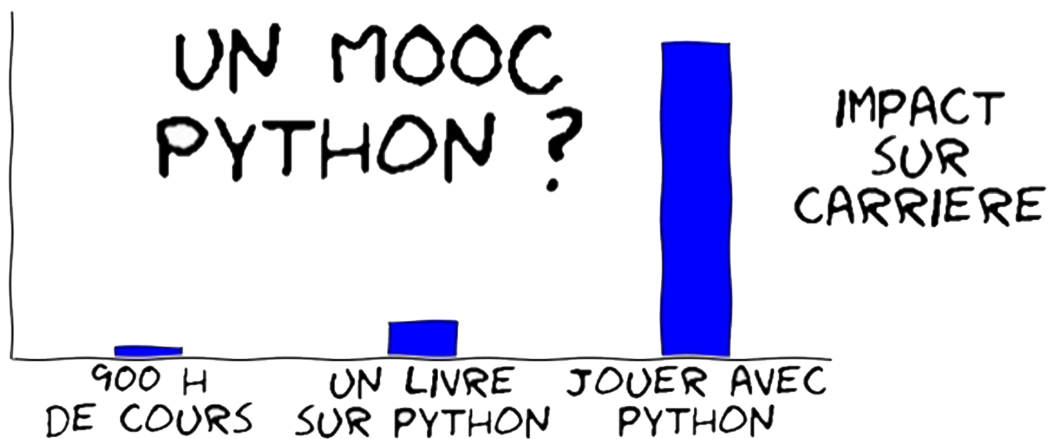




Des fondamentaux au concepts avancés du langage
Session 2 - 17 septembre 2018

Thierry Parmentelat

Arnaud Legout



<https://www.fun-mooc.fr>

Table des matières

	Page
1 Introduction au MOOC et aux outils Python	1
1.1 Versions de Python	1
1.2 Installer la distribution standard Python	3
1.3 Un peu de lecture	7
1.4 “Notebooks” Jupyter comme support de cours	9
1.5 Modes d’exécution	12
1.6 La suite de Fibonacci	14
1.7 La suite de Fibonacci (version pour terminal)	16
1.8 La ligne shebang	18
1.9 Dessiner un carré	19
1.10 Noms de variables	20
1.11 Les mots-clés de Python	23
1.12 Un peu de calcul sur les types	24
1.13 Gestion de la mémoire	25
1.14 Typages statique et dynamique	26
1.15 Utiliser Python comme une calculatrice	29
1.16 Affectations et Opérations (à la +=)	35
1.17 Notions sur la précision des calculs flottants	37
1.18 Opérations bit à bit (bitwise)	39
1.19 Estimer le plus petit (grand) flottant	41
2 Notions de base, premier programme en Python	45
2.1 Caractères accentués	45
2.2 Les outils de base sur les chaînes de caractères (str)	50
2.3 Formatage de chaînes de caractères	54
2.4 Obtenir une réponse de l’utilisateur	59
2.5 Expressions régulières et le module re	60
2.6 Expressions régulières	77
2.7 Les slices en Python	81
2.8 Méthodes spécifiques aux listes	86
2.9 Objets mutables et objets immuables	90
2.10 Tris de listes	92
2.11 Indentations en Python	94
2.12 Bonnes pratiques de présentation de code	97
2.13 L’instruction pass	100
2.14 Fonctions avec ou sans valeur de retour	102
2.15 Formatage des chaînes de caractères	106
2.16 Séquences	107
2.17 Listes	109
2.18 Instruction if et fonction def	110
2.19 Comptage dans les chaînes	112
2.20 Compréhensions (1)	113
2.21 Compréhensions (2)	115
3 Renforcement des notions de base, références partagées	117

3.1	Les fichiers	117
3.2	Fichiers et utilitaires	122
3.3	Formats de fichiers : JSON et autres	126
3.4	Fichiers systèmes	128
3.5	La construction de tuples	130
3.6	Sequence unpacking	134
3.7	Plusieurs variables dans une boucle for	138
3.8	Fichiers	141
3.9	Sequence unpacking	142
3.10	Dictionnaires	143
3.11	Clés immuables	150
3.12	Gérer des enregistrements	152
3.13	Dictionnaires et listes	155
3.14	Fusionner des données	156
3.15	Ensembles	160
3.16	Ensembles	166
3.17	Exercice sur les ensembles	168
3.18	try ... else ... finally	171
3.19	L'opérateur is	173
3.20	Listes infinies & références circulaires	177
3.21	Les différentes copies	179
3.22	L'instruction del	183
3.23	Affectation simultanée	185
3.24	Les instructions += et autres revisitées	186
3.25	Classe	188
4	Fonctions et portée des variables	191
4.1	Passage d'arguments par référence	191
4.2	Rappels sur docstring	193
4.3	isinstance	195
4.4	Type hints	198
4.5	Conditions & Expressions Booléennes	203
4.6	Évaluation des tests	207
4.7	Une forme alternative du if	210
4.8	Récapitulatif sur les conditions dans un if	212
4.9	L'instruction if	217
4.10	Expression conditionnelle	218
4.11	La boucle while ... else	219
4.12	Calculer le PGCD	221
4.13	Exercice	222
4.14	Mise à la puissance	225
4.15	Le module builtins	226
4.16	Visibilité des variables de boucle	231
4.17	L'exception UnboundLocalError	234
4.18	Les fonctions globals et locals	237
4.19	Passage d'arguments	240
4.20	Un piège courant	245
4.21	Arguments keyword-only	246
4.22	Passage d'arguments	248
5	Itération, importation et espace de nommage	251
5.1	Les instructions break et continue	251
5.2	Une limite de la boucle for	252
5.3	Itérateurs	254
5.4	Programmation fonctionnelle	255
5.5	Tri de listes	257
5.6	Comparaison de fonctions	263
5.7	Construction de liste par compréhension	265

5.8	Compréhensions imbriquées	267
5.9	Compréhensions	270
5.10	Le code de Vigenère	272
5.11	Expressions génératrices	276
5.12	<code>yield from</code> pour cascader deux générateurs	279
5.13	Les boucles <code>for</code>	281
5.14	Précisions sur l'importation	282
5.15	Où sont cherchés les modules ?	285
5.16	La clause <code>import as</code>	288
5.17	Récapitulatif sur <code>import</code>	291
5.18	La notion de package	295
5.19	Usage avancés de <code>import</code>	299
5.20	Décoder le module <code>this</code>	305
6	Conception des classes	309
6.1	Introduction aux classes	309
6.2	Enregistrements et instances	314
6.3	Les <code>property</code>	317
6.4	Un exemple de classes de la bibliothèque standard	322
6.5	Manipuler des ensembles d'instances	326
6.6	Surcharge d'opérateurs (1)	330
6.7	Méthodes spéciales (2/3)	336
6.8	Méthodes spéciales (3/3)	342
6.9	Héritage	345
6.10	Hériter des types built-in ?	350
6.11	Pour en savoir plus	354
6.12	<code>dataclasses</code>	354
6.13	Énumérations	356
6.14	Héritage, typage	358
6.15	Héritage multiple	361
6.16	La Method Resolution Order (MRO)	363
6.17	Les attributs	366
6.18	Espaces de nommage	369
6.19	Context managers et exceptions	379
6.20	Exercice sur l'utilisation des classes	382
6.21	Comment organiser les sources de votre projet Python	389
6.22	Outils périphériques	399
6.23	Quelques sujets d'exercice en vrac	401
6.24	Exercice - niveau avancé	407
7	L'écosystème data science Python	413
7.1	Installations supplémentaires	413
7.2	<code>numpy</code> en dimension 1	414
7.3	Type d'un tableau <code>numpy</code>	419
7.4	Forme d'un tableau <code>numpy</code>	421
7.5	Création de tableaux	426
7.6	Le broadcasting	432
7.7	Index et slices	438
7.8	Slicing	440
7.9	Opérations logiques	445
7.10	Algèbre linéaire	455
7.11	Indexation évoluée	460
7.12	Divers	469
7.13	Utilisation de la mémoire	469
7.14	Types structurés pour les cellules	472
7.15	Assemblages et découpages	473
7.16	Exercice - niveau basique	476
7.17	Exercice - niveau basique	477

7.18	Exercice - niveau intermédiaire	478
7.19	Exercice - niveau avancé	480
7.20	construire une matrice diagonale	481
7.21	remplir une matrice : $m(i, j) = x_i * x_j$	482
7.22	La data science en général	483
7.23	Series de pandas	487
7.24	DataFrame de pandas	496
7.25	Opération avancées en pandas	518
7.26	Séries temporelles en pandas	533
7.27	matplotlib - 2D	535
7.28	matplotlib 3D	545
7.29	Notebooks interactifs	559
7.30	Animations interactives avec matplotlib	570
7.31	Autres bibliothèques de visualisation	576
7.32	Application à la transformée de Fourier	583
7.33	Le théorème de Taylor illustré	591
7.34	Coronavirus	597
8	Programmation asynchrone avec asyncio	607
8.1	Pourquoi les threads c'est délicat ?	607
8.2	Avertissement relatif à asyncio et Python-3.7	614
8.3	Essayez vous-même	617
8.4	asyncio - un exemple un peu plus réaliste	618
8.5	Gestion de sous-process	620
8.6	Nouveautés par rapport aux vidéos	626
9	Sujets avancés	629
9.1	Décorateurs	629

Chapitre 1

Introduction au MOOC et aux outils Python

1.1 w1-s1-c1-versions-python

Versions de Python

Version de référence : Python-3.6

Comme on l'indique dans la vidéo, la version de Python qui a servi de référence pour le MOOC est la version 3.6, c'est notamment avec cette version que l'on a tourné les vidéos.

Versions plus anciennes

Certaines précautions sont à prendre si vous utilisez une version plus ancienne :

Python-3.5

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

```
[1]: age = 10
      # un exemple de f-string
      f"Jean a {age} ans"
```

```
[1]: 'Jean a 10 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 avec les chaînes de caractères - et dans le cas présent il faudrait remplacer par ceci :

```
[2]: age = 10

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

```
[2]: 'Jean a 10 ans'
```

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

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.

Version utilisée dans les notebooks / versions plus récentes

Tout le cours doit pouvoir s'exécuter tel quel avec une version plus récente de Python.

Cela dit, certains compléments illustrent des nouveautés apparues après la 3.6, comme les dataclasses qui sont apparues avec python-3.7, et que nous verrons en semaine 6.

Dans tous les cas, nous signalons systématiquement les notebooks qui nécessitent une version plus récente que 3.6.

Voici enfin, à toutes fins utiles, un premier fragment de code Python qui affiche la version de Python utilisée dans tous les notebooks de ce cours.

Nous reviendrons en détail sur l'utilisation des notebooks dans une prochaine séquence, dans l'immédiat pour exécuter ce code vous pouvez :

- désigner avec la souris la cellule de code ; vous verrez alors apparaître une petite flèche à côté du mot **In**, en cliquant cette flèche vous exécutez le code ;
- une autre méthode consiste à sélectionner la cellule de code avec la souris ; une fois que c'est fait vous pouvez cliquer sur le bouton **>| Run** dans la barre de menu (bleue claire) du notebook.

```
[3]: # ce premier fragment de code affiche des détails sur la
     # version de python qui exécute tous les notebooks du cours
import sys
print(sys.version_info)
```

```
sys.version_info(major=3, minor=7, micro=3, releaselevel='final', serial=0)
```

Pas de panique si vous n'y arrivez pas, nous consacrerons très bientôt une séquence entière à l'utilisation des notebooks :)

1.2 w1-s2-c1-installer-python

Installer la distribution standard Python

1.2.1 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

1.2.2 Sachez à qui vous parlez

Mais avant qu'on n'avance sur l'installation proprement dite, il nous faut insister sur un point qui déroute parfois les débutants. On a parfois besoin de recourir à l'emploi d'un terminal, surtout justement pendant la phase d'installation.

Lorsque c'est le cas, il est important de bien distinguer :

- les cas où on s'adresse au terminal (en jargon, on dit le shell),
- et les cas où on s'adresse à l'interpréteur Python.

C'est très important car ces deux programmes ne parlent pas du tout le même langage ! Il peut arriver au début qu'on écrive une commande juste, mais au mauvais interlocuteur, et cela peut être source de frustration. Essayons de bien comprendre ce point.

Le terminal

Je peux dire que je parle à mon terminal quand l'invite de commande (en jargon on dit le prompt) se termine par un dollar \$ - ou un simple chevron > sur Windows

Par exemple sur un mac :

```
~/git/flotpython/w1 $
```

Ou sur Windows :

```
C:\Users>
```

L'interprète Python

À partir du terminal, je peux lancer un interpréteur Python, qui se reconnaît car son prompt est fait de 3 chevrons >>>

```
~/git/flotpython/w1 $ python3
Python 3.7.0 (default, Jun 29 2018, 20:14:27)
[Clang 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Pour sortir de l'interpréteur Python, et retourner au terminal, j'utilise la fonction Python `exit()` :

```
~/git/flotpython/w1 $ python3
>>> 20 * 60
1200
>>> exit()
~/git/flotpython/w1 $ python3
```

Les erreurs typiques

Gardez bien cette distinction présente à l'esprit, lorsque vous lisez la suite. Voici quelques symptômes habituels de ce qu'on obtient si on se trompe.

Par exemple, la commande `python3 -V` est une commande qui s'adresse au terminal ; c'est pourquoi nous la faisons précéder d'un dollar \$.

Si vous essayez de la taper alors que vous êtes déjà dans un interpréteur python - ou sous IDLE d'ailleurs -, vous obtenez un message d'erreur de ce genre :

```
>>> python3 -V
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'python3' is not defined
```

Réciproquement, si vous essayez de taper du Python directement dans un terminal, ça se passe mal aussi, forcément. Par exemple sur Mac, avec des fragments Python tout simples :

```
~/git/flotpython/w1 $ import math
-bash: import: command not found
~/git/flotpython/w1 $ 30 * 60
-bash: 30: command not found
~/git/flotpython/w1 $ foo = 30 * 60
-bash: foo: command not found
```

1.2.3 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 seule 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 macOS :

- `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 \(en anglais\)](#)

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

1.2.4 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 prêt, 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 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 déjà 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](#)

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

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 gestionnaire de paquets pour l'installation de bibliothèques supplémentaires qui s'appelle `conda`.

1.3 w1-s2-c2-lecture

Un peu de lecture

1.3.1 Complément - niveau basique

Mise à jour de Juillet 2018

Le 12 Juillet 2018, Guido van Rossum [a annoncé qu'il quittait la fonction de BDFL](#) qu'il occupait depuis près de trois décennies. Il n'est pas tout à fait clair à ce stade comment va évoluer la gouvernance de Python.

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, cliquez dans la cellule de code, et faites au clavier "Majuscule/Entrée" ou "Shift/Enter")

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

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
```

In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than **right** now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

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](#) les mots-clé "python module pathlib".
- J'aimerais vous signaler également une initiative pour [traduire la documentation officielle en français](#).

Historique et survol

- La FAQ officielle de Python (en anglais) sur [les choix de conception et l'historique du langage](#).
- L'article de Wikipédia (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 [discours de Guido van Rossum à PyCon 2016](#).
- Sur YouTube, le [sketch des Monty Python](#), (malheureusement plus disponible sur YouTube) d'où proviennent 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.

1.3.2 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 \(Python Enhancement Proposals\)](#) - sur wikipedia
- [Le premier PEP : PEP-001](#) donc décrit en détail le cycle de vie des PEPs

- Le [PEP 008](#), qui préconise un style de présentation (style guide)
- L'[index de tous les PEPs](#)

1.4 w1-s4-c1-utiliser-les-notebooks

“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.

Avertissement : réglages du navigateur

Avant toute chose, pour un bon fonctionnement des notebooks, on rappelle qu'il est nécessaire d'avoir autorisé dans votre navigateur les cookies en provenance du site Internet **nbhosting.inria.fr**, qui héberge l'infrastructure qui héberge tous les notebooks.

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 de menu (sur fond bleu clair), contenant :

- un titre pour le notebook, avec un numéro de version ;
- une barre de menus avec les entrées **File**, **Insert**, **Cell**, **Kernel** ;
- 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 menu - en haut du notebook, donc - sur celui en forme de flèche triangulaire vers la droite (Play) :

```
[1]: 20 * 30
```

```
[1]: 600
```

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 sélectionner la première cellule,
- et à taper Shift+Enter jusqu’à atteindre la fin 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.

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

```
[2]: 1.4142135623730951
```

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 :

```
[3]: message = "Faites attention à l'ordre dans lequel vous évaluez les notebooks"
```

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

Faites attention à l’ordre dans lequel vous évaluez les notebooks

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

```
[5]: # 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 évalué, 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-même 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.

1.5 w1-s4-c2-interpreteur-et-notebooks

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	<code>python3 <fichier>.py</code>
ligne à ligne	<code>python3</code> en mode interactif ou sous <code>ipython3</code> 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 conséquent, 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 tout 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 :

```
[1]: 10 * 10
      20 * 20
      30 * 30
```

```
[1]: 900
```

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.

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

      print(a, b)
```

10 20

On peut naturellement mélanger des objets de plusieurs types, et donc mélanger des chaînes de caractères 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 préférera quelque chose comme :

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

a = 10 et b = 20

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

a = 10 et b = 20

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 :

```
[5]: 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, notamment lorsque l'expression est complexe et pour rendre explicite la valeur qui vient d'être affectée :

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

100

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.

1.6 w1-s4-c3-fibonacci-prompt

La suite de Fibonacci

1.6.1 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 :

- $u_0 = 0$
- $u_1 = 1$
- $\forall n \geq 2, u_n = u_{n-1} + u_{n-2}$

Ce qui donne pour les premières valeurs :

n	fibonacci(n)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	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.

```
[1]: def fibonacci(n):
    "retourne le nombre de fibonacci pour l'entier n"
    # pour les deux petites valeurs de n, on peut retourner n
    if n <= 1:
        return n
    # sinon on initialise f2 pour n-2 et f1 pour n-1
    f2, f1 = 0, 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 :

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

# NOTE:
# auto-exec-for-latex has used instead:
#####
entier = 12
#####
```

On imprime le résultat :

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

```
fibonacci(12) = 144
```

Exercice

Vous pouvez donc à présent :

- exécuter le code dans ce notebook
- télécharger ce code sur votre disque comme un fichier `fibonacci_prompt.py`
 - utiliser pour cela le menu “File -> Download as -> Python”
 - et renommer 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.

1.7 w1-s4-c4-fibonacci

La suite de Fibonacci (version pour terminal)

1.7.1 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.

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

Puis nous répétons la fonction `fibonacci` :

```
[2]: def fibonacci(n):
    "retourne le nombre de fibonacci pour l'entier n"
    # pour les deux premières valeurs de n, on peut renvoyer n
    if n <= 1:
        return n
    # sinon on initialise f2 pour n-2 et f1 pour n-1
    f2, f1 = 0, 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.

```
[3]: # 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

# NOTE:
# auto-exec-for-latex has used instead:
#####
entier = 8
#####
```

Nous pouvons à présent afficher le résultat :

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

`fibonacci(8) = 21`

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'interpréteur Python comme ceci :

```
$ python3 fibonacci.py 56
```

(sans taper le signe \$ qui indique simplement le prompt du terminal.)

1.8 w1-s4-c5-shebang

La ligne shebang

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

1.8.1 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 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 avons vu ce code en détail dans les deux compléments précédents). 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 :

```
$ chmod +x /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` ; de cette façon vous les rendez ainsi accessible à partir 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.

1.9 w1-s4-x1-turtle

Dessiner un carré

1.9.1 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 sauvegarder 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`.

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

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

```
[2]: 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 :

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

On peut alors dessiner notre carré :

```
[ ]: square(200)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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

```
[ ]: turtle.exitonclick()

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

1.9.2 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.

1.10 w1-s5-c1-noms-de-variables

Noms de variables

1.10.1 Complément - niveau basique

Revenons sur les noms de variables autorisés ou non.

