantablement lent, le fait de lui rajouter du caching redonne à l'ensemble un caractère linéaire.

En effet, si vous y réfléchissez une minute, vous verrez qu'avec le cache, lorsqu'on calcule `fibo_cache(n)`, on calcule d'abord `fibo_cache(n-1)`, puis lorsqu'on évalue `fibo_cache(n-2)` le résultat **est déjà dans le cache** si bien qu'on peut considérer ce deuxième calcul comme, sinon instantané, du moins du même ordre de grandeur qu'une addition.

On peut calculer par exemple :

```
In [ ]: fibo_cache(300)
```

qu'il serait hors de question de calculer sans le caching.

On peut naturellement inspecter le cache, qui est rangé dans l'attribut `cache` de l'objet fonction lui-même :

```
In [ ]: len(fibo_cache.cache)
```

et voir que, comme on aurait pu le prédire, on a calculé et mémorisé les 301 premiers résultats, pour n allant de 0 à 300.

Comment ça marche ? On l'a vu dans la vidéo avec `NbAppels`, tout se passe exactement comme si on avait écrit :

```
def fibo_cache(n):
    <le code>

fibo_cache = memoize(fibo_cache)
```

Donc `memoize` est une fonction qui prend en argument une fonction `a_decorer` qui ici vaut `fibo_cache`, et retourne une autre fonction, `decoree` ; on s'arrange naturellement pour que `decoree` retourne le même résultat que `a_decorer`, avec seulement des choses supplémentaires.

Les points clés de l'implémentation sont les suivants. * On attache à l'objet fonction `decoree`, sous la forme d'un attribut`cache`, un dictionnaire qui va nous permettre de retrouver les valeurs déjà calculées, à partir d'un hash des arguments. * On ne peut pas cacher le résultat d'un objet qui ne serait pas globalement immuable ; or si on essaie on reçoit l'exception `TypeError`, et dans ce cas on recalcule toujours le résultat. C'est de toute façons plus sûr. * Si on ne trouve pas les arguments dans le cache, on reçoit l'exception `KeyError`, dans ce cas on calcule le résultat, et on le retourne après l'avoir rangé dans le cache. * Vous remarquerez aussi qu'on initialise l'attribut `cache` dans l'objet `decoree` à l'appel du décorateur (une seule fois, juste après avoir défini la fonction), et non pas dans le code de `decoree` qui lui est évalué à chaque appel.

Cette implémentation, sans être parfaite, est tout à fait utilisable dans un environnement réel, modulo les remarques de bon sens suivantes : * évidemment l'approche ne fonctionne que pour des fonctions déterministes ; s'il y a de l'aléatoire dans la logique de la fonction, il ne faut pas utiliser ce décorateur ; * tout aussi évidemment, la consommation mémoire peut être importante si on applique le caching sans discrimination ; * enfin en l'état la fonction decorée ne peut pas être appelée avec des arguments nommés ; en effet on utilise le tuple `args` comme clé pour retrouver dans le cache la valeur associée aux arguments.

Décorateurs, *docstring* et `help`

En fait, avec cette implémentation, il reste aussi un petit souci :

```
In [ ]: help(fibo_cache)
```

Et ce n'est pas exactement ce qu'on veut ; ce qui se passe ici c'est que `help` utilise les attributs `__doc__` et `__name__` de l'objet qu'on lui passe. Et dans notre cas `fibo_cache` est une fonction qui a été créée par l'instruction :

```
def decoree(*args):
    # etc.
```