Les noms les plus simples sont constitués de lettres. Par exemple :

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

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

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

```
[2]: False
```

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 préférera ainsi :

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

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

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

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

```
[5]: age_moyen_dept75 = 80
```

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

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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.

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

Et en tout 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 regardez par exemple cette variable que nous n'avons définie nulle part mais qui pourtant existe bel et bien :

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

```
[7]: '__main__'
```

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 :

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Caractères exotiques

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

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

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

```
[9]: 0.7071067811865476
```

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.

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 , voyez par exemple la confusion que peut créer le fait de nommer un identificateur ou Π ou π ;
- 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 :

- [le PEP 3131](#) qui définit les caractères exotiques autorisés, et qui repose à son tour sur
- <http://www.unicode.org/reports/tr31/> (très technique!)

1.11 w1-s5-c2-mots-cles

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.

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

au dessus

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

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

     # NOTE
     # auto-exec-for-latex has skipped execution of this cell
```

Liste complète

Voici la liste complète des mots-clés :

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

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 fonctionnalité, dite 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 : variables, mots-clés, etc. D'une manière générale, nous vous déconseillons fortement d'utiliser un éditeur de texte qui n'offre pas cette fonctionnalité de coloration syntaxique.

Pour en savoir plus

On peut se reporter à cette page :

https://docs.python.org/3/reference/lexical_analysis.html#keywords

1.12 w1-s5-c3-introduction-types

Un peu de calcul sur les types

1.12.1 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 :

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

```
[1]: int
```

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

```
[2]: str
```

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 objets

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

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

```
[3]: int
```

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

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

```
[4]: list
```

1.12.2 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 :

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

```
[5]: True
```

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.

1.13 w1-s5-c4-garbage-collector

Gestion de la mémoire

1.13.1 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 !)

1.13.2 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 matière 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 informatique 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.

1.14 w1-s5-c5-type-checking

Typages statique et dynamique

1.14.1 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 caractères' */
}

```

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`.

```
[1]: 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 :

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

```
[2]: 326
```

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

```
[3]: ['a', 'b', 'c', 0, 20, 30, 'spam', 'eggs']
```

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 :

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

```
[4]: 'abcdef'
```

Mais par contre on ne pourrait pas faire

```
[5]: # ceci va déclencher une exception à l'exécution  
somme(12, [1, 2, 3])  
  
# NOTE:  
# auto-exec-for-latex has used instead:  
#####  
try:  
    somme(12, [1, 2, 3])  
except:  
    print('OOPS')  
#####
```

OOPS

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 a aucun des surcoûts qu'on trouve avec C ou C++ en matière de définition de type ;
- aucun contrôle de type n'est effectué a priori par le langage au moment de la définition de la fonction `somme` ;
- par contre au moment de l'exécution, 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 à l'usage, qu'en matière 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 fonctionnement 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.

1.15 w1-s6-c1-calculatrice

Utiliser Python comme une calculatrice

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

[1]: `20 * 60`

[1]: 1200

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

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

[2]: 110

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 :

[3]: `(2 * 30) + (10 * 5)`

[3]: 110

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

[4]: `48 / 5`

[4]: 9.6

Rappelez-vous des opérateurs suivants qui sont très pratiques :

code	opération
<code>//</code>	quotient
<code>%</code>	modulo
<code>**</code>	puissance

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

[5]: 9

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

[6]: 3

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

[7]: 1024

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 :

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

```
[8]: (-7.5+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 :

```
[9]: 1j * 1.j
```

```
[9]: (-1+0j)
```

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 :

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

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

```
[11]: 100
```

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

```
[12]: 50
```

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

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

```
[13]: 62.8318
```

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 :

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

Et ainsi imprimer les racines troisièmes de l'unité par la formule :

$$r_n = e^{2i\pi \frac{n}{3}}, \text{ pour } n \in \{0, 1, 2\}$$

```
[15]: 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))
```

```
n= 0 racine = (1+0j)
n= 1 racine = (-0.4999999999999998+0.8660254037844388j)
n= 2 racine = (-0.5000000000000004-0.8660254037844384j)
```

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 :

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

```
[16]: int
```

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

```
[17]: float
```

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

```
[18]: complex
```

Chaînes de caractères

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

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

```
Bonjour le monde !
```

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 :

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

      # NOTE:
```

```
# auto-exec-for-latex has used instead:
#####
reponse = '25'
#####
```

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

25

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

[22]: str

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

```
[23]: # 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
```

[23]: '2525'

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)

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

[24]: int

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

[25]: 50

Ou si on préfère, en une seule fois :

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

le double de votre age est 50

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	<code>int</code>
Flottant	<code>float</code>
Complexe	<code>complex</code>
Chaîne	<code>str</code>

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

On a illustré cette même technique dans les exemples suivants :

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

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

```
[28]: '2345'
```

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

```
[29]: (2345+0j)
```

Nous verrons plus tard que ceci se généralise à tous les types de Python, pour convertir un objet `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.

```
[30]: tres_grand_nombre = 23_456_789_012_345

tres_grand_nombre
```

```
[30]: 23456789012345
```

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

```
[31]: 123456.789012
```

Entiers et bases

Les calculatrices 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 :


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

200

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

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

200

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

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

200

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

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

200

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.

1.16 w1-s6-c2-affectation-operateurs

Affectations et Opérations (à la +=)

1.16.1 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.

Incrémenter

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

```
[1]: entier = 10

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

```
entier 12
```

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

```
[2]: entier = 10

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

```
entier 12
```

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 :

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

```
après décrémentation 8
après doublement 16
mis à moitié 8.0
```

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 :

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

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

```
liste [0, 3, 5]
après ajout [0, 3, 5, 'a', 'b']
```

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 :

```
[5]: entier = 2
print('entier:', entier)
entier **= 10
print('à la puissance dix:', entier)
```

```
entier %= 5
print('modulo 5:', entier)
```

```
entier: 2
à la puissance dix: 1024
modulo 5: 4
```

Et pour ceux qui connaissent déjà un peu Python, on peut même le faire avec des opérateurs de décalage, que nous verrons très bientôt :

```
[6]: entier <<= 2
print('double décalage gauche:', entier)
```

```
double décalage gauche: 16
```

1.17 w1-s6-c3-precision-flottants

Notions sur la précision des calculs flottants

1.17.1 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 :

```
[1]: 0.2 + 0.4
```

```
[1]: 0.6000000000000001
```

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.

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

```
[2]: False
```

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` :

```
[3]: # 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)
```

```
[3]: True
```

Ou encore d'ailleurs, équivalent et plus lisible :

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

```
[4]: True
```

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 :

```
[5]: from decimal import Decimal

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

```
[5]: True
```

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](#) ;
- une [présentation plus fouillée sur l'encodage des flottants \(en anglais\)](#) ; ce dernier document, très bien fait, ne dépend pas du langage Python mais illustre le standard IEE-754 sur des exemples concrets.

1.18 w1-s6-c4-entiers-bit-a-bit

Opérations bit à bit (bitwise)

1.18.1 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 \wedge

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

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

Ce qui nous donne comme décomposition binaire :

$$\begin{aligned} x49 &= 49 = 32 + 16 + 1 \rightarrow (0, 1, 1, 0, 0, 0, 1) \\ y81 &= 81 = 64 + 16 + 1 \rightarrow (1, 0, 1, 0, 0, 0, 1) \end{aligned}$$

Pour comprendre comment passer de $32 + 16 + 1$ à $(0, 1, 1, 0, 0, 0, 1)$ il suffit d'observer que :

$$32 + 16 + 1 = 0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$$

ET logique : opérateur $\&$ L'opération logique $\&$ va faire un et logique bit à bit entre les opérandes, ainsi

```
[2]: x49 & y81
```

```
[2]: 17
```

Et en effet :

$$\begin{aligned} x49 &\rightarrow (0, 1, 1, 0, 0, 0, 1) \\ y81 &\rightarrow (1, 0, 1, 0, 0, 0, 1) \\ x49 \& y81 &\rightarrow (0, 0, 1, 0, 0, 0, 1) \rightarrow 16 + 1 \rightarrow 17 \end{aligned}$$

OU logique : opérateur $|$ De même, l'opérateur logique $|$ fait simplement un ou logique, comme ceci :

```
[3]: x49 | y81
```

```
[3]: 113
```

On s'y retrouve parce que :

$$\begin{aligned} x49 &\rightarrow (0, 1, 1, 0, 0, 0, 1) \\ y81 &\rightarrow (1, 0, 1, 0, 0, 0, 1) \\ x49 | y81 &\rightarrow (1, 1, 1, 0, 0, 0, 1) \rightarrow 64 + 32 + 16 + 1 \rightarrow 113 \end{aligned}$$

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

```
[4]: x49 ^ y81
```

```
[4]: 96
```

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 $2^4 = 16$:

```
[5]: x49 << 4
```

```
[5]: 784
```

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

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

```
[6]: x49 >> 4
```

```
[6]: 3
```

$$\begin{aligned} x49 &\rightarrow (0, 1, 1, 0, 0, 0, 1) \\ x49 >> 4 &\rightarrow (0, 0, 0, 0, 0, 1, 1) \rightarrow 2 + 1 \rightarrow 3 \end{aligned}$$

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` :

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

```
[7]: '0b110001'
```

Dans l'autre sens, on peut aussi entrer un entier directement en base 2 comme ceci :

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

```
[8]: True
```

Ici, comme on le voit, `x49bis` est bien un entier.

Pour en savoir plus

[Section de la documentation Python.](#)

1.19 w1-s6-x1-flottants

Estimer le plus petit (grand) flottant

1.19.1 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 :

```
[1]: 10**-320
```

```
[1]: 1e-320
```

```
[2]: 10**-330
```

```
[2]: 0.0
```

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](#).

```
[3]: 10**-325
```

```
[3]: 0.0
```

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"

1.19.2 Complément - niveau avancé

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

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

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, m
    in=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant
    _dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

Et notamment, [comme expliqué ici](#).

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

```
Flottant minimum 2.2250738585072014e-308
Flottant maximum 1.7976931348623157e+308
```

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](#).

Chapitre 2

Notions de base, premier programme en Python

2.1 w2-s1-cl-accents

Caractères accentués

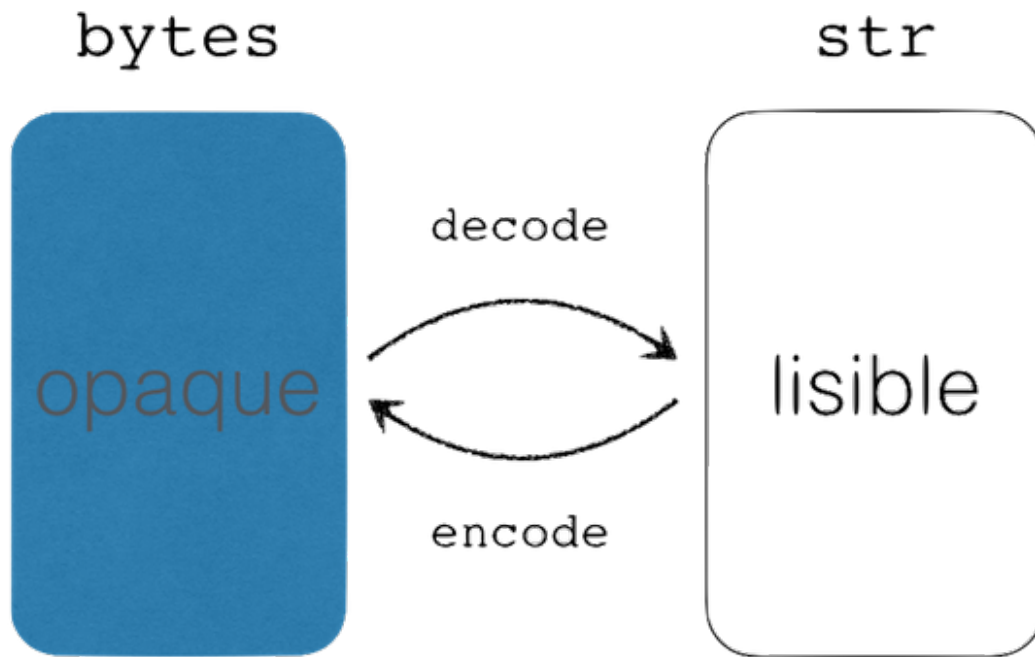
Ce complément expose quelques bases concernant les caractères accentués, et notamment les précautions à prendre pour pouvoir en insérer dans un programme Python. Nous allons voir que cette question, assez scabreuse, dépasse très largement le cadre de Python.

2.1.1 Complément - niveau basique

Un caractère n'est pas un octet

Avec Unicode, on a cassé le modèle un caractère == un octet. Aussi en Python 3, lorsqu'il s'agit de manipuler des données provenant de diverses sources de données :

- le type `byte` est approprié si vous voulez charger en mémoire les données binaires brutes, sous forme d'octets donc ;
- le type `str` est approprié pour représenter une chaîne de caractères - qui, à nouveau ne sont pas forcément des octets ;
- on passe de l'un à l'autre de ces types par des opérations d'encodage et décodage, comme illustré ci-dessous ;
- et pour toutes les opérations d'encodage et décodage, il est nécessaire de connaître l'encodage utilisé.



On peut appeler les méthodes `encode` et `decode` sans préciser l'encodage (dans ce cas Python choisit l'encodage par défaut sur votre système). Cela dit, il est de loin préférable d'être explicite et de choisir son encodage. En cas de doute, il est recommandé de spécifier explicitement `utf-8`, qui se généralise au détriment d'encodages anciens comme `cp1252` (Windows) et `iso8859-*`, que de laisser le système hôte choisir pour vous.

Utilisation des accents et autres cédilles

Python 3 supporte Unicode par défaut. Vous pouvez donc, maintenant, utiliser sans aucun risque des accents ou des cédilles dans vos chaînes de caractères. Il faut cependant faire attention à deux choses :

- Python supporte Unicode, donc tous les caractères du monde, mais les ordinateurs n'ont pas forcément les polices de caractères nécessaires pour afficher ces caractères ;
- Python permet d'utiliser des caractères Unicode pour les noms de variables, mais nous vous recommandons dans toute la mesure du possible d'écrire votre code en anglais, comme c'est le cas pour la quasi-totalité du code que vous serez amenés à utiliser sous forme de bibliothèques. Ceci est particulièrement important pour les noms de lignes et de colonnes dans un dataset afin de faciliter les transferts entre logiciels, la majorité des logiciels n'acceptant pas les accents et cédilles dans les noms de variables.

Ainsi, il faut bien distinguer les chaînes de caractères qui doivent par nature être adaptées au langage des utilisateurs du programme, et le code source qui lui est destiné aux programmeurs et qui doit donc éviter d'utiliser autre chose que de l'anglais.

2.1.2 Complément - niveau intermédiaire

Où peut-on mettre des accents ?

Cela étant dit, si vous devez vraiment mettre des accents dans vos sources, voici ce qu'il faut savoir.

Noms de variables

- S'il n'était pas possible en Python 2 d'utiliser un caractère accentué dans un nom de variable (ou d'un identificateur au sens large), cela est à présent permis en Python 3 :

```
[1]: # pas recommandé, mais autorisé par le langage
nb_élèves = 12
```

- On peut même utiliser des symboles, comme par exemple

```
[2]: from math import cos, pi as Π
θ = Π / 4
cos(θ)
```

```
[2]: 0.7071067811865476
```

- Je vous recommande toutefois de ne pas utiliser cette possibilité, si vous n'êtes pas extrêmement familier avec les caractères Unicode.
- Enfin, pour être exhaustif, sachez que seule une partie des caractères Unicode sont autorisés dans ce cadre, c'est heureux parce que les caractères comme, par exemple, [l'espace non-sécable](#) pourraient, s'ils étaient autorisés, être la cause de milliers d'heures de debugging à frustration garantie :)

Pour les curieux, vous pouvez en savoir plus [à cet endroit de la documentation officielle \(en anglais\)](#).

Chaînes de caractères

- Vous pouvez naturellement mettre des accents dans les chaînes de caractères. Cela dit, les données manipulées par un programme proviennent pour l'essentiel de sources externes, comme une base de données ou un formulaire Web, et donc le plus souvent pas directement du code source. Les chaînes de caractères présentes dans du vrai code sont bien souvent limitées à des messages de logging, et le plus souvent d'ailleurs en anglais, donc sans accent.
- Lorsque votre programme doit interagir avec les utilisateurs et qu'il doit donc parler leur langue, c'est une bonne pratique de créer un fichier spécifique, que l'on appelle fichier de ressources, qui contient toutes les chaînes de caractères spécifiques à une langue. Ainsi, la traduction de votre programme consistera à simplement traduire ce fichier de ressources.

```
message = "on peut mettre un caractère accentué dans une chaîne"
```

Commentaires

- Enfin on peut aussi bien sûr mettre dans les commentaires n'importe quel caractère Unicode, et donc notamment des caractères accentués si on choisit malgré tout d'écrire le code en français.

```
# on peut mettre un caractère accentué dans un commentaire
# ainsi que cos(Θ), ∀x ∈ f(t)dt vous voyez l'idée générale
```

Qu'est-ce qu'un encodage ?

Comme vous le savez, la mémoire - ou le disque - d'un ordinateur ne permet que de stocker des représentations binaires. Il n'y a donc pas de façon "naturelle" de représenter un caractère comme 'A', un guillemet ou un point-virgule.

On utilise pour cela un encodage, par exemple [le code US-ASCII](#) stipule, pour faire simple, qu'un 'A' est représenté par l'octet 65 qui s'écrit en binaire 01000001. Il se trouve qu'il existe plusieurs encodages, bien sûr incompatibles, selon les systèmes et les langues. Vous trouverez plus de détails ci-dessous.

Le point important est que pour pouvoir ouvrir un fichier "proprement", il faut bien entendu disposer du contenu du fichier, mais il faut aussi connaître l'encodage qui a été utilisé pour l'écrire.

Précautions à prendre pour l'encodage de votre code source

L'encodage ne concerne pas simplement les objets chaîne de caractères, mais également votre code source. Python 3 considère que votre code source utilise par défaut l'encodage **UTF-8**. Nous vous conseillons de conserver cet encodage qui est celui qui vous offrira le plus de flexibilité.

Vous pouvez malgré tout changer l'encodage de votre code source en faisant figurer dans vos fichiers, en première ou deuxième ligne, une déclaration comme ceci :

```
# -*- coding: <nom_de_l_encodage> -*-
```

ou plus simplement, comme ceci :

```
# coding: <nom_de_l_encodage>
```

Notons que la première option est également interprétée par l'éditeur de texte Emacs pour utiliser le même encodage. En dehors de l'utilisation d'Emacs, la deuxième option, plus simple et donc plus pythonique, est à préférer.

Le nom **UTF-8** fait référence à Unicode (ou pour être précis, à l'encodage le plus répandu parmi ceux qui sont définis dans la norme Unicode, comme nous le verrons plus bas). Sur certains systèmes plus anciens vous pourrez être amenés à utiliser un autre encodage. Pour déterminer la valeur à utiliser dans votre cas précis vous pouvez faire dans l'interpréteur interactif :

```
# ceci doit être exécuté sur votre machine
import sys
print(sys.getdefaultencoding())
```

Par exemple avec d'anciennes versions de Windows (en principe de plus en plus rares) vous pouvez être amenés à écrire :

```
# coding: cp1252
```

La syntaxe de la ligne `coding` est précisée dans [cette documentation](#) et dans le [PEP 263](#).

Le grand malentendu

Si je vous envoie un fichier contenant du français encodé avec, disons, [ISO/IEC 8859-15 - a.k.a. Latin-9](#) ; vous pouvez voir dans la table qu'un caractère '€' va être matérialisé dans mon fichier par un octet '0xA4', soit 164.

Imaginez maintenant que vous essayez d'ouvrir ce même fichier depuis un vieil ordinateur Windows configuré pour le français. Si on ne lui donne aucune indication sur l'encodage, le programme qui va lire ce fichier sur Windows va utiliser l'encodage par défaut du système, c'est-à-dire [CP1252](#). Comme vous le voyez dans cette table, l'octet '0xA4' correspond au caractère ¤ et c'est ça que vous allez voir à la place de €.

Contrairement à ce qu'on pourrait espérer, ce type de problème ne peut pas se régler en ajoutant une balise `# coding: <nom_de_l_encodage>`, qui n'agit que sur l'encodage utilisé pour lire le fichier source en question (celui qui contient la balise).

Pour régler correctement ce type de problème, il vous faut préciser explicitement l'encodage à utiliser pour décoder le fichier. Et donc avoir un moyen fiable de déterminer cet encodage ; ce qui n'est pas toujours aisé d'ailleurs, mais c'est une autre discussion malheureusement. Ce qui signifie que pour être totalement propre, il faut pouvoir préciser explicitement le paramètre `encoding` à l'appel de toutes les méthodes qui sont susceptibles d'en avoir besoin.

Pourquoi ça marche en local ?

Lorsque le producteur (le programme qui écrit le fichier) et le consommateur (le programme qui le lit) tournent dans le même ordinateur, tout fonctionne bien - en général - parce que les deux programmes se ramènent à l'encodage défini comme l'encodage par défaut.

Il y a toutefois une limite, si vous utilisez un Linux configuré de manière minimale, il se peut qu'il utilise par défaut l'encodage `US-ASCII` - voir plus bas - qui étant très ancien ne "connaît" pas un simple é, ni a fortiori €. Pour écrire du français, il faut donc au minimum que l'encodage par défaut de votre ordinateur contienne les caractères français, comme par exemple :

- ISO 8859-1 (`Latin-1`)
- ISO 8859-15 (`Latin-9`)
- UTF-8
- CP1252

À nouveau dans cette liste, il faut clairement préférer UTF-8 lorsque c'est possible.

Un peu d'histoire sur les encodages

Le code `US-ASCII`

Jusque dans les années 1980, les ordinateurs ne parlaient pour l'essentiel que l'anglais. La première vague de standardisation avait créé l'encodage dit `ASCII`, ou encore `US-ASCII` [voir par exemple ici](#), ou encore [en version longue ici](#).

Le code `US-ASCII` s'étend sur 128 valeurs, soit 7 bits, mais est le plus souvent implémenté sur un octet pour préserver l'alignement, le dernier bit pouvant être utilisé par exemple pour ajouter un code correcteur d'erreur - ce qui à l'époque des modems n'était pas superflu. Bref, la pratique courante était alors de manipuler une chaîne de caractères comme un tableau d'octets.

Les encodages `ISO8859-*` (`Latin*`)

Dans les années 1990, pour satisfaire les besoins des pays européens, ont été définis plusieurs encodages alternatifs, connus sous le nom de `ISO/IEC 8859-*`, nommés aussi `Latin-*`. Idéalement, on aurait pu et certainement dû définir un seul encodage pour représenter tous les nouveaux caractères, mais entre toutes les langues européennes, le nombre de caractères à ajouter était substantiel, et cet encodage unifié aurait largement dépassé 256 caractères différents, il n'aurait donc pas été possible de tout faire tenir sur un octet.

On a préféré préserver la “bonne propriété” du modèle un caractère == un octet, ceci afin de préserver le code existant qui aurait sinon dû être retouché ou réécrit.

Dès lors il n’y avait pas d’autre choix que de définir plusieurs encodages distincts, par exemple, pour le français on a utilisé à l’époque [ISO/IEC 8859-1 \(Latin-1\)](#), pour le russe [ISO/IEC 5589-5 \(Latin/Cyrillic\)](#).

À ce stade, le ver était dans le fruit. Depuis cette époque pour ouvrir un fichier il faut connaître son encodage.

Unicode

Lorsque l’on a ensuite cherché à manipuler aussi les langues asiatiques, il a de toute façon fallu définir de nouveaux encodages beaucoup plus larges. C’est ce qui a été fait par le standard [Unicode](#) qui définit 3 nouveaux encodages :

- [UTF-8](#) : un encodage à taille variable, à base d’octets, qui maximise la compatibilité avec US-ASCII ;
- [UTF-16](#) : un encodage à taille variable, à base de mots de 16 bits ;
- [UTF-32](#) : un encodage à taille fixe, à base de mots de 32 bits ;

Ces 3 standards couvrent le même jeu de caractères (113 021 tout de même dans la dernière version). Parmi ceux-ci le plus utilisé est certainement UTF-8. Un texte ne contenant que des caractères du code US-ASCII initial peut être lu avec l’encodage UTF-8.

Pour être enfin tout à fait exhaustif, si on sait qu’un fichier est au format Unicode, on peut déterminer quel est l’encodage qu’il utilise, en se basant sur les 4 premiers octets du document. Ainsi dans ce cas particulier (lorsqu’on est sûr qu’un document utilise un des trois encodages Unicode) il n’est plus nécessaire de connaître son encodage de manière “externe”.

2.2 w2-s2-c1-outils-chaines

Les outils de base sur les chaînes de caractères (**str**)

2.2.1 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

```
[ ]: help(str)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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 :

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

```
[1]: ['abc', 'def', 'ghi', 'jkl']
```

Et à l'inverse :

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

```
[2]: '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.

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

```
[3]: ['abc', 'def', 'ghi', 'jkl', '']
```

Qui s'inverse correctement cependant :

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

```
[4]: '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 :

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

```
[5]: 'zoodefzoodefzoodef'
```

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

```
[6]: 'zoodefzoodefabcdef'
```

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

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

```
[7]: 'les chevaliers qui disent 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 :

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

```
[8]: 'abc:def:ghi'
```

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 :

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

```
[9]: 'une chaîne avec des trucs qui dépassent'
```

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

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

```
[10]: ['abc', 'def', 'ghi', 'jkl']
```

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 :

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

```
[11]: 3
```

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

```
[12]: -1
```

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

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

```
[13]: 13
```

```
[14]: # notez que le résultat correspond  
# tout de même toujours au début de la chaîne  
  
# NB: en python les indices commencent à zéro  
# donc la notation ma_chaine[n]  
# permet d'accéder au n+1 ème caractère de la chaîne  
"abcdefcdefghefghijk"[13]
```

```
[14]: 'f'
```

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` :

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

```
[15]: 3
```

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

```
OOPS <class 'ValueError'> substring not found
```

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 :

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

```
[17]: True
```

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

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

```
[18]: 3
```

Signalons enfin les méthodes de commodité suivantes :

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

```
[19]: True
```

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

```
[20]: True
```

S'agissant des deux dernières, remarquons que :

```
chaine.startswith(sous_chaine)  $\iff$  chaine.find(sous_chaine) == 0
```

```
chaine.endswith(sous_chaine)  $\iff$  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 :

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

```
[21]: 'MONTY PYTHON'
```

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

```
[22]: 'monty python'
```

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

```
[23]: 'MONTY python'
```

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

```
[24]: 'Monty python'
```

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

```
[25]: 'Monty Python'
```

Pour en savoir plus

Tous ces outils sont [documentés en détail ici \(en anglais\)](#).

2.3 w2-s2-c2-formatage

Formatage de chaînes de caractères

2.3.1 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.

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

```
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 :

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

une seule ligne

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

une autre 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 :

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

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

le module math est <module 'math' from '/Users/tparment/miniconda3/envs/flopython-course/lib/python3.7/lib-dynload/math.cpython-37m-darwin.so'>

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).

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

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

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

<__main__.Personne object at 0x107c4ef28>

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 tout cas pas immédiatement ;
- d'autre part, les espaces ajoutées peuvent être plus néfastes qu'utiles ;
- enfin, on peut avoir besoin de préciser un nombre de chiffres significatifs, ou de choisir comment présenter une date.

C'est pourquoi il est plus courant de formater les chaînes - c'est-à-dire de calculer des chaînes 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 :

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

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

```
[8]: 'Jean Dupont a 35 ans'
```

Vous remarquez d'abord que la chaîne commence par `f`", c'est bien sûr pour cela qu'on l'appelle 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 `{}`.

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

```
[9]: 'dans 10 ans Jean aura 45 ans'
```

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

```
[10]: "nous avons pour l'instant 3 notes"
```

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

Comme vous le voyez, les f-strings fournissent une méthode très simple et expressive pour formater des données dans des chaînes 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 formatage était d'utiliser la méthode `format` qui est définie sur les objets `str` et qui s'utilise comme ceci :

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

```
[11]: 'Jean Dupont a 35 ans'
```

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 :

```
[12]: "{1} {0} a {2} ans".format(prenom, nom, age)
```

```
[12]: 'Dupont Jean a 35 ans'
```

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

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

```
[13]: 'Jean Dupont a 35 ans'
```

Petite digression : remarquez l'usage des parenthèses, qui me permettent de couper ma ligne en deux, car sinon ce code serait trop long pour la PEP8 ; on s'efforce toujours de ne pas dépasser 80 caractères de large, dans notre cas c'est utile notamment pour l'édition du cours au format PDF.

Reprenons : 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 :

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

```
[14]: 'Jean Dupont a 35 ans'
```

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

2.3.2 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 formater 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 avec l'opérateur `%` :

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

```
[15]: 'Jean Dupont a 35 ans'
```

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

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

```
[16]: 'Dupont, Jean, 35 ans'
```

2.3.3 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; cela se fait en précisant un format à l'intérieur des `{}` comme ceci :

`f"bla {2*math.pi:.2f} bla"`
expression
format

- à gauche du `:` vous pouvez mettre n'importe quelle expression (opérations arithmétiques, appels de fonctions, ...); bien sûr s'il n'y a pas de `:` tout ce qui est entre les `{}` constitue l'expression à évaluer;
- à droite du `:` vous pouvez préciser un format, onus allons en voir quelques exemples.

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 comment on obtient une valeur de `pi` arrondie :

```
[17]: from math import pi
```

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

```
[18]: '2pi avec seulement 2 chiffres apres la virgule 6.28'
```

Vous remarquez que la façon de construire un format est la même pour les f-strings et pour `format`.

0 en début de nombre

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

```
[19]: x = 15
      f"{x:04d}"
```

```
[19]: '0015'
```


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 largeur fixe, on utilise pour cela les formats `<^>` et `>` pour afficher à gauche, au centre, ou à droite d'une zone de largeur fixe :

```
[20]: # 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} €")
```

```
Apollin    --      Dupont    --      127 €
Myrtille   --   Lamartine   --    25432 €
Prune      --        Soc    --   827465 €
```

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.

2.4 w2-s2-c3-la-fonction-input

Obtenir une réponse de l'utilisateur

2.4.1 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 :

```
[1]: nom_ville = input("Entrez le nom de la ville : ")

# NOTE:
# auto-exec-for-latex has used instead:
#####
nom_ville = 'Paris'
#####

[2]: print(f"nom_ville={nom_ville}")
```

```
nom_ville=Paris
```

Attention à bien vérifier/convertir

Notez bien que `input` renvoie toujours une chaîne de caractères (`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 :

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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.

2.5 w2-s2-c4-expressions-regulieres

Expressions régulières et le module `re`

2.5.1 Complément - niveau basique

Avertissement

Après avoir joué ce cours plusieurs années de suite, l'expérience nous montre qu'il est difficile de trouver le bon moment pour appréhender les expressions régulières.

D'un côté il s'agit de manipulations de chaînes de caractères, mais d'un autre cela nécessite de créer des instances de classes, et donc d'avoir vu la programmation orientée objet. Du coup, les premières années nous les avons étudiées tout à la fin du cours, ce qui avait pu créer une certaine frustration.

C'est pourquoi nous avons décidé à présent de les étudier très tôt, dans cette séquence consacrée aux chaînes de caractères. Les étudiants qui seraient déçus par ce contenu sont invités à y retourner après la semaine 6, consacrée à la programmation objet.

Il nous semble important de savoir que ces fonctionnalités existent dans le langage, le détail de leur utilisation n'est toutefois pas critique, et on peut parfaitement faire l'impasse sur ce complément en première lecture.

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 mac), 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 librairie. En python, les expressions régulières sont disponibles de manière plus traditionnelle, via le module `re` (regular expressions) de la librairie standard. Le propos de ce complément est de vous en donner une première introduction.

```
[1]: import re
```

Survol

Pour ceux qui ne souhaitent pas approfondir, voici un premier exemple ; on cherche à savoir si un objet chaîne est ou non de la forme `*-*.txt`, et si oui, à calculer la partie de la chaîne qui remplace le `*` :

```
[2]: # un objet 'expression régulière' - on dit aussi "pattern"
      regexp = "(.*)-(.*)\.txt"
```

```
[3]: # la chaîne de départ
      chaine = "abcdef.txt"
```

```
[4]: # la fonction qui calcule si la chaîne "matche" le pattern
      match = re.match(regexp, chaine)
      match is None
```

```
[4]: True
```

Le fait que l'objet `match` vaut `None` indique que la chaîne n'est pas de la bonne forme (il manque un - dans le nom) ; avec une autre chaîne par contre :

```
[5]: # la chaîne de départ
      chaine = "abc-def.txt"
```

```
[6]: match = re.match(regexp, chaine)
      match is None
```

```
[6]: False
```

Ici `match` est un objet, qui nous permet ensuite d'"extraire" les différentes parties, comme ceci :

```
[7]: match[1]
```

```
[7]: 'abc'
```

```
[8]: match[2]
```

```
[8]: 'def'
```

Bien sûr on peut faire des choses beaucoup plus élaborées avec `re`, mais en première lecture cette introduction doit vous suffire pour avoir une idée de ce qu'on peut faire avec les expressions régulières.

Synonymes

Avant d'aller plus loin signalons qu'on utilise indifféremment les termes pour désigner essentiellement la même chose :

- expression régulière
- en anglais regular expression - d'où le nom du module dans `import re`
- en anglais raccourci regexp, c'est de facto devenu un nom commun
- en français on trouve aussi parfois le terme d'expression rationnelle, c'est plus rare et un peu pédant
- en anglais on utilise aussi facilement le terme de pattern
- qui du coup a été traduit en français par motif ; bon ça c'est d'un emploi assez rare.

Après selon les contextes ces termes peuvent être utilisés pour désigner des choses subtilement différentes - par exemple pour distinguer la chaîne qui spécifie un pattern de l'objet regexp qui en est déduit ; mais à ce stade de la présentation on peut signaler tous ces termes et les assimiler en gros à la même notion.

2.5.2 Complément - niveau intermédiaire

Approfondissons à présent :

Dans un terminal, `*.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 comme on vient de le voir, pour filtrer la même famille de chaînes que `*-*.txt` avec le module `re`, il nous a fallu écrire l'expression régulière sous une forme légèrement différente.

Je vous conseille d'avoir sous la main la [documentation du module re](#) pendant que vous lisez ce complément.

Avertissement

Dans ce complément nous serons amenés à utiliser des traits qui dépendent du `LOCALE`, c'est-à-dire, pour faire simple, de la configuration de l'ordinateur vis-à-vis de la langue.

Tant que vous exécutez ceci dans le notebook sur la plateforme, en principe tout le monde verra exactement la même chose. Par contre, si vous faites tourner le même code sur votre ordinateur, il se peut que vous obteniez des résultats légèrement différents.

Un exemple simple

```
findall
```

On se donne deux exemples de chaînes

```
[9]: 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`

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

```
---- dans >Lacus a donec, vitae gravida proin sociis.<
['a ', 'gravida ']
---- dans >Neque ipsum! rhoncus cras quam.<
['ipsum!', 'quam.']
```

Ce code permet de chercher toutes (`findall`) les occurrences de l'expression régulière, qui ici est définie par la chaîne :

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

digression : les raw-strings Pour anticiper un peu, signalons que cette façon de créer une chaîne en la préfixant par un `r` s'appelle une raw-string ; l'intérêt c'est de ne pas interpréter les backslashes `\`

On voit tout de suite l'intérêt sur un exemple :

```
[11]: print("sans raw-string\nun newline")
```

```
sans raw-string
un newline
```

```
[12]: print(r"dans\nunraw-string")
```

```
dans\nunraw-string
```

Comme vous le voyez dans une chaîne "normale" les caractères backslash ont une signification particulière ; mais nous ce qu'on veut faire, quand on crée une expression régulière, c'est de laisser les backslashes intacts, car c'est à la couche de regex de les interpréter.

repreons 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`). Ceci est défini en fonction de votre LOCALE, on y reviendra.
- `[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

```
[13]: # 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))

      # NB: Comme vous le devinez, ici la notation for ... in ...
```

```
# permet de parcourir successivement tous les éléments de la séquence
```

```
---- dans >Lacus a donec, vitae gravida proin sociis.<
['La', 'a', 'vita', 'gravida']
---- dans >Neque ipsum! rhoncus cras quam.<
['ipsum', 'cra', 'quam']
```

split

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

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

```
---- dans >Lacus a donec, vitae gravida proin sociis.<
['Lacus', 'a', 'donec', 'vitae', 'gravida', 'proin', 'sociis', '']
---- dans >Neque ipsum! rhoncus cras quam.<
['Neque', 'ipsum', 'rhoncus', 'cras', 'quam', '']
```

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

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

```
---- dans >Lacus a donec, vitae gravida proin sociis.<
XLacusY XaY XdonecY, XvitaeY XgravidaY XproinY XsociisY.
---- dans >Neque ipsum! rhoncus cras quam.<
XNequeY XipsumY! XrhoncusY XcrasY XquamY.
```

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

```
[16]: 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}")
```

```
both compared with is → True
both compared with == → True
```

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. 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 dans une chaîne usuelle. 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

```
[17]: 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.

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

Si on applique cette expression régulière à toutes nos entrées

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

```
890hj000nnm890 → <re.Match object; span=(0, 14), match='890hj000nnm890'>
123abc456def789 → <re.Match object; span=(0, 15), match='123abc456def789'>
8090abababab879 → None
```

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

```
[20]: # pour simplement visualiser si on a un 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.

```
[21]: # la même chose mais un peu moins encombrant
print(f"REGEXP={regex1}\n")
for sample in samples:
    match = re.match(regex1, sample)
    print(f"{sample:>16} → {nice(match)}")
```

REGEXP=[0-9]+[A-Za-z]+[0-9]+[A-Za-z]+[0-9]+

```
890hj000nnm890 → Match!
123abc456def789 → Match!
8090abababab879 → no
```

Ici plutôt que d'utiliser les raccourcis comme `\w` j'ai préféré écrire explicitement les ensembles de caractères en jeu. De cette façon, on rend son code indépendant du LOCALE si c'est ce qu'on veut faire. Il y a deux morceaux qui interviennent tour à tour :

- `[0-9]+` signifie une suite de 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)

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

```
[22]: '123abc456def789'
```

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.

```
[23]: # la même regexp, mais on donne un nom au groupe de chiffres central
regex2 = "[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 :

```
[24]: print(re.match(regex2, haystack).group('needle'))
```

456

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

```
[25]: 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

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

```
REGEXP=(?P<id>[0-9]+) [A-Za-z]+(?P<needle>[0-9]+) [A-Za-z]+(?P=id)
```

```
890hj000nnm890 → Match!
123abc456def789 → no
8090abababab879 → no
```

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 librairie - Compilation des expressions régulières

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

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 librairie, lorsque vous avez le même pattern 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) ⇔ re.compile(regexp).match(sample)
```

Donc à chaque fois qu'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.

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

# imaginez 10**6 chaînes dans samples
for sample in samples:
    match = re.match(regex3, sample)
    print(f"{sample:>16} → {nice(match)}")
```

```
890hj000nnm890 → Match!
123abc456def789 → no
8090abababab879 → no
```

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

# on compile la chaîne en automate une seule fois
re_obj3 = re.compile(regex3)

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

```
890hj000nnm890 → Match!
123abc456def789 → no
8090abababab879 → no
```

Cette deuxième version ne compile qu'une fois la chaîne en automate, et donc est 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 occurrences (`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.