Pour arranger ça et faire en sorte que `help` nous affiche ce qu'on veut, il faut s'occuper de ces deux attributs. Et plutôt que de faire ça à la main, il existe un utilitaire `functools.wraps`, qui fait tout le travail nécessaire. Ce qui nous donne une deuxième version de ce décorateur, avec deux lignes supplémentaires signalées par des `+++` ; :

```
In [ ]: # une deuxième implémentation de memoize, avec la doc

        import functools                                    # +++

        # un decorateur de fonction
        # implémenté comme une fonction
        def memoize(a_decorer):
            """
            Un décorateur pour conserver les résultats
            précédents et éviter de les recalculer
            """
            # on décore la fonction pour qu'elle ait les
            # propriétés de a_decorer : __doc__ et __name__
```

```
            @functools.wraps(a_decorer)                    # +++
            def decoree (*args):
                # si on a déjà calculé le résultat
                # on le renvoie
                try:
                    return decoree.cache[args]
                # si les arguments ne sont pas hashables,
                # par exemple une liste, on ne peut pas cacher
                # et on reçoit TypeError
                except TypeError:
                    return a_decorer(*args)
                # les arguments sont hashables mais on
                # n'a pas encore calculé cette valeur
                except KeyError:
                    # on fait vraiment le calcul
                    result = a_decorer(*args)
                    # on le range dans le cache
                    decoree.cache[args] = result
                    # on le retourne
                    return result
            # on initialise l'attribut 'cache'
            decoree.cache = {}
            return decoree

In [ ]: # créer une fonction décorée
        @memoize
        def fibo_cache2(n):
            """
            Un fibonacci hyper-lent (exponentiel) se transforme
            en temps linéaire une fois que les résultats sont cachés
            """
            return n if n <= 1 else fibo_cache2(n-1) + fibo_cache2(n-2)
```

Et on obtient à présent une aide en ligne cohérente :

```
In [ ]: help(fibo_cache2)
```

On peut décorer les classes aussi

De la même façon qu'on peut décorer une fonction, on peut décorer une classe.

Pour ne pas alourdir le complément, et aussi parce que le mécanisme de métaclasse offre une autre alternative qui est souvent plus pertinente, nous ne donnons pas d'exemple ici, cela vous est laissé à titre d'exercice si vous êtes intéressé.

Un décorateur peut lui-même avoir des arguments

Reprenons l'exemple de `memoize`, mais imaginons qu'on veuille ajouter un trait de "durée de validité du cache". Le code du décorateur a besoin de connaître la durée pendant laquelle on doit garder les résultats dans le cache.

On veut pouvoir préciser ce paramètre, appelons le `cache_timeout`, pour chaque fonction ; par exemple on voudrait écrire quelque chose comme

```
@memoize_expire(600)
def resolve_host(hostname):
    ...

@memoize_expire(3600*24)
def network_neighbours(hostname):
    ...
```

Ceci est possible également avec les décorateurs, avec cette syntaxe précisément. Le modèle qu'il faut avoir à l'esprit pour bien comprendre le code qui suit est le suivant et se base sur deux objets : * le premier objet, `memoize_expire`, est ce qu'on appelle une *factory* à décorateurs, c'est-à-dire que l'interpréteur va d'abord appeler `memoize_expire(600)` qui doit retourner un décorateur ; * le deuxième objet est ce décorateur retourné par `memoize_expire(600)` qui lui même doit se comporter commes les décorateurs sans argument que l'on a vus jusqu'ici.

Pour faire court, cela signifie que l'interpréteur fera

```
resolve_host = (memoize_expire(600))(resolve_host)
```

Ou encore si vous préférez

```
memoize = memoize_expire(600)
resolve_host = memoize(resolve_host)
```

Ce qui nous mène au code suivant :

```
In [ ]: import time

        # comme pour memoize, on est limité ici et on ne peut pas
        # supporter les appels à la **kwds, voir plus haut
        # la discussion sur l'implémentation de memoize

        # memoize_expire est une factory à décorateur
        def memoize_expire(timeout):

            # memoize_expire va retourner un decorateur sans argument
            # c'est à dire un objet qui se comporte
            # comme notre tout premier `memoize`
            def memoize(a_decorer):
                # à partir d'ici on fait un peu comme dans
                # la premiere version de memoize
                def decoree(*args):
                    try:
                        # sauf que disons qu'on met dans le cache un tuple
                        # (valeur, timestamp)
                        valeur, timestamp = decoree.cache[args]
                        # et la on peut acceder a timeout
                        # parce que la liaison en python est lexicale
                        if (time.time()-timestamp) <= timeout:
                            return valeur
                        else:
                            # on fait comme si on ne connaissait pas,
                            raise KeyError
```

```
                # si les arguments ne sont pas hashables,
                # par exemple une liste, on ne peut pas cacher
                # et on reçoit TypeError
                except TypeError:
                    return a_decorer(*args)
                # les arguments sont hashables mais on
                # n'a pas encore calculé cette valeur
                except KeyError:
                    result = a_decorer(*args)
                    decoree.cache[args] = (result, time.time())
                    return result

            decoree.cache = {}
            return decoree
        # le retour de memoize_expire, c'est memoize
        return memoize
```

```
In [ ]: @memoize_expire(0.5)
        def fibo_cache_expire(n):
            return n if n<=1 else fibo_cache_expire(n-2)+fibo_cache_expire(n-1)
```

```
In [ ]: fibo_cache_expire(300)
```

```
In [ ]: fibo_cache_expire.cache[(200,)]
```

Remarquez la clôture Pour conclure sur cet exemple, vous remarquez que dans le code de `decoree` on accède à la variable `timeout`. Ça peut paraître un peu étonnant, si vous pensez que `decoree` est appelée **bien après** que la fonction `memoize_expire` ait fini son travail. En effet, `memoize_expire` est évaluée **une fois** juste après **la définition** de `fibo_cache`. Et donc on pourrait penser que la valeur de `timeout` ne serait plus disponible dans le contexte de `decoree`.

Pour comprendre ce qui se passe, il faut se souvenir que python est un langage à liaison lexicale. Cela signifie que la *résolution* de la variable `timeout` se fait au moment de la compilation (de la production du byte-code), et non au moment où est appelé `decoree`.

Ce type de construction s'appelle une **clotûre**, en référence au lambda calcul : on parle de terme clos lorsqu'il n'y a plus de référence non résolue dans une expression. C'est une technique de programmation très répandue notamment dans les applications réactives, où on programme beaucoup avec des *callbacks* ; par exemple il est presque impossible de programmer en JavaScript sans écrire une clôture.

On peut chaîner les décorateurs

Pour revenir à notre sujet, signalons enfin que l'on peut aussi "chaîner les décorateurs" ; imaginons par exemple qu'on dispose d'un décorateur `add_field` qui ajoute dans une classe un *getter* et un *setter* basés sur un nom d'attribut.

C'est-à-dire que

```
@add_field('name')
class Foo:
    pass
```

donnerait pour `Foo` une classe qui dispose des méthodes `get_name` et `set_name` (exercice pour les courageux : écrire `add_field`).

Alors la syntaxe des décorateurs vous permet de faire quelque chose comme :

```
@add_field('name')
@add_field('address')
class Foo:
    pass
```

Ce qui revient à faire :

```
class Foo: pass
Foo = (add_field('address'))(Foo)
Foo = (add_field('name'))(Foo)
```

Discussion

Dans la pratique, écrire un décorateur est un exercice assez délicat. Le vrai problème est bien souvent la création d'objets supplémentaires : on n'appelle plus la fonction de départ mais un wrapper autour de la fonction de départ.

Ceci a tout un tas de conséquences, et le lecteur attentif aura par exemple remarqué : * que dans l'état du code de `singleton`, bien que l'on ait correctement mis à jour `__doc__` et `__name__` sur la classe décorée, `help(Spam)` ne renvoie pas le texte attendu, il semble que `help` sur une instance de classe ne se comporte pas exactement comme attendu ; * que si on essaie de combiner les décorateurs `NbAppels` et `memoize` sur une - encore nouvelle - version de fibonacci, le code obtenu ne converge pas ; en fait les technique que nous avons utilisées dans les deux cas ne sont pas compatibles entre elles.