```
[29]: # 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.

```
[30]: match.string
```

```
[30]: '    Isaac Newton, physicist'
```

```
[31]: match.re
```

```
[31]: re.compile(r'(\w+) (?P<name>\w+)', re.UNICODE)
```

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**).

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

```
[32]: ('Isaac', 'Newton')
```

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

```
[33]: 'Isaac'
```

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

```
[34]: 'Newton'
```

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

```
[35]: 'Newton'
```

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

```
[36]: {'name': 'Newton'}
```

Comme on le voit pour l'accès par rang les indices commencent à 1 pour des raisons historiques (on pouvait déjà référencer \1 dans l'éditeur Unix sed à 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 - qui en général est une sous-chaîne de la chaîne de départ :

```
[37]: # la sous-chaîne filtrée
match.group(0)
```

```
[37]: 'Isaac Newton'
```

```
[38]: # la chaîne de départ
sample
```

```
[38]: '    Isaac Newton, physicist'
```

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

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

```
[39]: 'last_name Newton first_name Isaac'
```

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

```
[40]: # NB: seq[i:j] est une opération de slicing que nous verrons plus tard
# Elle retourne une séquence contenant les éléments de i à j-1 de seq
begin, end = match.span('name')
sample[begin:end]
```

```
[40]: 'Newton'
```

Les différents modes (flags)

Enfin il faut noter qu'on peut passer à `re.compile` un certain nombre de 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, eh bien, ne pas faire la différence entre minuscules et majuscules,
- `UNICODE` (alias `U`) pour rendre les séquences `\w` et autres basées sur les propriétés des caractères dans la norme Unicode,
- `LOCALE` (alias `L`) cette fois `\w` dépend du `locale` courant,
- `MULTILINE` (alias `M`), et
- `DOTALL` (alias `S`) - pour ces deux derniers flags, voir la discussion à la fin du complément.

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

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

```
MAX_REPEAT 0 MAXREPEAT
  LITERAL 97
MAX_REPEAT 1 MAXREPEAT
  LITERAL 98

0. INFO 4 0b0 1 MAXREPEAT (to 5)
5: REPEAT_ONE 6 0 MAXREPEAT (to 12)
9.  LITERAL_UNI_IGNORE 0x61 ('a')
11.  SUCCESS
12: REPEAT_ONE 6 1 MAXREPEAT (to 19)
16.  LITERAL_UNI_IGNORE 0x62 ('b')
18.  SUCCESS
19: SUCCESS
```

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

```
a*b+ -> Match!
```

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 précédant 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, qui peuvent alors dépendre de l'environnement (UNICODE et LOCALE) 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).

```
[43]: 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)}")
```

```
abcd / abcd          → Match!
abcd / ab[cd][cd]    → Match!
abcd / ab[a-z]d      → Match!
abcd / abc.          → Match!
abcd / abc\\.        → no
```

Pour ce dernier exemple, comme on a backslashé le . il faut que la chaîne en entrée contienne vraiment un .

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

Match!

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 reconnaître 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 |

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

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

```
abc / abc|def → Match!
def / abc|def → Match!
aef / abc|def → no
```

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 chaîne ;
- `\A` a le même sens (sauf en mode MULTILINE), et je le recommande de préférence à `^` qui est déjà pas mal surchargé ;
- `$` matche une fin de ligne ;
- `\Z` est voisin de `$` mais pas tout à fait identique.

Reportez-vous à la documentation pour le détails des différences. Attention aussi à entrer le `^` correctement, il vous faut le caractère ASCII et non un voisin dans la ménagerie Unicode.

```
[46]: 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},"
          f" search → {nice(search)}")
```

```
abcd / bc      match → no      , search → Match!
abcd / \Aabc   match → Match! , search → Match!
abcd / ^abc    match → Match! , search → Match!
abcd / \Abc    match → no      , search → no
abcd / ^bc     match → no      , search → no
abcd / bcd\Z   match → no      , search → Match!
abcd / bcd$    match → no      , search → Match!
abcd / bc\Z    match → no      , search → no
abcd / bc$     match → no      , search → no
```

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

Parenthésier - (grouper)

Pour pouvoir faire des montages élaborés, il faut pouvoir parenthésier.

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

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

```
abcf → Match!
adef → Match!
abef → no
abf  → no
```

Les parenthèses jouent un rôle additionnel de groupe, ce qui signifie qu'on peut retrouver le texte correspondant à l'expression régulière comprise dans les (). Par exemple, pour le premier match

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

```
abcf, a(bc|de)f → ('bc',)
```

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 n'importe quel nombre, même nul, d'occurrences - par exemple, (ab)* pour indiquer '' 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 occurrence - 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.

```
[50]: # NB: la construction
# [op(elt) for elt in iterable]
# est une compréhension de liste que nous étudierons plus tard.
# Elle retourne une liste contenant les résultats
# de l'opération op sur chaque élément de la liste de départ

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)}")
```

```

/ \A(ab)*\Z      → Match!
ab / \A(ab)*\Z    → Match!
ababab / \A(ab)*\Z → Match!
abababab / \A(ab)*\Z → Match!
baba / \A(ab)*\Z  → no
/ \A(ab)+\Z      → no
ab / \A(ab)+\Z    → Match!
ababab / \A(ab)+\Z → Match!
abababab / \A(ab)+\Z → Match!
baba / \A(ab)+\Z  → no
/ \A(ab){3}\Z    → no
ab / \A(ab){3}\Z  → no
ababab / \A(ab){3}\Z → Match!
abababab / \A(ab){3}\Z → no
baba / \A(ab){3}\Z → no
/ \A(ab){3,4}\Z  → no
ab / \A(ab){3,4}\Z → no
ababab / \A(ab){3,4}\Z → Match!
abababab / \A(ab){3,4}\Z → Match!
baba / \A(ab){3,4}\Z → no

```

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 matche 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.

```

[51]: # 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)
```

```
[51]: '<h1>Title</h1>'
```

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 matche, dans une approche dite non-greedy, avec les opérateurs suivants :

- `*?` : `*` mais non-greedy,
- `+?` : `+` mais non-greedy,
- `??` : `?` mais non-greedy,

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

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

```
[52]: '<h1>'
```

S'agissant du traitement des fins de ligne

Il peut être utile, pour conclure cette présentation, de préciser un peu le comportement de la librairie vis-à-vis des fins de ligne.

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

Le module `re` en garde des traces, puisque

```
[53]: # un exemple de traitement des 'newlines'
sample = """une entrée
sur
plusieurs
lignes
"""
```

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

```
[54]: ('une entrée',)
```

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 côté 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 newlines

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

```
[55]: ('une entrée\nsur\nplusieurs\nlignes\n',)
```

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

```
[56]: # (depuis Python 3) sans mettre de flag, \w matche l'Unicode
match = re.compile("([\w ]*").match(sample)
match.groups()
```

```
[56]: ('une entrée',)
```

```
[57]: # pour matcher les caractères ASCII avec \w
# il faut mentionner le flag ASCII re.A
match = re.compile("([\w ]*", flags=re.A).match(sample)
match.groups()
```

```
[57]: ('une entr',)
```

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

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

```
[58]: ('une entrée\nsur\nplusieurs\nlignes\n',)
```

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/>, il en existe beaucoup d'autres.

Un élève, qui a eu notamment des soucis avec le `\w` sur pythex.org (dont, on l'a vu, la signification dépend du locale de la machine hôte) recommande pour sa part <https://regex101.com/> : > Ce site est très didactique et lui reconnaît les caractères accentués sur un `\w` sans rajouter de flag (même si cette option est possible).

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 expressions 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,
- ...

2.6 w2-s2-x1-expressions-regulieres

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, il pourra être profitable d'avoir sous la main :

- la [documentation officielle](#) ;
- et <https://regex101.com/> (par exemple) qui permet de mettre au point de manière interactive, et donc d'avoir un retour presque immédiat, pour accélérer la mise au point.

2.6.1 Exercice - niveau intermédiaire (1)

Identificateurs Python

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

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 `_`.

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

```
GridBox(children=(HTML(value='<span style="font-size:medium;">chaîne</span>', _dom_classes=('head
```

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

regexpythonid = r"votre_regexp"
```

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

# NOTE
```

```
# auto-exec-for-latex has skipped execution of this cell
```

2.6.2 Exercice - niveau intermédiaire (2)

Lignes avec nom et prénom

```
[4]: # 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.

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

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>chaîne</span>', _dom_classes=('head
```

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 constitue 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.

Dans la correction - et ce sera pareil pour tous les exercices de regex où on demande des groupes - la correction affiche uniquement les groupes demandés ; ici on va vous montrer les groupes `nom` et `prenom` ; vous avez parfaitement le droit d'utiliser des groupes supplémentaires, nommés ou pas d'ailleurs, dans votre propre regex.

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

regexp_agenda = r"(?P<prenom>)(?P<nom>)\Z"
```

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

2.6.3 Exercice - niveau intermédiaire (3)

Numéros de téléphone

```
[7]: # 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 :

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

GridBox(children=(HTML(value='chaîne', _dom_classes=('head

```
[9]: # votre regexp
# à nouveau il faut terminer la regexp par \Z
regexp_phone = r"(0|\+(?P<intl>33))(?P<number>[0-9]{9})\Z"
```

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

2.6.4 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 :

```
[10]: import re
```

```
[11]: # 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)
```

```
[11]: 'ABC'
```

```
[12]: # ou cela

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

[12]: 'ABC'

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

```
/Users/tparment/miniconda3/envs/flotpython-course/lib/python3.7/site-packag
es/ipykernel_launcher.py:3: DeprecationWarning: Flags not at the start o
f the expression 'abc(?i)'
This is separate from the ipykernel package so we can avoid doing imports
until
```

[13]: 'ABC'

Pour plus de précisions sur ce trait, que nous avons laissé de côté 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 une 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.

```
[14]: # pour charger l'exercice
      from corrections.regex_url import exo_url
```

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

```
GridBox(children=(HTML(value='<span style="font-size:small;">chaîne</span>', _dom_classes=('header
```

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

      regexp_url = "<votre_regexp>"
```

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

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

2.7 w2-s3-c1-slices

Les slices en Python

2.7.1 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.

```
[1]: chaine = "abcdefghijklmnopqrstuvwxy"
      print(chaine)
```

```
abcdefghijklmnopqrstuvwxy
```

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 :

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

```
[2]: 'cdef'
```

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 **début** est inclus ;
- le second indice **fin** est exclu ;

— on obtient en tout **fin-début** 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 :

	0	1	2	3	4	5	6	7	8	9
	a	b	c	d	e	f	g	h	i	j
[0:3]	x	x	x	[
[3:7]					x	x	x	x	[
[0:7]	x	x	x	x	x	x	x	[

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

```
[3]: True
```

Bornes omises On peut omettre une borne :

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

```
[4]: 'abcdef'
```

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

```
[5]: 'yz'
```

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

```
[6]: 'abcdefghijklmnopqrstuvwxyz'
```

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


```
[7]: chaine[3:-3]
```

```
[7]: 'defghijklmnopqrstuvw'
```

```
[8]: chaine[-3:]
```

```
[8]: 'xyz'
```

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 :

```
[9]: # 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]
```

```
[9]: 'dfhjlnprtv'
```

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.

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

```
[10]: 'dfhjlnprtv'
```

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 :

```
[11]: chaine[-3:3]
```

```
[11]: ''
```

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

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

```
[12]: 'xvtrpnljhf'
```

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 possibilités qui sont offertes avec les slices.

Listes Par exemple sur les listes :

```
[13]: liste = [1, 2, 4, 8, 16, 32]
      liste
```

```
[13]: [1, 2, 4, 8, 16, 32]
```

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

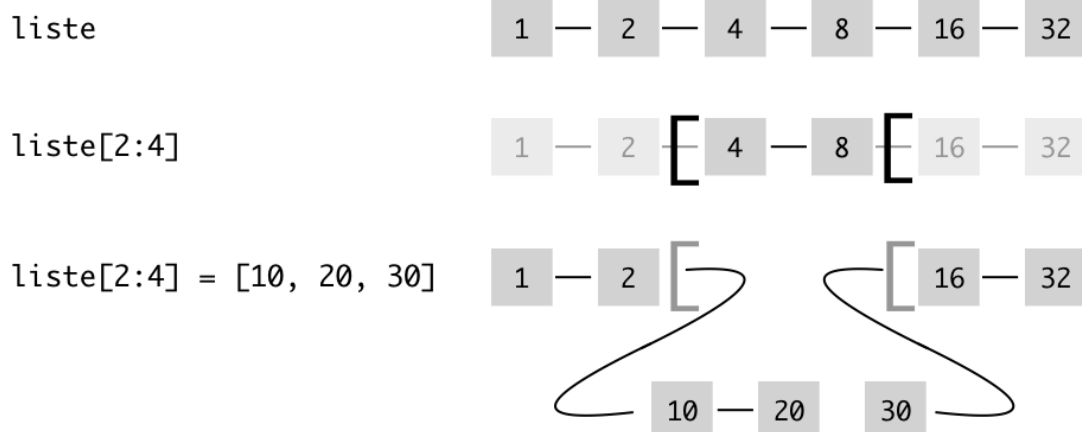
```
[14]: [32, 8]
```

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

```
[15]: liste[2:4] = [10, 20, 30]
      liste
```

```
[15]: [1, 2, 10, 20, 30, 16, 32]
```

Voici une représentation imagée de ce qui se passe lorsqu'on exécute cette dernière ligne de code ; cela revient en quelque sorte à remplacer la slice à gauche de l'affectation (ici `liste[2:4]`) par la liste à droite de l'affectation (ici `[10, 20, 30]`) - ce qui a, en général, pour effet de modifier la longueur de la liste.



2.7.2 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 détail, voici un aperçu de ce que l'on peut faire avec des slices sur des objets **numpy** :

```
[16]: # 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
```

```
[16]: array([1, 2, 3, 4, 5])
```

```
[17]: # 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
```

```
[17]: array([[11, 12, 13, 14, 15],
             [21, 22, 23, 24, 25],
             [31, 32, 33, 34, 35],
             [41, 42, 43, 44, 45],
             [51, 52, 53, 54, 55]])
```

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

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

```
[18]: array([[22, 23, 24],
             [32, 33, 34],
             [42, 43, 44]])
```

On peut bien sûr également utiliser un pas :

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

```
[19]: array([[11, 15],
             [51, 55]])
```

Ou bien retourner complètement dans une direction :

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

```
[20]: array([[51, 52, 53, 54, 55],
             [41, 42, 43, 44, 45],
             [31, 32, 33, 34, 35],
             [21, 22, 23, 24, 25],
             [11, 12, 13, 14, 15]])
```

2.8 w2-s4-c1-listes

Méthodes spécifiques aux listes

2.8.1 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` :

```
[ ]: help(list)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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 :

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

```
liste [0, 1, 2, 3]
```

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 :

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

```
liste [0, 1, 2, 3, 'ap']
```

`extend`

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

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

```
liste [0, 1, 2, 3, 'ap', 'ex1', 'ex2']
```

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.

```
[4]: # 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)
```

```
[0, 1, 2]
```

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

```
[10, 11, 12]
```

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

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

```
a1 [0, 1, 2]
a2 [10, 11, 12]
a3 [0, 1, 2, 10, 11, 12]
```

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 :

```
[8]: %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.

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

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

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

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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 crée 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` :

```
[9]: liste
```

```
[9]: [0, 1, 2, 3, 'ap', 'ex1', 'ex2']
```

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 :

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

```
liste [0, 1, '1 bis', 2, 3, 'ap', 'ex1', 'ex2']
```

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 :

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

```
liste [0, 1, '1 bis', 2, 3, '3 bis', 'ap', 'ex1', 'ex2']
```

`remove`

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

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

```
liste [0, 1, '1 bis', 2, '3 bis', 'ap', 'ex1', 'ex2']
```

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 :

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

```
popped 0 liste [1, '1 bis', 2, '3 bis', 'ap', 'ex1', 'ex2']
```

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

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

```
popped ex2 liste [1, '1 bis', 2, '3 bis', 'ap', 'ex1']
```

reverse

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

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

```
liste ['ex1', 'ap', '3 bis', 2, '1 bis', 1]
```

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

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

```
liste2 [1, '1 bis', 2, '3 bis', 'ap', 'ex1']
```

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 :

```
[17]: # fonctionne dans tous les environnements Python
      help(list.pop)
```

Help on method_descriptor:

```
pop(self, index=-1, /)
    Remove and return item at index (default last).

    Raises IndexError if list is empty or index is out of range.
```

```
[18]: # spécifique aux notebooks
      # l'affichage obtenu est légèrement différent
      # tapez la touche 'Esc' - ou cliquez la petite croix
      # pour faire disparaître 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 mot connu dans l'environnement, vous voyez apparaître un dialogue avec les noms connus qui commencent par ce mot ici `li` ; utilisez les flèches pour choisir, et 'Return' pour sélectionner.

```
[ ]: # placez votre curseur à la fin de la ligne après 'li'
      # et appuyez sur la touche 'Tab'
      li

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

2.9 w2-s4-c2-listes-mutables

Objets mutables et objets immuables

2.9.1 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.

```
[1]: # 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](#) :

```
[ ]: %%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`

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

```
[2]: # à se stade avec des chaines on observe
s1 = 'abc'
s2 = s1
s1 += 'def'
print(s1)
print(s2)
```

```
abcdef
abc
```

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 :

```
[ ]: %%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`

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

```
[3]: # alors qu'avec les listes on observe
liste1 = ['a', 'b', 'c']
liste2 = liste1
# on modifie l'objet
liste1 += ['d', 'e', 'f']
print(liste1)
print(liste2)
```

```
['a', 'b', 'c', 'd', 'e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
```

Conclusion

Ce comportement n'est pas propre à l'usage de l'opérateur `+=`, 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.

2.10 w2-s4-c3-tris-de-liste-1

Tris de listes

2.10.1 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 :

```
[1]: liste = [8, 7, 4, 3, 2, 9, 1, 5, 6]
      print('avant tri', liste)
      liste.sort()
      print('apres tri', liste)
```

avant tri [8, 7, 4, 3, 2, 9, 1, 5, 6]

apres tri [1, 2, 3, 4, 5, 6, 7, 8, 9]

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.

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 :

```
[2]: %load_ext ipythontutor
```

```
[ ]: %%ipythontutor height=200 ratio=0.8
      liste = [3, 2, 9, 1]
      liste.sort()

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

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 :

```
[ ]: %%ipythontutor height=200 ratio=0.8
      liste1 = [3, 2, 9, 1]
      liste2 = sorted(liste1)

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

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 :

```
[3]: liste = [8, 7, 4, 3, 2, 9, 1, 5, 6]
      print('avant tri', liste)
      liste.sort(reverse=True)
      print('après tri décroissant', liste)
```

```
avant tri [8, 7, 4, 3, 2, 9, 1, 5, 6]
après tri décroissant [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

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 :

```
[4]: liste = ['spam', 'egg', 'bacon', 'beef']
      liste.sort()
      print('après tri', liste)
```

```
après tri ['bacon', 'beef', 'egg', 'spam']
```

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 :

```
[5]: # deux caractères minuscules se comparent
      # comme on s'y attend
      'a' < 'z'
```

```
[5]: True
```

Bon, mais par contre :

```
[6]: # si l'un est en minuscule et l'autre en majuscule,
      # ce n'est plus le cas
      'Z' < 'a'
```

```
[6]: True
```

Ce qui à son tour explique ceci :

```
[7]: # la conséquence de 'Z' < 'a', c'est que
      liste = ['abc', 'Zoo']
      liste.sort()
```

```
print(liste)
```

```
['Zoo', 'abc']
```

Et lorsque les chaînes contiennent des espaces ou autres ponctuations, le résultat du tri peut paraître surprenant :

```
[8]: # attention ici notre première chaîne commence par une espace
      # et le caractère 'Espace' est plus petit
      # que tous les autres caractères imprimables
      liste = [' zoo', 'ane']
      liste.sort()
      print(liste)
```

```
[' zoo', 'ane']
```

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.