De manière plus générale, il y a des gens pour trouver des défauts à ce système de décorateurs ; je vous renvoie notamment à ce blog qui, pour résumer, insiste sur le fait que les objets décorés n'ont **pas exactement** les mêmes propriétés que les objets originaux. L'auteur y explique que lorsqu'on fait de l'introspection profonde - c'est-à-dire lorsqu'on écrit du code qui "fouille" dans les objets qui représentent le code lui-même - les objets décorés ont parfois du mal à se *faire passer* pour les objets qu'ils remplacent.

À chacun de voir les avantages et les inconvénients de cette technique. C'est là encore beaucoup une question de goût. Dans certains cas simples, comme par exemple pour `NbAppels`, la décoration revient à simplement ajouter du code avant et après l'appel à la fonction à décorer. Et dans ce cas, vous remarquerez qu'on peut aussi faire le même genre de choses avec un *context manager* (je laisse ça en exercice aux étudiants intéressés).

Ce qui est clair toutefois est que la technique des décorateurs est quelque chose qui peut être très utile, mais dont il ne faut pas abuser. En particulier de notre point de vue, la possibilité de combiner les décorateurs, si elle existe bien dans le langage d'un point de vue syntaxique, est dans la pratique à utiliser avec la plus extrême prudence.

Pour en savoir plus

Maintenant que vous savez presque tout sur les décorateurs, vous pouvez retourner lire ce recueil de décorateurs mais plus en détails.

CORRIGÉS SEMAINE 2 À 6

9.2 Corrigé semaine 2 : Notions de base pour écrire son premier programme en Python

9.2.1 pythonid (regexp) - Semaine 2 Séquence 2

```
# un identificateur commence par une lettre ou un underscore
# et peut être suivi par n'importe quel nombre de
# lettre, chiffre ou underscore, ce qui se trouve être \w
# si on ne se met pas en mode unicode
pythonid = "[a-zA-Z_]\w*"
```

9.2.2 pythonid (bis) - Semaine 2 Séquence 2

```
# on peut aussi bien sûr l'écrire en clair
pythonid_bis = "[a-zA-Z_][a-zA-Z0-9_]*"
```

9.2.3 agenda (regexp) - Semaine 2 Séquence 2

```
# l'exercice est basé sur re.match, ce qui signifie que
# le match est cherché au début de la chaine
# MAIS il nous faut bien mettre \Z à la fin de notre regexp,
# sinon par exemple avec la cinquième entrée le nom 'Du Pré'
# sera reconnu partiellement comme simplement 'Du'
# au lieu d'être rejeté à cause de l'espace
# du coup pensez à bien toujours définir
# vos regexps avec des raw-strings
# remarquez sinon l'utilisation à la fin de :? pour signifier qu'on peut
# mettre ou non un deuxième séparateur ':'
#
agenda = r"\A(?P<prenom>[-\w]*):(?P<nom>[-\w]+):?\Z"
```

9.2.4 phone (regexp) - Semaine 2 Séquence 2

```
# idem concernant le \Z final
# il faut bien backslasher le + dans le +33
# car sinon cela veut dire 'un ou plusieurs'
phone = r"(\+33|0)(?P<number>[0-9]{9})\Z"
```