2.11 w2-s5-c1-indentations

Indentations en Python

2.11.1 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 :

```
[1]: # la convention la plus généralement utilisée
      # consiste à utiliser une indentation de 4 espaces
      if 'g' in 'egg':
          print('OUI')
      else:
          print('NON')
```

OUI

Voyons tout de suite comment on pourrait écrire plusieurs tests imbriqués :

```
[2]: 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')
```

a mais pas 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.

2.11.2 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 environnements 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 :

```
[3]: if 'a' in entree:
      if 'b' in entree:
          cas11 = True
          print('a et b')
      else:
          cas12 = True
          print('a mais pas b')
```

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
^I^Iprint('a et b')
^Ielse:
^I^Icas12 = True
^I^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 sauvegardé 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.

2.11.3 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 :

```
[4]: # code accepté mais pas du tout recommandé
if 'a' in 'pas du tout recommande':
    succes = True
    print('OUI')
else:
    print('NON')
```

OUI

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 :

```
[ ]: # 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')

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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.

2.12 w2-s5-c2-presentation

Bonnes pratiques de présentation de code

2.12.1 Complément - niveau basique

La PEP-008

On trouve [dans la PEP-008 \(en anglais\)](#) les conventions de codage qui s'appliquent à toute la librairie 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 librairie 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.

```
[1]: from modtools import show_module_html
import pprint
show_module_html(pprint)
```

```
[1]: <IPython.core.display.HTML object>
```

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`~~
 Ni
~~`x=-y+z`~~
 Ni encore
~~`x=y+-z`~~

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 (un espace en trop avant la parenthèse ouvrante)
~~`def foo (x, y, z):`~~
 Ni surtout comme ceci (pas d'espace entre les paramètres)
~~`def foo(x,y,z):`~~
- La même règle s'applique naturellement aux appels de fonction :
`foo(x, y, z)`
 et non pas
~~`foo(x,y,z)`~~
 ni
~~`def foo(x,y,z):`~~

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 librairie standard.

Coupsures 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 (\)

```
[2]: show_module_html(pprint,
                      beg="def pprint",
                      end="def pformat")
```

```
[2]: <IPython.core.display.HTML object>
```

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 plusieurs 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 :

```
[3]: show_module_html(pprint,
                        beg="components), readable, recursive",
                        end="elif len(object) ",
                        lineno_width=3)
```

```
[3]: <IPython.core.display.HTML object>
```

Dans ce fragment au contraire, vous voyez en ligne 521 qu'il a fallu cette fois insérer un backslash \ comme caractère de continuation pour que l'instruction puisse se poursuivre en ligne 522.

Coupages de lignes - épilogue

Dans tous les 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.

2.12.2 Complément - niveau intermédiaire

Outils liés à PEP008

Il existe plusieurs outils liés à la PEP0008, 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 :

- l'outil `pep8` pour vérifier, et
- l'outil `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`.

2.13

w2-s5-c3-pass

L'instruction `pass`

2.13.1 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 :

```
[1]: # 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 :

```
[2]: liste = list(range(10))  
print('avant', liste)  
while liste.pop() != 5:  
    pass  
print('après', liste)
```

avant [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

après [0, 1, 2, 3, 4]

On voit qu'ici encore l'instruction `pass` a toute son utilité.

2.13.2 Complément - niveau intermédiaire

Un if sans then

```
[3]: # on utilise dans ces exemples une condition fausse  
condition = False
```

Imaginons qu'on parte d'un code hypothétique qui fasse ceci :

```
[4]: # la version initiale  
if condition:  
    print("non")  
else:  
    print("bingo")
```

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 réécrit 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 :

```
[5]: # on peut s'en sortir en ajoutant une instruction pass
if condition:
    # print "non"
    pass
else:
    print("bingo")
```

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 :

```
[6]: class Foo:
    pass
```

```
[7]: foo = Foo()
```

2.14 w2-s6-c1-valeur-de-retour

Fonctions avec ou sans valeur de retour

2.14.1 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 :

```
[1]: def affiche_carre(n):
    print("le carre de", n, "vaut", n*n)
```

qui s'utiliserait comme ceci :

```
[2]: affiche_carre(12)
```

le carre de 12 vaut 144

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.

L'instruction `return`

Voici comment on pourrait écrire une fonction `carre` qui retourne (on dit aussi renvoie) le carré de son argument :

```
[3]: def carre(n):  
      return n*n  
  
      if carre(8) <= 100:  
          print('petit appartement')
```

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 :

```
[4]: # ce premier appel provoque l'impression d'une ligne  
      retour = affiche_carre(15)
```

le carre de 15 vaut 225

```
[5]: # voyons ce qu'a retourné la fonction affiche_carre  
      print('retour =', retour)
```

retour = None

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 :

```
[6]: 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  
    # NB: elif est un raccourci pour else if  
    # c'est utile pour éviter une indentation excessive  
    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 :

```
[7]: for test in [-2, 1, 2, 4, 19, 35]:  
    print(f"premier({test:2d}) = {premier(test)}")
```

```
premier(-2) = None  
premier( 1) = False  
premier( 2) = True  
premier( 4) = False  
premier(19) = True  
premier(35) = False
```

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 :

```
[8]: 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 trouvé 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 tout cas, les deux versions sont tout à fait équivalentes, comme on le voit ici :

```
[9]: 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")
```

```
pour n = -2 : premier → None
      premier_sans_else → None
```

```
pour n =  2 : premier → True
      premier_sans_else → True
```

```
pour n =  4 : premier → False
      premier_sans_else → False
```

```
pour n = 19 : premier → True
      premier_sans_else → True
```

```
pour n = 35 : premier → False
      premier_sans_else → False
```

Digression sur les chaînes

Vous remarquerez dans cette dernière cellule, si vous regardez bien le paramètre de `print`, qu'on peut accoler deux chaînes (ici deux f-strings) sans même les ajouter ; un petit détail pour éviter d'alourdir le code :

```
[10]: # quand deux chaînes apparaissent immédiatement
      # l'une après l'autre sans opérateur, elles sont concaténées
      "abc" "def"
```

```
[10]: 'abcdef'
```

2.15

w2-s6-x1-label

Formatage des chaînes de caractères

2.15.1 Exercice - niveau basique

```
[1]: # charger l'exercice
      from corrections.exo_label import exo_label
```

Vous devez écrire une fonction qui prend deux arguments :

- une chaîne de caractères qui désigne le prénom d'un élève ;
- un entier qui indique la note obtenue.

Elle devra retourner une chaîne de caractères selon que la note est

- $0 \leq \text{note} < 10$
- $10 \leq \text{note} < 16$
- $16 \leq \text{note} \leq 20$

comme on le voit sur les exemples :

```
[2]: exo_label.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;">appel</span>', _dom_classes=('header
```

```
[3]: # à vous de jouer
      def label(prenom, note):
          "votre code"
```

```
[ ]: # pour corriger
      exo_label.correction(label)

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```


2.16

w2-s6-x2-inconnue

Séquences

2.16.1 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`.

```
[1]: # nous allons tirer profit ici d'une
      # constante définie dans le module string
      import string
      chaine = string.ascii_lowercase

      # et voici sa valeur
      print(chaine)
```

abcdefghijklmnopqrstuvwxyz

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" :

```
[2]: chaine[3:6] == "def"
```

[2]: True

1) Écrivez une slice pour obtenir "vwx" (n'hésitez pas à utiliser les indices négatifs) :

```
[ ]: chaine[ <votre_code> ] == "vwx"

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

2) Une slice pour obtenir "wxyz" (avec une seule constante) :

```
[ ]: chaine[ <votre_code> ] == "wxyz"

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

3) Une slice pour obtenir "dfhjlnprtvxz" (avec deux constantes) :

```
[ ]: chaine[ <votre_code> ] == "dfhjlnprtvxz"

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

4) Une slice pour obtenir “xurolifc” (avec deux constantes) :

```
[ ]: chaine[ <votre_code> ] == "xurolifc"

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

2.16.2 Exercice - niveau intermédiaire

Longueur

```
[3]: # 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 :

```
[4]: len('abcd')
```

```
[4]: 4
```

```
[5]: # 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.

```
[ ]: # correction
exo_inconnue.correction(inconnue)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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.

2.17 w2-s6-x3-laccess

Listes

2.17.1 Exercice - niveau basique

```
[1]: 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.

```
[2]: exo_laccess.example()
```

GridBox(children=(HTML(value='appel', _dom_classes=('header

```
[3]: # écrivez votre code ici
def laccess(liste):
    return "votre code"
```

```
[ ]: # pour le corriger
exo_laccess.correction(laccess)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Une fois que votre code fonctionne, vous pouvez regarder si par hasard il marcherait aussi avec des chaînes :

```
[4]: from corrections.exo_laccess import exo_laccess_strings
```

```
[ ]: exo_laccess_strings.correction(laccess)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

2.18 w2-s6-x4-if-et-def

Instruction **if** et fonction **def**

2.18.1 Exercice - niveau basique

Fonction de divisibilité

```
[1]: # chargement de l'exercice
from corrections.exo_divisible import exo_divisible
```

L'exercice consiste à écrire une fonction baptisée `divisible` qui retourne une valeur booléenne, qui indique si un des deux arguments est divisible par l'autre.

Vous pouvez supposer les entrées `a` et `b` entiers et non nuls, mais pas forcément positifs.

```
[2]: def divisible(a, b):
      "<votre_code>"
```

Vous pouvez à présent tester votre code en évaluant ceci, qui écrira un message d'erreur si un des jeux de test ne donne pas le résultat attendu.

```
[ ]: # tester votre code
exo_divisible.correction(divisible)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

2.18.2 Exercice - niveau basique

Fonction définie par morceaux

```
[3]: # chargement de l'exercice
from corrections.exo_morceaux import exo_morceaux
```

On veut définir en Python une fonction qui est définie par morceaux :

$$f : x \longrightarrow \begin{cases} -x - 5 & \text{si } x \leq -5 \\ 0 & \text{si } x \in [-5, 5] \\ \frac{1}{5}x - 1 & \text{si } x \geq 5 \end{cases}$$

```
[4]: # donc par exemple
     exo_morceaux.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[5]: # à vous de jouer

def morceaux(x):
    return 0 # "votre code"
```

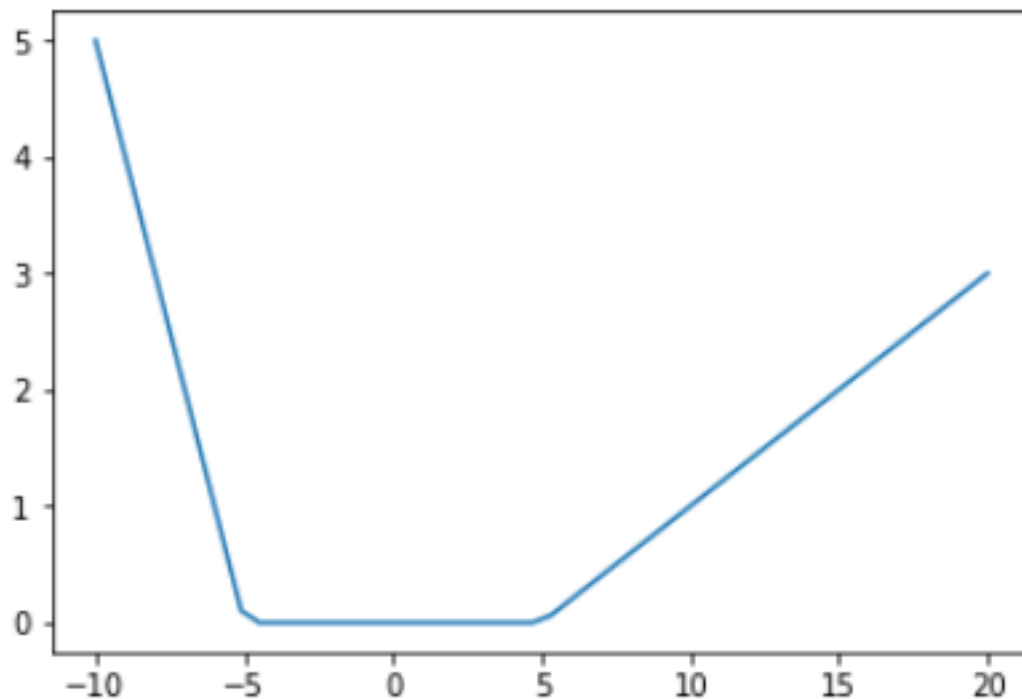
```
[ ]: # pour corriger votre code
     exo_morceaux.correction(morceaux)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Représentation graphique

L'exercice est terminé, mais nous allons maintenant voir ensemble comment vous pourriez visualiser votre fonction.

Voici ce qui est attendu comme courbe pour morceaux (image fixe) :



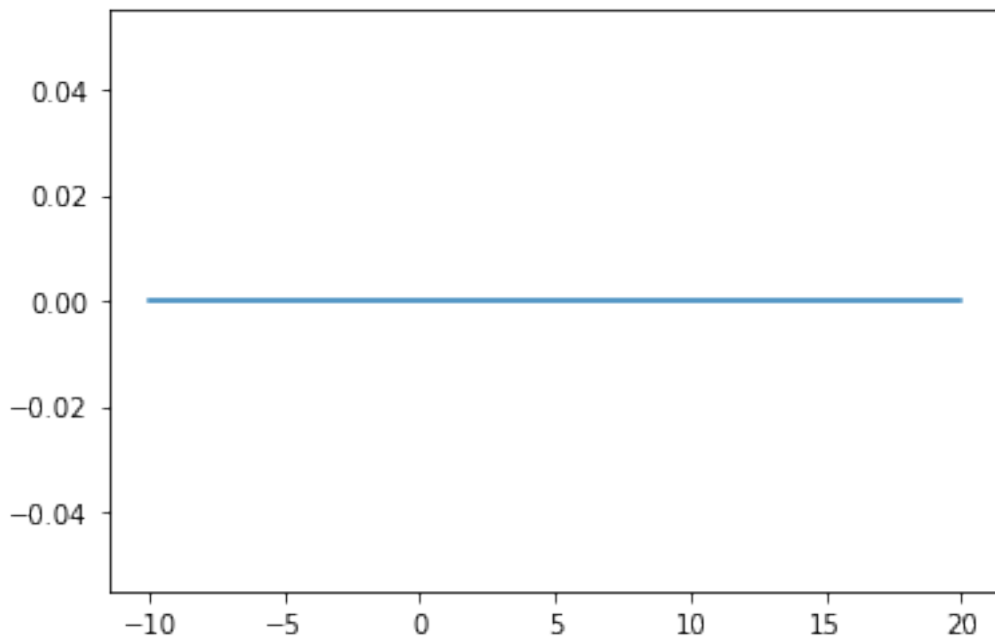
En partant de votre code, vous pouvez produire votre propre courbe en utilisant `numpy` et `matplotlib` comme ceci :

```
[6]: # on importe les bibliothèques
     import numpy as np
     import matplotlib.pyplot as plt
```

```
[7]: # un échantillon des X entre -10 et 20
X = np.linspace(-10, 20)

# et les Y correspondants
Y = np.vectorize(morceaux)(X)
```

```
[8]: # on n'a plus qu'à dessiner
plt.plot(X, Y)
plt.show()
```



2.19 w2-s6-x5-wc

Comptage dans les chaînes

2.19.1 Exercice - niveau basique

Nous remercions Benoit Izac pour cette contribution aux exercices.

2.19.2 La commande UNIX wc(1)

Sur les systèmes de type UNIX, la commande `wc` permet de compter le nombre de lignes, de mots et d'octets (ou de caractères) présents sur l'entrée standard ou contenus dans un fichier.

L'exercice consiste à écrire une fonction nommée `wc` qui prendra en argument une chaîne de caractères et retournera une liste contenant trois éléments :

1. le nombre de lignes (plus précisément le nombre de retours à la ligne) ;

2. le nombre de mots (un mot étant séparé par des espaces);
3. le nombre de caractères (on utilisera uniquement le jeu de caractères ASCII).

```
[1]: # chargement de l'exercice
from corrections.exo_wc import exo_wc
```

```
[2]: # exemple
exo_wc.example()
```

GridBox(children=(HTML(value='appel', _dom_classes=('header

Indice : nous avons vu rapidement la boucle `for`, sachez toutefois qu'on peut tout à fait résoudre l'exercice en utilisant uniquement la bibliothèque standard.

Remarque : usuellement, ce genre de fonctions retournerait plutôt un tuple qu'une liste, mais comme nous ne voyons les tuples que la semaine prochaine..

À vous de jouer :

```
[3]: # la fonction à implémenter
def wc(string):
    # remplacer pass par votre code
    pass
```

```
[ ]: # correction
exo_wc.correction(wc)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

2.20 w2-s7-x1-liste-p

Compréhensions (1)

2.20.1 Exercice - niveau basique

Liste des valeurs d'une fonction

```
[1]: # 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 $P(x)$.

```
[2]: # voici un exemple de ce qui est attendu
exo_liste_P.example()
```

GridBox(children=(HTML(value='arguments', _dom_classes=('he

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) :

```
[3]: def P(x):
      "<votre code>"

      def liste_P(liste_x):
          "votre code"

      # NOTE:
      # auto-exec-for-latex has used hidden code instead
```

Et vous pouvez le vérifier en évaluant cette cellule :

```
[ ]: # pour vérifier votre code
      exo_liste_P.correction(liste_P)

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

2.20.2 Récréation

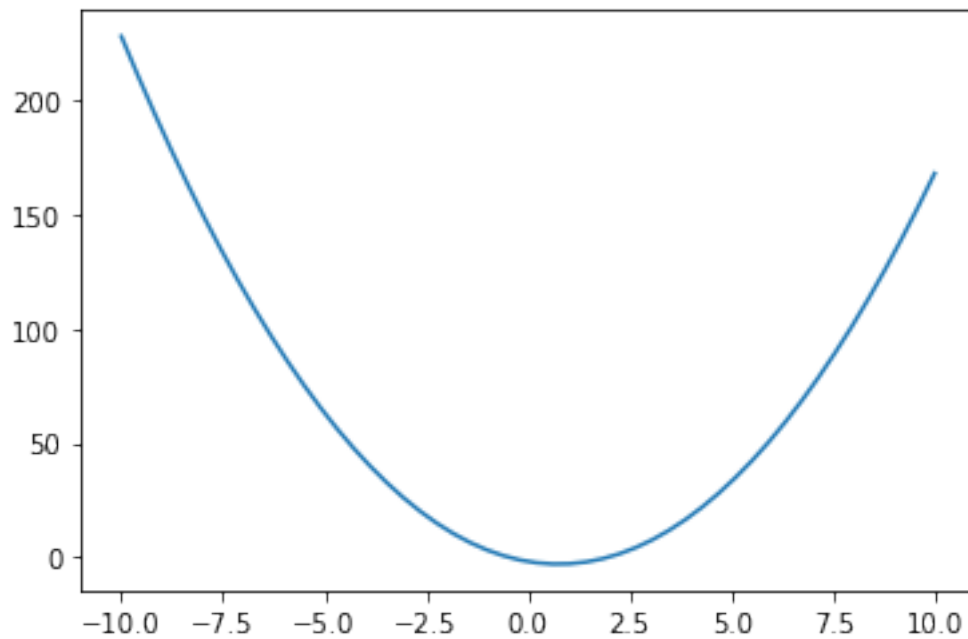
Si vous avez correctement implémenté la fonction `liste_P` telle que demandé dans le premier exercice, vous pouvez visualiser le polynôme P en utilisant `matplotlib` avec le code suivant :

```
[4]: # on importe les bibliothèques
      import numpy as np
      import matplotlib.pyplot as plt
```

```
[5]: # un échantillon des X entre -10 et 10
      X = np.linspace(-10, 10)

      # et les Y correspondants
      Y = liste_P(X)
```

```
[6]: # on n'a plus qu'à dessiner
      plt.plot(X, Y)
      plt.show()
```

2.21 w2-s7-x2-carre

Compréhensions (2)

2.21.1 Exercice - niveau intermédiaire

Mise au carré

```
[1]: # 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 la fonction polynomiale du notebook précédent. Cette fois, chaque ligne contient, séparés par des points-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.

```
[2]: # exemples
exo_carre.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>arguments</span>', _dom_classes=('he
```

```
[3]: # écrivez votre code ici
def carre(ligne):
    "<votre_code>"
```

```
[ ]: # pour corriger  
exo_carre.correction(carre)  
  
# NOTE  
# auto-exec-for-latex has skipped execution of this cell
```

Chapitre 3

Renforcement des notions de base, références partagées

3.1 w3-s1-cl-fichiers

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 :

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

Les modes d'ouverture

Les modes d'ouverture les plus utilisés sont :

- 'r' (la chaîne contenant l'unique caractère `r`) pour ouvrir un fichier en lecture seulement ;
- 'w' en écriture seulement ; le contenu précédent du fichier, s'il existait, est perdu ;
- 'a' en écriture seulement ; mais pour ajouter du contenu en fin de fichier.

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

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

```
sortie.write(f"{i}\n")
```

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

```

```
0
1
100
101

```

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

- ouvrir le fichier en lecture et en écriture (mode +);
- ouvrir le fichier en mode binaire (mode b).

Ces variantes sont décrites dans [la section sur la fonction built-in open](#) dans la documentation Python.

3.1.2 Complément - niveau intermédiaire

Un fichier est un itérateur

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

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

```

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

```

```
True

```

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

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

```

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

```

3.1.3 Complément - niveau avancé

Autres méthodes

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

Digression - `repr()`

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

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

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

```
abc
def
```

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

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

Lire un contenu - bas niveau

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

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

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

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

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

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

Bloc 1 >>'100\n'<<

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

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

La méthode `flush`

Les entrées-sorties sur fichier sont bien souvent bufferisées par le système d'exploitation. Cela signifie qu'un appel à `write` ne provoque pas forcément une écriture immédiate, car pour des raisons de performance on attend d'avoir suffisamment de matière avant d'écrire sur le disque.

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

Fichiers textuels et fichiers binaires

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

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

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

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

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

- ouvrir le fichier en mode binaire - pour cela on ajoute le caractère `b` au mode d'ouverture;
- et on peut alors interagir avec le fichier avec des objets de type `bytes`

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

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

```
[14]: # phase 2: on ouvre le fichier en mode binaire
with open('strbytes', 'rb') as bytesfile:
    # on lit tout le contenu
```

```

octets = bytesfile.read()
# qui est de type bytes
print("on a lu un objet de type", type(octets))
# si on regarde chaque octet un par un
for i, octet in enumerate(octets):
    print(f"{i} → {repr(chr(octet))} [{hex(octet)}]")

```

```

on a lu un objet de type <class 'bytes'>
0 → 'd' [0x64]
1 → 'Ã' [0xc3]
2 → '©' [0xa9]
3 → 'j' [0x6a]
4 → 'Ã' [0xc3]
5 → '\xa0' [0xa0]
6 → ' ' [0x20]
7 → 'l' [0x6c]
8 → '"' [0x27]
9 → 'Ã' [0xc3]
10 → '©' [0xa9]
11 → 't' [0x74]
12 → 'Ã' [0xc3]
13 → '©' [0xa9]
14 → '\n' [0xa]

```

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

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

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

```

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

```

```

en mode texte, 11 caractères
en mode binaire, 15 octets

```

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

Pour en savoir plus

Pour une description exhaustive vous pouvez vous reporter :

- au [glossaire sur la notion de object file](#),
- et aussi et surtout au [module io](#) qui décrit plus en détail les fonctionnalités disponibles.

3.2 w3-s1-c2-utilitaires-sur-fichiers

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 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 une interface unique et moderne, que nous recommandons évidemment d'utiliser pour du nouveau code.

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

- `os.path.join` ajoute `'/'` ou `'\\'` entre deux morceaux de chemin, selon l'OS
- `os.path.basename` trouve le nom de fichier dans un chemin
- `os.path.dirname` trouve le nom du directory dans un chemin
- `os.path.abspath` calcule un chemin absolu, c'est-à-dire à partir de la racine du filesystem
- `os.path.exists` pour savoir si un chemin existe ou pas (fichier ou répertoire)
- `os.path.isfile` (et `isdir`) pour savoir si un chemin est un fichier (et un répertoire)
- `os.path.getsize` pour obtenir la taille du fichier
- `os.path.getatime` et aussi `getmtime` et `getctime` pour obtenir les dates de création/modification d'un fichier
- `os.remove` (ou son ancien nom `os.unlink`), qui permet de supprimer un fichier
- `os.rmdir` pour supprimer un répertoire (mais qui doit être vide)
- `os.removedirs` pour supprimer tout un répertoire avec son contenu, récursivement si nécessaire
- `os.rename` pour renommer un fichier
- `glob.glob` comme dans par exemple `glob.glob("*.txt")`

Le module **`pathlib`**

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

Orienté Objet

Comme on l'a mentionné `pathlib` offre une interface orientée objet ; mais qu'est-ce que ça veut dire au juste ?

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

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

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

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

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

Pour commencer, nous allons vérifier si le fichier en question existe.

Pour ça nous créons un objet qui est une instance de la classe `Path`, comme ceci :

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

Vous remarquerez que c'est cohérent avec par exemple :

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

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

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

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

et le notebook vous montrera la liste des méthodes disponibles.

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

```
[5]: PosixPath('fichier-temoin')
```

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

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

```
[6]: False
```

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

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

```
[8]: True
```

Métadonnées

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

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

```
[9]: os.stat_result(st_mode=33188, st_ino=12922412997, st_dev=16777220, st_nlink
      =1, st_uid=501, st_gid=20, st_size=11, st_atime=1590418476, st_mtime=159
      0418476, st_ctime=1590418476)
```

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 :

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

```
[10]: 11
```

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

```
[11]: 1590418476.0843477
```

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

```
[12]: datetime.datetime(2020, 5, 25, 16, 54, 36, 84348)
```

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

```
[13]: '16:54'
```

Détruire un fichier

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

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

no need to remove

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

```
[16]: False
```

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

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

Recherche de fichiers

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

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

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

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

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

```
data/cities_europe.json
data/marine-e2-ext.json
data/cities_france.json
data/cities_idf.json
data/marine-e1-abb.json
data/cities_world.json
data/marine-e1-ext.json
data/marine-e2-abb.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`

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

```
[20]: pathlib.PosixPath
```

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 :

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

```
[21]: True
```

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

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

```
[22]: True
```

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

3.3 w3-s1-c3-format-json-et-autres

Formats de fichiers : JSON et autres

3.3.1 Compléments - niveau basique

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

Le problème

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

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

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

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

Le format JSON

Le format sans aucun doute le plus populaire à l'heure actuelle est [le format JSON](#) pour JavaScript Object Notation.

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

La bibliothèque standard de Python contient [le module json](#) que nous illustrons très rapidement ici :

```
[1]: import json

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

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

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

Limitations de json

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

- `tuple`, qui se fait encoder comme une liste ;
- `complex`, `set` et `frozenset`, que l'on ne peut pas encoder du tout (sans étendre la bibliothèque).

C'est ce qui explique ce qui suit :

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

première partie de data True

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

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

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

3.3.2 Compléments - niveau intermédiaire

Le format `csv`

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

Le format pickle

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

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

Le format XML

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

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

Il existe plusieurs souches différentes de bibliothèques prenant en charge le format XML, [qui sont introduites ici](#).

Pour en savoir plus

Voyez la page sur [les formats de fichiers](#) dans la documentation Python.

3.4 w3-s1-c4-fichiers-systeme

Fichiers systèmes

3.4.1 Complément - niveau avancé

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

Introduction

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

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

— `open` ;

```
— close;  
— read;  
— write.
```

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

```
— entrée standard (standard input, en abrégé stdin);  
— sortie standard (standard output, en abrégé stdout);  
— erreur standard (standard error, en abrégé stderr).
```

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

Ainsi par exemple lorsque l'on écrit :

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

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

Le module `sys`

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

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

```
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='UTF-8'>  
<ipykernel.iostream.OutputStream object at 0x10593df60>  
<ipykernel.iostream.OutputStream object at 0x104e27b00>
```

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 :

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

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

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

sur le terminal
de nouveau sur le terminal

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

dans le fichier

3.5 w3-s2-c1-tuple-et-virgule

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 :

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

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


[2]: True

Comme on le voit :

- en réalité la parenthèse est parfois superflue ; mais il se trouve qu'elle est largement utilisée pour améliorer la lisibilité des programmes, sauf dans le cas du tuple unpacking ; nous verrons aussi plus bas qu'elle est parfois nécessaire selon l'endroit où le tuple apparaît dans le programme ;
- la dernière virgule est optionnelle aussi, c'est le cas pour les tuples à au moins 2 éléments - nous verrons plus bas le cas des tuples à un seul élément.

Conseil pour la présentation sur plusieurs lignes

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

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

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

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

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

Tuples à un élément

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

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

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

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

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

```
[5]: int
```

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

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

```
[6]: tuple
```

```
[7]: simple3 == simple4
```

```
[7]: True
```

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

Parenthèse parfois obligatoire

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

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

```
[8]: True
```

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

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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 :

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

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

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

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

apres ajout (1, 2, 3, 4)

Construire des tuples élaborés

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

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

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

```
[11]: [0, 1, 5, 6, 7, 8, 'Inconnu']
```

```
[12]: # on convertit le résultat en tuple
      mon_tuple = tuple(liste)
      mon_tuple
```

```
[12]: (0, 1, 5, 6, 7, 8, 'Inconnu')
```

Digression sur les noms de fonctions prédéfinies

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

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

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

```
[13]: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
[ ]: # 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)

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

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

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

3.6 w3-s2-c2-sequence-unpacking

Sequence unpacking

3.6.1 Complément - niveau basique

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

Déjà rencontré

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

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

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

Un exemple simple

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

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

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

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

```
gauche 100 droite spam
```

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

On préférera la formulation équivalente suivante :

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

```
gauche 100 droite spam
```

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

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

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

```
gauche 100 droite spam
```

Autres types

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

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

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

```
gauche 1 milieu 2 droit 3
```

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

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

```
gauche 1 milieu 2 droit 3
```

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

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

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

La façon pythonique d'échanger deux variables

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

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

a 2 b 1

Extended unpacking

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

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

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

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

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

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

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

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

```
[10]: # si on sait que data contient prenom, nom,
      # et un nombre inconnu d'autres informations
      data = [ 'Jean', 'Dupont', '061234567', '12', 'rue du four', '57000', 'METZ', ]

      # on peut utiliser la variable _
      # ce n'est pas une variable spéciale dans le langage,
      # mais cela indique au lecteur que l'on ne va pas s'en servir
      prenom, nom, *_ = data
      print(f"prenom={prenom} nom={nom}")
```

prenom=Jean nom=Dupont

3.6.2 Complément - niveau intermédiaire

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

Plusieurs occurrences d'une même variable

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

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

```
a = 3
```

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

```
[12]: 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 :

```
[13]: entree = [1, 2, 3]

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

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

```
milieu 2
right 3
```

En profondeur

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

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

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

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

```
trois 3
```

Ou encore, sans doute un peu plus lisible :

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

```
trois 3
```

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

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

```
trois 3
```

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

Extended unpacking et profondeur

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

```
[18]: # un exemple très alambiqué
      tree = [1, 2, [(3, 33, 'three', 'thirty-three')],
              ( [4, 44, ('forty', 'forty-four')])]
      tree
```

```
[18]: [1, 2, [(3, 33, 'three', 'thirty-three')], [4, 44, ('forty', 'forty-four')]]
      ]
```

```
[19]: # unpacking avec plusieurs variables *extended
      _, ((_, *x3, _),), (_, x4) = tree
      print(f"x3={x3}, x4={x4}")
```

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

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

3.6.3 Pour en savoir plus

— [Le PEP \(en anglais\) qui introduit le extended unpacking.](#)

3.7 w3-s2-c3-for-sur-plusieurs-variables

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 :

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



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

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

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

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

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

3.7.2 Complément - niveau intermédiaire

La fonction `zip`

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

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

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

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

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

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

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

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

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

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

Tout ceci se généralise naturellement à plus de deux variables :

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

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

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

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

```
1 10
2 20
```

La fonction `enumerate`

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

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

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

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

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

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

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

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

3.8 w3-s2-x1-comptage

Fichiers

3.8.1 Exercice - niveau basique

Calcul du nombre de lignes, de mots et de caractères

```
[1]: # 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.

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

```
[2]: <IPython.core.display.HTML object>
```

```
[3]: # 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.

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

exo_comptage.correction(comptage)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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

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

Votre fonction ne semble pas créer le fichier de sortie

Accès aux fichiers d'exemples

Vous pouvez télécharger les fichiers d'exemples :

- [Romeo and Juliet](#)
- [Lorem Ipsum](#)
- [“Une charogne” en UTF-8](#)

Pour les courageux, je vous donne également [“Une charogne” en ISO-8859-15](#), qui contient le même texte que “Une charogne”, mais encodé en Latin-9, connu aussi sous le nom ISO-8859-15.

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

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

3.9 w3-s2-x2-surgery

Sequence unpacking

3.9.1 Exercice - niveau basique

```
[1]: # 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.

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

GridBox(children=(HTML(value='appel', _dom_classes=('header

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

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

3.10 w3-s4-c1-dictionnaires

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 :

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

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

Création - la fonction `dict`

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

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

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

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

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

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

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

Accès atomique

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

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

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

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

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

```
valeur pour marc 35
valeur pour inconnu 0
```

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

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

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

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

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

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

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

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

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

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

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

```
False
```

Parcourir toutes les entrées

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

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

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

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

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

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

```
alice  
eric  
bob
```

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

```
30  
39  
42
```

La fonction **len**

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

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

```
3 entrées dans annuaire
```

Pour en savoir plus sur le type **dict**

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

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

3.10.2 Complément - niveau intermédiaire

La méthode **update**

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

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

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

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

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

collections.OrderedDict : dictionnaire et ordre d'insertion

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

```
[16]: %%python2
      # coding: utf-8

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

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

```
a 2
c 3
b 1
```

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 :

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

```
c 3
b 1
a 2
```

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

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

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

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

collections.defaultdict : initialisation automatique

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

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


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

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

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

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

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

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

```
[21]: from collections import defaultdict

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

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

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

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

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

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

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

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

sorted(compteurs.items())
```

```
[22]: [(' ', 8),
      ('a', 5),
      ('c', 3),
```

```
('d', 1),
('e', 8),
('h', 1),
('l', 4),
('m', 1),
('n', 3),
('o', 2),
('p', 2),
('q', 1),
('r', 4),
('s', 4),
('t', 3),
('u', 3),
('v', 1),
('è', 1)]
```

Signalons enfin une fonctionnalité un peu analogue, quoiqu'un peu 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 :

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

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

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

```
[24]: 70
```

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

```
[25]: 50
```

```
[26]: # comme on le voit
annuaire
```

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

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

3.10.3 Complément - niveau avancé

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

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

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

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

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

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

```
a
b
```

Et un test d'appartenance avec `in` :

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

```
True
```

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

```
False
```

Mais ce ne sont pas des listes

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

```
[31]: False
```

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

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

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

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

74.3 ns ± 1.84 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

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

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

15.2 ms ± 182 µs per loop (mean ± std. dev. of 7 runs, 20 loops each)

En fait ce sont des vues. Une autre propriété un peu inattendue de ces objets, c'est que ce sont des vues ; ce qu'on veut dire par là (pour ceux qui connaissent, cela fait 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 :

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

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

a
b

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

a
b
c

Reportez vous à [la section sur les vues de dictionnaires](#) pour plus de détails.

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

3.11 w3-s4-c2-cles-immuables

Clés immuables

3.11.1 Complément - niveau intermédiaire

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

Une clé doit être immuable

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

C'est-à-dire que, pour simplifier, on localise la présence d'une clé en calculant d'abord

$$f(cl) = hash$$

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

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

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

- on range un tuple (*clef*, *valeur*) à un premier indice $f(clef) = hash_1$;
- on modifie la valeur de *clef* qui devient *clef'* ;
- on recherche notre valeur à l'indice $f(clef') = hash_2 \neq hash_1$.

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

Une clé doit être globalement immuable

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

Type	Mutable ?
<code>int, float</code>	immuable
<code>complex, bool</code>	immuable
<code>str</code>	immuable
<code>list</code>	mutable
<code>dict</code>	mutable
<code>set</code>	mutable
<code>frozenset</code>	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 :

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

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

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

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

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

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

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

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

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

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

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

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

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

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

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

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

Épilogue

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

3.12 w3-s4-c3-record-et-dictionnaire

Gérer des enregistrements

3.12.1 Complément - niveau intermédiaire

Implémenter un enregistrement comme un dictionnaire

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

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

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

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

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

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

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

Ce qui nous donne :

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

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

Un dictionnaire pour indexer les enregistrements

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

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

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

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

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

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

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

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 :

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

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

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

Techniques similaires

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

3.12.2 Complément - niveau avancé

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

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

```
[7]: class Personne:

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

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

Pour construire ma liste de personnes, je fais alors :

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

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


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

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

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

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

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

Et de cette façon :

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

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

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

3.13 w3-s4-x1-graph-dict

Dictionnaires et listes

3.13.1 Exercice - niveau basique

```
[1]: 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 :

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

```
s1 10 s2
s2 12 s3
s3 25 s1
s1 14 s3
```

qui signifierait :

- un graphe à 3 sommets s1, s2 et s3;
- et 4 arêtes
 - une entre s1 et s2 de longueur 10;
 - une entre s2 et s3 de longueur 12;
 - etc...

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

Dans cet exercice on choisit :

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

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

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

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

Notes

- Vous remarquerez que l'exemple ci-dessus retourne un dictionnaire standard ; une solution qui utiliserait `defaultdict` est acceptable également ;
- Notez bien également que dans le résultat, la longueur d'un arc est attendue comme un `int`.

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

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

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

3.14 w3-s4-x2-marine-dict

Fusionner des données

3.14.1 Exercices

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

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

Contexte

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

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

- mode étendu : chaque mesure (bateau x position x temps) est accompagnée de tous les détails du bateau (`id`, `nom`, `pays de rattachement`, etc.) ;
- mode abrégé : chaque mesure est uniquement attachée à l'`id` du bateau.

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

Chargement des données

Commençons par charger les données de l'exercice :

```
[1]: 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.

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

```
[255801560, 49.3815, -4.412167, '2013-10-08T21:51:00', 'AUTOPRIDE', 'PT', '
', 'ZEEBRUGGE']
```

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

```
[227254910, 49.91799, -5.315172, '2013-10-08T22:59:00']
```

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

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

But de l'exercice

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

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

De manière plus imagée, si :

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

Et si :

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

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

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

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

```
[5]: # 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
```

```
==== clé
992271012
==== valeur
[992271012, 47.64744, -3.509282, '2013-10-08T21:50:00', 'PENMEN', 'FR', '',
  '']
```

Remarquez ci-dessus l'utilisation d'un utilitaire parfois pratique : le [module pprint](#) pour pretty-printer.