9.2.5 url (regexp) - Semaine 2 Séquence 2

```
# en ignorant la casse on pourra ne mentionner les noms de protocoles
# qu'en minuscules
i_flag = "(?i)"
# pour élaborer la chaine (proto1|proto2|...)
protos_list = ['http', 'https', 'ftp', 'ssh', ]
protos      = "(?P<proto>" + "|".join(protos_list) + ")"
# à l'intérieur de la zone 'user/password', la partie
# password est optionnelle - mais on ne veut pas le ':' dans
# le groupe 'password' - il nous faut deux groupes
password    = r"(:(?P<password>[^:]+))?"
# la partie user-password elle-même est optionnelle
# on utilise ici un raw f-string avec le préfixe rf
# pour insérer la regexp <password> dans la regexp <user>
user        = rf"((?P<user>\w+){password}@)?"
# pour le hostname on accepte des lettres, chiffres, underscore et '.'
# attention à backslaher . car sinon ceci va matcher tout y compris /
hostname    = r"(?P<hostname>[\w\.]+)"
# le port est optionnel
port        = r"(:(?P<port>\d+))?"
# après le premier slash
path        = r"(?P<path>.*)"
# on assemble le tout
url = i_flag + protos + "://" + user + hostname + port + '/' + path
```

9.2.6 label - Semaine 2 Séquence 6

```
def label(prenom, note):
    if note < 10:
        return f"{prenom} est recalé"
    elif note < 16:
        return f"{prenom} est reçu"
    else:
        return f"félicitations à {prenom}"
```

9.2.7 label (bis) - Semaine 2 Séquence 6

```
def label_bis(prenom, note):
    if note < 10:
        return f"{prenom} est recalé"
    # on n'en a pas vraiment besoin ici, mais
    # juste pour illustrer cette construction
    elif 10 <= note < 16:
        return f"{prenom} est reçu"
    else:
        return f"félicitations à {prenom}"
```

9.2.8 label (ter) - Semaine 2 Séquence 6

```
# on n'a pas encore vu l'expression conditionnelle
# et dans ce cas précis ce n'est pas forcément une
```

```
# idée géniale, mais pour votre curiosité on peut aussi
# faire comme ceci
def label_ter(prenom, note):
    return f"{prenom} est recalé" if note < 10 \
    else f"{prenom} est reçu" if 10 <= note < 16 \
    else f"félicitations à {prenom}"
```

9.2.9 inconnue - Semaine 2 Séquence 6

```
# pour enlever à gauche et à droite une chaine de longueur x
# on peut faire composite[ x : -x ]
# or ici x vaut len(connue)
def inconnue(composite, connue):
    return composite[ len(connue) : -len(connue) ]
```

9.2.10 inconnue (bis) - Semaine 2 Séquence 6

```
# ce qui peut aussi s'écrire comme ceci si on préfère
def inconnue_bis(composite, connue):
    return composite[ len(connue) : len(composite)-len(connue) ]
```

9.2.11 laccess - Semaine 2 Séquence 6

```
def laccess(liste):
    """
    retourne un élément de la liste selon la taille
    """
    # si la liste est vide il n'y a rien à faire
    if not liste:
        return
    # si la liste est de taille paire
    if len(liste) % 2 == 0:
        return liste[-1]
    else:
        return liste[len(liste)//2]
```

9.2.12 laccess (bis) - Semaine 2 Séquence 6

```
# une autre version qui utilise
# un trait qu'on n'a pas encore vu
def laccess(liste):
    # si la liste est vide il n'y a rien à faire
    if not liste:
        return
    # l'index à utiliser selon la taille
    index = -1 if len(liste) % 2 == 0 else len(liste) // 2
    return liste[index]
```

9.2.13 divisible - Semaine 2 Séquence 6

```
def divisible(a, b):
    "renvoie True si un des deux arguments divise l'autre"
```

```
        # b divise a si et seulement si le reste
        # de la division de a par b est nul
        if a % b == 0:
            return True
        # et il faut regarder aussi si a divise b
        if b % a == 0:
            return True
        return False
```

9.2.14 divisible (bis) - Semaine 2 Séquence 6

```
    def divisible_bis(a, b):
        "renvoie True si un des deux arguments divise l'autre"
        # on n'a pas encore vu les opérateurs logiques, mais
        # on peut aussi faire tout simplement comme ça
        # sans faire de if du tout
        return a % b == 0 or b % a == 0
```