Votre code

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

Validation

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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

```
[ ]: exo_index.correction(index, extended)  
  
# NOTE  
# auto-exec-for-latex has skipped execution of this cell
```

Exercice - niveau intermédiaire

```
[7]: # 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 :

```
[8]: # le résultat attendu  
result_merge = exo_merge.resultat(extended, abbreviated)  
  
# à quoi ressemble le résultat pour un bateau au hasard  
from pprint import pprint  
for key_value in result_merge.items():  
    pprint(key_value)  
    break
```

```
(992271012,  
 ['PENMEN',  
  'FR',  
  (47.64744, -3.509282, '2013-10-08T21:50:00'),  
  (47.64748, -3.509307, '2013-10-08T22:56:00')])
```

Votre code

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

Validation

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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 :

```
[10]: # 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 :

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

3.15 w3-s5-c1-ensembles

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 :

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

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

Création - la fonction `set`

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

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

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

Créer un ensemble vide

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

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

```
[3]: dict
```

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

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

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

```
<class 'set'>
```

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

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

```
<class 'set'>
```

Un élément dans un ensemble doit être globalement immuable

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

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

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

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

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

Création - la fonction `frozenset`

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

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

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

Opérations simples

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

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

Test d'appartenance

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

```
[7]: True
```

Cardinal

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

```
[8]: 4
```


Manipulations

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

```
[9]: {1, 2}
```

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

```
[10]: set()
```

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

```
[11]: {1}
```

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

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

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

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

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

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

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

```
[15]: {1, 2}
```

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

```
remove a levé l'exception 'foo'
```

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

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

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

Opérations classiques sur les ensembles

Donnons-nous deux ensembles simples :

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

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

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

Remarques :

- les notations des opérateurs sur les ensembles rappellent les opérateurs “bit-à-bit” sur les entiers ;
- ces opérateurs sont également disponibles sous la forme de méthodes.

Union

```
[19]: A2 | A3
```

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

Intersection

```
[20]: A2 & A3
```

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

Différence

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

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

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

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

Différence symétrique

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

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

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

Comparaisons

Ici encore on se donne deux ensembles :

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

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

Égalité

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

```
[25]: True
```

Inclusion

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

```
[26]: True
```

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

```
[27]: True
```

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

```
[28]: False
```

Ensembles disjoints

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

```
[29]: True
```

Pour en savoir plus

Reportez vous à [la section sur les ensembles](#) dans la documentation Python.

3.16 w3-s5-x1-read-set

Ensembles

3.16.1 Exercice - niveau basique

```
[1]: # 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 :

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

```
4615
12
9228
6158
12
```

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

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

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

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

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

3.16.2 Deuxième partie - niveau basique

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

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

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

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

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

Par exemple :

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

```
4615
12
9228
6158
12
```

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

```
2048
8192
9228
2049
3
4
2053
2054
6158
4099
8
12
```

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

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

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

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

3.17 w3-s5-x2-marine-set

Exercice sur les ensembles

3.17.1 Exercice - niveau intermédiaire

```
[1]: # 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.

```
[2]: 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]

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

```
[246497000, 49.38633, -4.0195, '2013-10-08T21:51:00', 'ELISABETH', 'NL', ''
, 'ALGER']
```

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

```
[228109000, 48.18373, -5.049758, '2013-10-08T22:58:00']
```

But de l'exercice

```
[5]: # 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).

```
[6]: # 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)
```

```
----- Les entrées
Dans extended: 4 entrées
Dans abbreviated: 4 entrées
----- Le résultat du diff
Dans extended mais pas dans abbreviated 2
Dans les deux 2
Dans abbreviated mais pas dans extended 2
```

Votre code

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

      # NOTE:
      # auto-exec-for-latex has used hidden code instead
```

Validation

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

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

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 :

```
[8]: # 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)

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

```
----- Les entrées
Dans extended: 205 entrées
Dans abbreviated: 200 entrées
----- Le résultat du diff
Dans extended mais pas dans abbreviated 70
Dans les deux 132
Dans abbreviated mais pas dans extended 68
```

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

- on a supprimé les entrées correspondant à des bateaux différents mais de même nom ; cette situation peut arriver dans la réalité (c'est pourquoi d'ailleurs les bateaux ont un id) mais ici ce n'est pas le cas ;
- il se peut par contre qu'un même bateau fasse l'objet de plusieurs mesures dans `extended` et/ou dans `abbreviated`.

3.18 w3-s6-c1-try-finally

try ... else ... finally

3.18.1 Complément - niveau intermédiaire

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

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

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

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

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

on passe ici même si on a vu un return

```
[2]: 1.0
```

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

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

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

```
else
```

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

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

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

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

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

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

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

on passe ici seulement avec un nombre non nul

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

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

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

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

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

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

```
else:
    print("on ne passe jamais ici à cause des return")
```

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

```
[8]: 1.0
```

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

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

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

Pour en savoir plus

Voyez [le tutorial sur les exceptions](#) dans la documentation officielle.

3.19 w3-s7-c1-operateur-is-et-fonction-id L'opérateur **is**

3.19.1 Complément - niveau basique

```
[1]: %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

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

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

```
== True
```

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

is False

Scénario 2

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

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

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

== True

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

is True

La même chose sous pythontutor

Scénario 1

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Scénario 2

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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 préférera écrire :

```
[6]: undef = None

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

indéfini

Plutôt que :

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

indéfini

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

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

3.19.2 Complément - niveau intermédiaire

La fonction `id`

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

```
[8]: id(True)
```

```
[8]: 4516232672
```

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

$$(a \text{ is } b) \iff (id(a) == id(b))$$

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 :

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

a 4516636128 b 4516636128

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 :

```
[10]: a is b
```

```
[10]: True
```

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

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

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

```
[11]: True
```

Ou encore, plus surprenant :

```
[12]: a = "foo"  
      b = "foo"  
      a is b
```

```
[12]: True
```

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

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

Pour en savoir plus

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

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

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

3.20 w3-s7-c2-references-circulaires

Listes infinies & références circulaires

3.20.1 Complément - niveau intermédiaire

```
[1]: %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 :

```
[2]: infini_1 = [None]
```

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

```
[3]: 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 :

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

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

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

```
[4]: infini_1 == infini_1
```

```
[4]: True
```

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

```
[5]: infini_2 = [None]
      infini_2[0] = infini_2
```

```
print(infini_2)
```

```
[...]
```

```
[ ]: # attention, ceci provoque une erreur à l'exécution
# RecursionError: maximum recursion depth exceeded in comparison
infini_1 == infini_2

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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 :

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

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

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

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

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

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

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

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

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

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```


3.21 w3-s7-c3-les-differentes-copies

Les différentes copies

```
[1]: %load_ext ipythontutor
```

3.21.1 Complément - niveau basique

Deux types de copie

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

- La shallow copy - de l'anglais shallow qui signifie superficiel ;
- La deep copy - de deep qui signifie profond.

Le module **copy**

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

- `copy.copy` pour une copie superficielle ;
- `copy.deepcopy` pour une copie en profondeur.

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

Un exemple

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

La copie superficielle / shallow copie / **copy.copy** N'oubliez pas de cliquer le bouton **Forward** dans la fenêtre pythontutor :

```
[ ]: %%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)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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 :

```
[ ]: %%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)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Ici, il faut remarquer que :

- les deux objets mutables accessibles via `source`, c'est-à-dire la liste `source[0]` et l'ensemble `source[1]`, ont été tous deux dupliqués ;
- le tuple correspondant à `source[2]` n'est pas dupliqué, mais comme il n'est pas mutable on ne peut pas modifier la copie au travers de la source ;
- de manière générale, on a la bonne propriété que la source et sa copie ne partagent rien qui soit modifiable ;
- et donc on ne peut pas modifier l'un au travers de l'autre.

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

3.21.2 Complément - niveau intermédiaire

```
[3]: # 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 == :

```

[4]: print('source == shallow_copy:', source == shallow_copy)
     print('source == deep_copy:', source == deep_copy)

```

```

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

```

Inspectons les objets de premier niveau

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

```

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

```

```

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

```

```

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

```

```

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

```

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

```

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

```

```

source[0] is deep_copy[0] -> False
source[1] is deep_copy[1] -> False
source[2] is deep_copy[2] -> True
source[3] is deep_copy[3] -> True

```

```
source[4] is deep_copy[4] -> True
```

On retrouve ici ce qu'on avait déjà remarqué sous pythontutor, à savoir que les trois derniers objets - immuables - n'ont pas été dupliqués comme on aurait pu s'y attendre.

On modifie la source

Il doit être clair à présent que, précisément parce que `deep_copy` est une copie en profondeur, on peut modifier `source` sans impacter du tout `deep_copy`.

S'agissant de `shallow_copy`, par contre, seuls les éléments de premier niveau ont été copiés. Aussi si on fait une modification par exemple à l'intérieur de la liste qui est le premier fils de `source`, cela sera répercuté dans `shallow_copy` :

```
[8]: print("avant, source      ", source)
      print("avant, shallow_copy", shallow_copy)
      source[0].append(4)
      print("après, source      ", source)
      print("après, shallow_copy", shallow_copy)
```

```
avant, source      [[1, 2, 3], {1, 2, 3}, (1, 2, 3), '123', 123]
avant, shallow_copy [[1, 2, 3], {1, 2, 3}, (1, 2, 3), '123', 123]
après, source      [[1, 2, 3, 4], {1, 2, 3}, (1, 2, 3), '123', 123]
après, shallow_copy [[1, 2, 3, 4], {1, 2, 3}, (1, 2, 3), '123', 123]
```

Si par contre on remplace complètement un élément de premier niveau dans la source, cela ne sera pas répercuté dans la copie superficielle :

```
[9]: print("avant, source      ", source)
      print("avant, shallow_copy", shallow_copy)
      source[0] = 'remplacement'
      print("après, source      ", source)
      print("après, shallow_copy", shallow_copy)
```

```
avant, source      [[1, 2, 3, 4], {1, 2, 3}, (1, 2, 3), '123', 123]
avant, shallow_copy [[1, 2, 3, 4], {1, 2, 3}, (1, 2, 3), '123', 123]
après, source      ['remplacement', {1, 2, 3}, (1, 2, 3), '123', 123]
après, shallow_copy [[1, 2, 3, 4], {1, 2, 3}, (1, 2, 3), '123', 123]
```

Copie et circularité

Le module `copy` est capable de copier - même en profondeur - des objets contenant des références circulaires.

```
[10]: l = [None]
      l[0] = l
      l
```

```
[10]: [[...]]
```

```
[11]: copy.copy(l)
```

```
[11]: [[[...]]]
```

```
[12]: copy.deepcopy(1)
```

```
[12]: [...]
```

Pour en savoir plus

On peut se reporter à [la section sur le module copy](#) dans la documentation Python.

3.22 w3-s7-c4-instruction-del

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.

```
[1]: # la variable a n'est pas définie
try:
    print('a=', a)
except NameError as e:
    print("a n'est pas définie")
```

a n'est pas définie

```
[2]: # on la définit
a = 10

# aucun souci ici, l'exception n'est pas levée
try:
    print('a=', a)
except NameError as e:
    print("a n'est pas définie")
```

a= 10

```
[3]: # maintenant on peut effacer la variable
del a

# c'est comme si on ne l'avait pas définie
# dans la cellule précédente
try:
    print('a=', a)
except NameError as e:
```

```
print("a n'est pas définie")
```

a n'est pas définie

Sur une liste

On peut enlever d'une liste les éléments qui correspondent à une slice :

```
[4]: # on se donne une liste
1 = list(range(12))
print(1)
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

```
[5]: # on considère une slice dans cette liste
print('slice=', 1[2:10:3])

# voyons ce que ça donne si on efface cette slice
del 1[2:10:3]
print("après del", 1)
```

slice= [2, 5, 8]

après del [0, 1, 3, 4, 6, 7, 9, 10, 11]

Sur un dictionnaire

Avec del on peut enlever une clé, et donc la valeur correspondante, d'un dictionnaire :

```
[6]: # partons d'un dictionnaire simple
d = dict(foo='bar', spam='eggs', a='b')
print(d)
```

{'foo': 'bar', 'spam': 'eggs', 'a': 'b'}

```
[7]: # on peut enlever une clé avec del
del d['a']
print(d)
```

{'foo': 'bar', 'spam': 'eggs'}

On peut passer plusieurs arguments à del

```
[8]: # Voyons où en sont nos données
print('l', l)
print('d', d)
```

l [0, 1, 3, 4, 6, 7, 9, 10, 11]
d {'foo': 'bar', 'spam': 'eggs'}

```
[9]: # on peut invoquer 'del' avec plusieurs expressions
      # séparées par une virgule
      del l[3:], d['spam']

      print('l', l)
      print('d', d)
```

```
l [0, 1, 3]
d {'foo': 'bar'}
```

Pour en savoir plus

La page sur [l'instruction del](#) dans la documentation Python.

3.23

w3-s7-c5-affectation-simultanee

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 :

```
[1]: x, y = ['spam', 'egg']
```

Dans ce complément nous allons voir une autre forme de l'affectation, qui consiste à affecter le même objet à plusieurs variables. Commençons par un exemple simple :

```
[2]: a = b = 1
      print('a', a, 'b', b)
```

```
a 1 b 1
```

La raison pour laquelle nous abordons cette construction maintenant est qu'elle a une forte relation avec les références partagées ; pour bien le voir, nous allons utiliser une valeur mutable comme valeur à affecter :

```
[3]: # on affecte a et b au même objet liste vide
      a = b = []
```

Dès lors nous sommes dans le cas typique d’une référence partagée ; une modification de **a** va se répercuter sur **b** puisque ces deux variables désignent le même objet :

```
[4]: a.append(1)
      print('a', a, 'b', b)
```

```
a [1] b [1]
```

Ceci est à mettre en contraste avec plusieurs affectations séparées :

```
[5]: # si on utilise deux affectations différentes
      a = []
```

```
b = []

# alors on peut changer a sans changer b
a.append(1)
print('a', a, 'b', b)
```

```
a [1] b []
```

On voit que dans ce cas chaque affectation crée une liste vide différente, et les deux variables ne partagent plus de donnée.

D’une manière générale, utiliser l’affectation simultanée vers un objet mutable crée mécaniquement des références partagées, aussi vérifiez bien dans ce cas que c’est votre intention.

3.24 w3-s7-c6-affectation-operateurs-2

Les instructions += et autres revisitées

3.24.1 Complément - niveau intermédiaire

Nous avons vu en première semaine (Séquence “Les types numériques”) une première introduction à l’instruction += et ses dérivées comme *=, **=, etc.

Ces constructions ont une définition à géométrie variable

En C quand on utilise += (ou encore ++) on modifie la mémoire en place - historiquement, cet opérateur permettait au programmeur d’aider à l’optimisation du code pour utiliser les instructions assembleur idoines.

Ces constructions en Python s’inspirent clairement de C, aussi dans l’esprit ces constructions devraient fonctionner en modifiant l’objet référencé par la variable.

Mais les types numériques en Python ne sont pas mutables, alors que les listes le sont. Du coup le comportement de += est différent selon qu’on l’utilise sur un nombre ou sur une liste, ou plus généralement selon qu’on l’invoque sur un type mutable ou non. Voyons cela sur des exemples très simples :

```
[1]: # Premier exemple avec un entier

# on commence avec une référence partagée
a = b = 3
a is b
```

```
[1]: True
```

```
[2]: # on utilise += sur une des deux variables
a += 1

# ceci n'a pas modifié b
# c'est normal, l'entier n'est pas mutable

print(a)
print(b)
print(a is b)
```



```
4
3
False
```

```
[3]: # Deuxième exemple, cette fois avec une liste

# la même référence partagée
a = b = []
a is b
```

```
[3]: True
```

```
[4]: # pareil, on fait += sur une des variables
a += [1]

# cette fois on a modifié a et b
# car += a pu modifier la liste en place
print(a)
print(b)
print(a is b)
```

```
[1]
[1]
True
```

Vous voyez donc que la sémantique de += (c'est bien entendu le cas pour toutes les autres formes d'instructions qui combinent l'affectation avec un opérateur) est différente suivant que l'objet référencé par le terme de gauche est mutable ou immuable.

Pour cette raison, c'est là une opinion personnelle, cette famille d'instructions n'est pas le trait le plus réussi dans le langage, et je ne recommande pas de l'utiliser.

Précision sur la définition de +=

Nous avons 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 :

```
[5]: # si on fait x += y sur une liste
# on fait un effet de bord sur la liste
# comme on vient de le voir

a = []
print("avant", id(a))
a += [1]
print("après", id(a))
```

avant 4433311432
 après 4433311432

```
[6]: # alors que si on fait x = x + y sur une liste
      # on crée un nouvel objet liste

      a = []
      print("avant", id(a))
      a = a + [1]
      print("après", id(a))
```

avant 4433112648
 après 4433110856

Vous voyez donc que vis-à-vis des références partagées, ces deux façons de faire mènent à un résultat différent.

3.25 w3-s8-x1-fifo

Classe

3.25.1 Exercice - niveau basique

```
[1]: # 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.

```
[2]: # voici un exemple de scénario
      exo_fifo.example()
```

GridBox(children=(HTML(value='scénario 1', _dom_classes=('h

```
[3]: # vous pouvez définir votre classe ici
      class Fifo:
          def __init__(self):
              "votre code"
          def incoming(self, value):
              "votre code"
          def outgoing(self):
              "votre code"
```

```
[ ]: # et la vérifier ici  
exo_fifo.correction(Fifo)  
  
# NOTE  
# auto-exec-for-latex has skipped execution of this cell
```


Chapitre 4

Fonctions et portée des variables

4.1 w4-s1-cl-passage-par-reference

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

```
[1]: def ma_fonction(dans_fonction):  
      print(dans_fonction)  
  
      dans_appelant = ["texte"]  
      ma_fonction(dans_appelant)
```

['texte']

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.

```
[2]: # chargeons la magie pour pythontutor
%load_ext ipythontutor
```

```
[ ]: %%ipythontutor curInstr=4
def ma_fonction(dans_fonction):
    print(dans_fonction)

dans_appelant = ["texte"]
ma_fonction(dans_appelant)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Ce qui signifie qu'on peut voir le code ci-dessus comme étant - pour simplifier - équivalent à ceci :

```
[3]: dans_appelant = ["texte"]

# ma_fonction (dans_appelant)
# → on entre dans la fonction
dans_fonction = dans_appelant
print(dans_fonction)
```

```
['texte']
```

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 :

```
[4]: 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)
```

```
dans appelant    ['texte'] 4590833288
dans ma_fonction ['texte'] 4590833288
```

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 :

```
[5]: # on ne peut pas modifier un immuable dans une fonction
def increment(n):
    n += 1

compteur = 10
increment(compteur)
print(compteur)
```

```
10
```

```
[6]: # on peut par contre ajouter dans une liste
def insert(liste, valeur):
```

```
liste.append(valeur)

liste = ["un"]
insert(liste, "texte")
print(liste)
```

```
['un', 'texte']
```

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 avons 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 w4-s1-c2-docstring

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.

```
[1]: 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 :

```
[2]: help(flatten)
```

Help on function flatten in module __main__:

```
flatten(containers)
    returns a list of the elements of the elements in containers
```

Soit programmativement :

```
[3]: flatten.__doc__
```

```
[3]: 'returns a list of the elements of the elements in containers'
```

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 :

```
[4]: # 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)
```

Help on function flatten in module `__main__`:

```
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
```

```
[5]: # 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)
```

Help on function flatten in module `__main__`:

```
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
```

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 w4-s1-c3-isinstance

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 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 commodité 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) :

```
[1]: 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)
```

```
[2]: print("entier", factoriel(4))
      print("chaîne", factoriel("8"))
      print("tuple", factoriel((4, 8)))
```

```
entier 24
chaîne 40320
tuple [24, 40320]
```

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 :

```
[3]: from types import FunctionType
     isinstance(factoriel, FunctionType)
```

[3]: True

Mais méfiez-vous toutefois des fonctions built-in, qui sont de type `BuiltinFunctionType`

```
[4]: from types import BuiltinFunctionType
     isinstance(len, BuiltinFunctionType)
```

[4]: True

```
[5]: # alors qu'on pourrait penser que
     isinstance(len, FunctionType)
```

[5]: False

`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.

```
[6]: 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 :

```
[7]: # pour 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 mammifère ?", isinstance(baleine, Mammifere))
```

l'objet baleine est-il un mammifère ? True

```
[8]: # ici c'est moins évident, mais la réponse est 'True' aussi
print("l'objet baleine est-il un animal ?", isinstance(baleine, Animal))
```

l'objet baleine est-il un animal ? True

Vous voyez qu'ici, bien que l'objet baleine soit 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 matière 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](#).

```
[9]: # voici par ailleurs la liste de ses attributs
import types
dir(types)
```

```
[9]: ['AsyncGeneratorType',
      'BuiltinFunctionType',
      'BuiltinMethodType',
      'ClassMethodDescriptorType',
      'CodeType',
      'CoroutineType',
      'DynamicClassAttribute',
      'FrameType',
      'FunctionType',
      'GeneratorType',
      'GetSetDescriptorType',
      'LambdaType',
      'MappingProxyType',
      'MemberDescriptorType',
      'MethodDescriptorType',
      'MethodType',
      'MethodWrapperType',
```

```
'ModuleType',
'SimpleNamespace',
'TracebackType',
'WrapperDescriptorType',
'_GeneratorWrapper',
'__all__',
'__builtins__',
'__cached__',
'__doc__',
'__file__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'_calculate_meta',
'coroutine',
'new_class',
'prepare_class',
'resolve_bases']
```

4.4 w4-s1-c4-type-hints

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

```
[1]: # pour typer une variable avec les type hints
nb_items : int = 0
```

```
[2]: nb_items
```

```
[2]: 0
```

typer les paramètres et le retour d'une fonction

```
[3]: # une fonction factorielle avec des type hints
def fact(n : int) -> int:
    return 1 if n <= 1 else n * fact(n-1)
```

```
[4]: fact(12)
```

```
[4]: 479001600
```

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.

```
[5]: # 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)
```

```
[5]: 479001600
```

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 ambiguïtés.

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

```
[6]: from typing import List

[7]: # une fonction qui
      # attend un paramètre qui soit une liste d'entiers,
      # et qui retourne une liste de chaînes
      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 built-in `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 :

```
[8]: from typing import Iterable

[9]: def lower_split(sep: str, inputs : Iterable[str]) -> str:
      return sep.join([x.lower() for x in inputs])
```

```
[10]: lower_split('--', ('AB', 'CD', 'EF'))
```

```
[10]: '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 :

```
[11]: 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 à voir, mais...) pour vous signaler un objet python assez étrange :

```
[12]: # L'objet ... existe bel et bien en Python
el = ...
el
```

```
[12]: Ellipsis
```

qui sert principalement pour le slicing multidimensionnel 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 :

```
[13]: # 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
```

```
[14]: # ou cela
from typing import Any

def partially_typed(n1: int, n2: Any) -> Any:
```

```
return None
```

alias

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 :

```
[15]: from typing import NewType

UserId = NewType('UserId', int)

user1_id : UserId = 0
```

plutôt que ceci, qui est beaucoup moins parlant :

```
[16]: 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` :

```
[17]: 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 fonctionnalité, 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

w4-s2-c1-conditions-1

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 :

```
[1]: # ici la condition s'évalue à 0, donc on ne fait rien  
if 3 - 3:  
    print("ne passera pas par là")
```

```
[2]: # 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")
```

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.

```
[3]: if "":  
    print("ne passera pas par là")  
if []:  
    print("ne passera pas par là")  
if ():  
    print("ne passera pas par là")
```

```
[4]: # 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 :

```
[5]: bas = 12
    haut = 25.82

    # égalité ?
    if bas == haut:
        print('==')
```

```
[6]: # non égalité ?
    if bas != haut:
        print('!=')
```

`!=`

En général, deux objets de types différents ne peuvent pas être égaux.

```
[7]: # 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 :

```
[8]: bas_reel = 12.
```

```
[9]: print(bas, bas_reel)
```

12 12.0

```
[10]: # le réel 12 et
    # l'entier 12 sont égaux
    if bas == bas_reel:
        print('int == float')
```

`int == float`

```
[11]: # ditto pour int et complex
    if (12 + 0j) == 12:
        print('int == complex')
```

`int == complex`

Signalons à titre un peu anecdotique une syntaxe ancienne : historiquement et seulement en Python 2 on pouvait aussi noter `<>` le test de non égalité. On trouve ceci dans du code ancien mais il faut éviter de l'utiliser :

```
[12]: %%python2
      # coding: utf-8

      # l'ancienne forme de !=
      if 12 <> 25:
          print("<> est obsolete et ne fonctionne qu'en python2")
```

`<>` est obsolete et ne fonctionne qu'en python2

Les opérateurs de comparaison

Sans grande surprise on peut aussi écrire

```
[13]: if bas <= haut:
      print('<=')
      if bas < haut:
          print('<')
```

`<=`
`<`

```
[14]: 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

```
[15]: x = (bas + haut) / 2
      print(x)
```

18.91

```
[16]: # deux tests en une expression
      if bas <= x <= haut:
          print("dans l'intervalle")
```

dans l'intervalle

On peut utiliser les comparaisons sur une palette assez large de types, comme par exemple avec les listes

```
[17]: # on peut comparer deux listes, mais ATTENTION
      [1, 2] <= [2, 3]
```

[17]: True

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 :

```
[18]: # on ne peut pas par contre comparer deux nombres complexes
try:
    2j <= 3j
except Exception as e:
    print("OOPS", type(e), e)
```

```
OOPS <class 'TypeError'> '<=' not supported between instances of 'complex'
and 'complex'
```

Connecteurs logiques et / ou / non

On peut bien sûr combiner facilement plusieurs expressions entre elles, grâce aux opérateurs **and**, **or** et **not**

```
[19]: # 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")
```

OK mais pourrait être mieux

En matière 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 préférera de beaucoup la formulation équivalente :

```
[20]: # 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")
```

OK, c'est équivalent et plus clair

```
[21]: # 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 w4-s2-c2-evaluation-conditions

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 :

```
[1]: 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')
```

avec b

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 savants, 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.

```
[2]: # on se donne une liste
liste = ['premier', 'deuxieme', 'troisieme']
print(f"liste={liste}")
```

liste=['premier', 'deuxieme', 'troisieme']

```
[3]: # 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}")
```

après pop(0), element=premier et liste=['deuxieme', 'troisieme']

```
[4]: # et ainsi de suite
element = liste.pop(0)
print(f"après pop(0), element={element} et liste={liste}")
```

après pop(0), element=deuxieme et liste=['troisieme']

Conditions avec effet de bord

Une fois ce rappel fait, voyons maintenant l'exemple suivant :

```
[5]: liste = list(range(5))
print('liste en entree:', liste, 'de taille', len(liste))
```

liste en entree: [0, 1, 2, 3, 4] de taille 5

```
[6]: 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))
```

cas 1

liste en sortie de taille 4

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 :

```
[7]: liste = list(range(5, 10))
print('en entree: liste=', liste, 'de taille', len(liste))
```

en entree: liste= [5, 6, 7, 8, 9] de taille 5

```
[8]: 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))
```

```
cas 4
en sortie: liste= [8, 9] de taille 2
```

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 :

```
[9]: def true():
      print('true')
      return True
```

```
[10]: def false():
       print('false')
       return False
```

```
[11]: true()
```

```
true
```

```
[11]: 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 :

```
[12]: false() and true()
```

```
false
```

```
[12]: False
```

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` :

```
[13]: true() or false()
```

```
true
```

```
[13]: True
```

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 :

```
[14]: true() and (false() or true()) or (true () and false())
```

```
true  
false  
true
```

```
[14]: True
```

4.7 w4-s2-c3-if-comme-expression

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 :


```
[1]: x = True # ou quoi que ce soit d'autre
      if x:
          y = 12
      else:
          y = 35
      print(y)
```

12

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 :

```
<résultat_si_vrai> if <condition> else <résultat_si_faux>
```

Ainsi on pourrait écrire l'exemple ci-dessus de manière plus simple et plus concise comme ceci :

```
[2]: y = 12 if x else 35
      print(y)
```

12

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 :

```
[3]: # 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)
```

0

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 ambiguïté.

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 w4-s2-c4-conditions-2

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 :

```
[1]: # 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")
```

```
18 divisible par trois
19 non 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 ambiguïté - comment se comportera `if` avec cette entrée.

```
[2]: def show_bool(x):
      print(f"condition {repr(x):>10} considérée comme {bool(x)}")
```

```
[3]: for exp in [None, "", 'a', [], [1], (), (1, 2), {}, {'a': 1}, set(), {1}]:
      show_bool(exp)
```

```
condition      None considérée comme False
condition      '' considérée comme False
condition      'a' considérée comme True
condition      [] considérée comme False
condition      [1] considérée comme True
condition      () considérée comme False
condition      (1, 2) considérée comme True
condition      {} considérée comme False
condition      {'a': 1} considérée comme True
condition      set() considérée comme False
condition      {1} considérée comme True
```

Quelques exemples d’expressions

Référence à une variable et dérivés

```
[4]: a = list(range(4))
      print(a)
```

```
[0, 1, 2, 3]
```

```
[5]: 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")
```

```
a n'est pas vide
a[1] n'est pas nul
```

Appels de fonction ou de méthode

```
[6]: chaine = "jean"
      if chaine.upper():
          print("la chaine mise en majuscule n'est pas vide")
```

la chaine mise en majuscule n'est pas vide

```
[7]: # 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")
```

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 :

```
[8]: 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")
```

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	<code>==, !=, is, is not</code>
Appartenance	<code>in</code>
Comparaison	<code><=, <, >, >=</code>
Logiques	<code>and, or, not</code>

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 :

```
[9]: [] == ()
```

```
[9]: False
```

```
[10]: [1, 2] == (1, 2)
```

```
[10]: False
```

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 :

```
[11]: 2 and [1, 2]
```

```
[11]: [1, 2]
```

```
[12]: None or "abcde"
```

```
[12]: '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 :

```
[13]: 1 and 2 and 3
```

```
[13]: 3
```

```
[14]: 1 and 2 and 3 and '' and 4
```

```
[14]: ''
```

```
[15]: [] or "" or {}
```

```
[15]: {}
```

```
[16]: [] or "" or {} or 4 or set()
```

```
[16]: 4
```

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.

```
[17]: # 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")
```

une construction illisible

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 w4-s2-x1-dispatch

L'instruction `if`

4.9.1 Exercice - niveau basique

Répartiteur (1)

```
[1]: # on charge l'exercice
from corrections.exo_dispatch import exo_dispatch1
```

On vous demande d'écrire une fonction `dispatch1`, qui prend en argument deux entiers `a` et `b`, et qui renvoie selon les cas :

	<i>a pair</i>	<i>a impair</i>
<i>b pair</i>	$a^2 + b^2$	$(a - 1) * b$
<i>b impair</i>	$a * (b - 1)$	$a^2 - b^2$

```
[2]: # un petit exemple
exo_dispatch1.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[3]: def dispatch1(a, b):
      "<votre_code>"
```

```
[ ]: # pour vérifier votre code
exo_dispatch1.correction(dispatch1)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

4.9.2 Exercice - niveau basique

Répartiteur (2)

```
[4]: # chargement de l'exercice
from corrections.exo_dispatch import exo_dispatch2
```

Dans une seconde version de cet exercice, on vous demande d'écrire une fonction `dispatch2` qui prend en arguments :

- `a` et `b` deux entiers
- `A` et `B` deux ensembles (chacun pouvant être matérialisé par un ensemble, une liste ou un tuple)

et qui renvoie selon les cas :

	$a \in A$	$a \notin A$
$b \in B$	$a^2 + b^2$	$(a - 1) * b$
$b \notin B$	$a * (b - 1)$	$a^2 + b^2$

```
[5]: def dispatch2(a, b, A, B):
      "<votre_code>"
```

```
[ ]: # pour vérifier votre code
exo_dispatch2.correction(dispatch2)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

4.10 w4-s2-x2-libelle

Expression conditionnelle

4.10.1 Exercice - niveau basique

Analyse et mise en forme

```
[1]: # 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 présentes dans la ligne ne sont pas significatives et doivent être ignorées.

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 ignorées, 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

```
[2]: # voici quelques exemples de ce qui est attendu
      exo_libelle.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[3]: # écrivez votre code ici
      def libelle(ligne):
          "<votre_code>"
```

```
[ ]: # pour le vérifier
      exo_libelle.correction(libelle)

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

4.11 w4-s3-c2-boucle-while

La boucle **while ... else**

4.11.1 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'interpréteur 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.

4.11.2 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 :

```
[1]: # une autre façon de parcourir une liste
liste = ['a', 'b', 'c']

while liste:
    element = liste.pop()
    print(element)
```

c
b
a

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**

```
[2]: # 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...')
```

```
[3]: # sortie de la boucle sans break
# on passe par else
scan(['a'], False)
```

```
sans break
a
else...
```

```
[4]: # on sort de la boucle par le break
scan(['a'], True)
```

```
avec break
a
```

Ce trait est toutefois très rarement utilisé.

4.12 w4-s3-x1-pgcd

Calculer le PGCD

4.12.1 Exercice - niveau basique

```
[1]: # 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 :

```
[2]: exo_pgcd.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

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 $a \geq b$)

```
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`.

```
[3]: # à vous de jouer
def pgcd(a, b):
    "<votre code>"
```

```
[ ]: # pour vérifier votre code
exo_pgcd.correction(pgcd)
```

```
# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

4.13 w4-s3-x2-taxes

Exercice

4.13.1 Niveau basique

```
[1]: 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 barème est [publié ici par le gouvernement anglais](#), voici les données de 2018 qui sont utilisées pour l'exercice :

Tranche	Revenu imposable	Taux
Non imposable	jusque £12.500	0%
Taux de base	£12.501 à £50.000	20%
Taux élevé	£50.001 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.

```
[2]: exo_taxes.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

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.

```
[3]: def taxes(income):
    # ce n'est pas la bonne réponse
    return (income-11_500) * (20/100)
```

```
[ ]: exo_taxes.correction(taxes)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

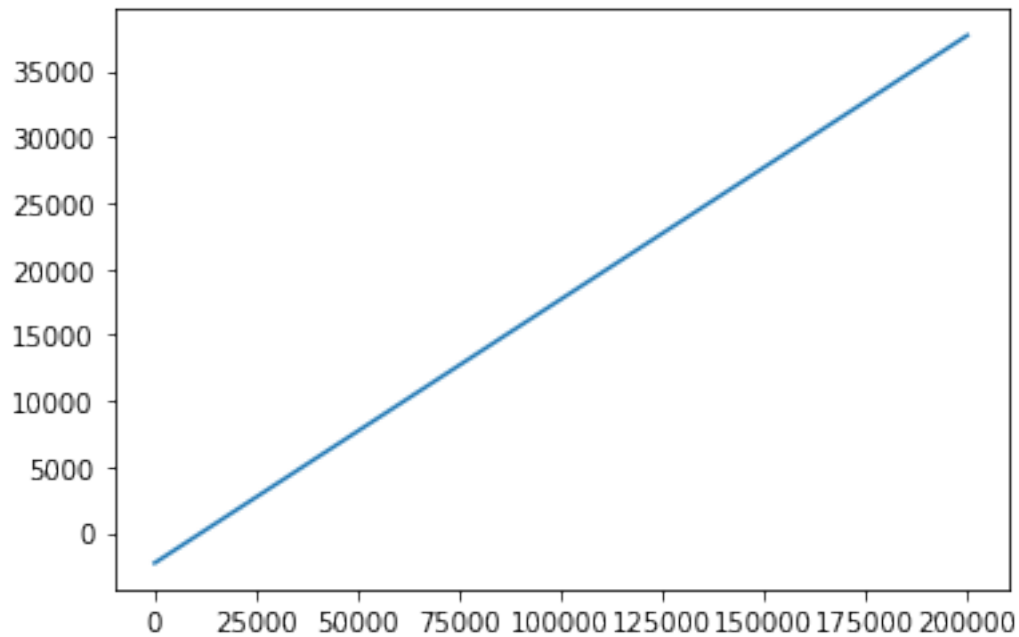
Représentation graphique

Comme d'habitude vous pouvez voir la représentation graphique de votre fonction :

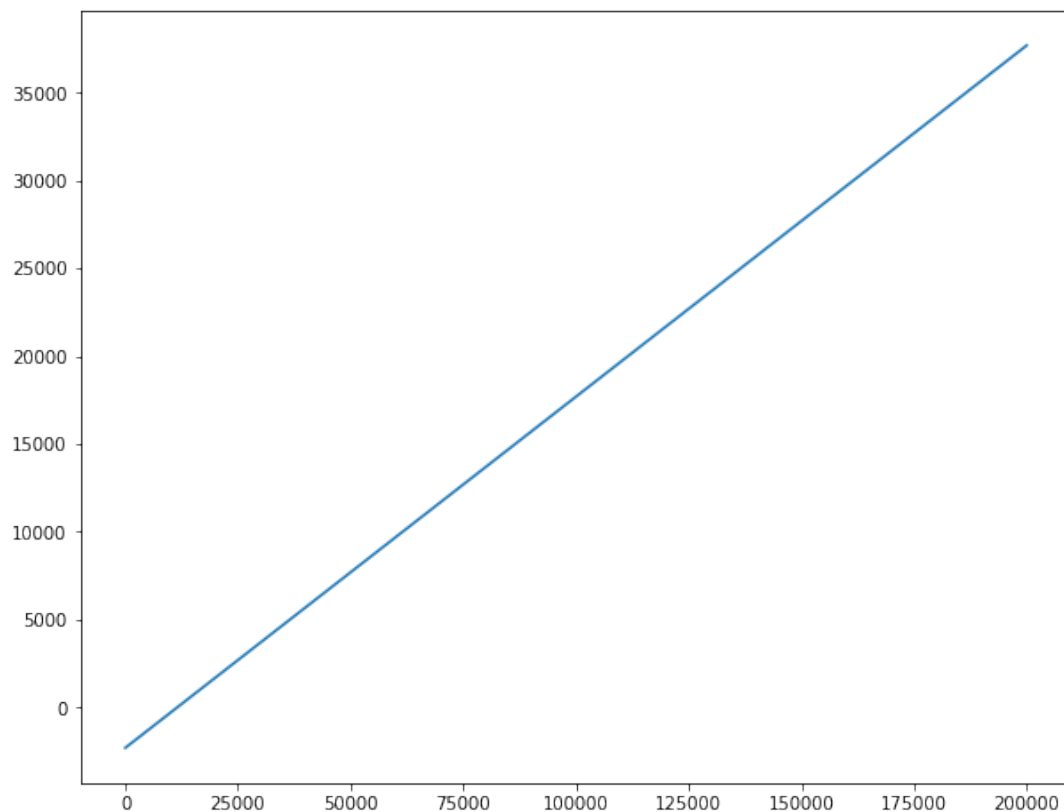
```
[4]: import numpy as np
import matplotlib.pyplot as plt
```

```
[5]: %matplotlib inline
plt.ion()
```

```
[6]: X = np.linspace(0, 200_000)
Y = [taxes(x) for x in X]
plt.plot(X, Y);
```



```
[7]: # et pour changer la taille de la figure
plt.figure(figsize=(10, 8))
plt.plot(X, Y);
```



Merci à Adrien Ollier d'avoir proposé cet exercice

4.13.2 exercice - niveau basique

```
[1]: from corrections.exo_spreadsheet import exo_spreadsheet
```

Le but de l'exercice est d'écrire la fonction `spreadsheet`

- qui prend en entrée un entier - disons `index`
- et qui en sortie retourne le nom de la colonne correspondante dans un fichier Excel.

Pour rappel dans un tableur les colonnes sont appelées d'abord A jusque Z, puis on passe à AA, AB, et ainsi de suite jusque ZZ, puis AAA, etc etc..

La valeur d'entrée `index = 0` est considérée comme non valide.

```
[2]: # voici quelques exemples charnière
exo_spreadsheet.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

Comme vous le voyez, il s'agit en quelque sorte de traduire cet entier en base 26 - si on considère les 26 lettres comme les chiffres (digits) de la numération dans cette base.

Pour vous aider, on rappelle l'existence des fonctions prédéfinies `chr()` et `ord()` qui permettent de passer des caractères Unicode à leur représentation numérique (codepoint) et vice-versa.

On vous invite également à envisager la progression suivante :

1. Exercice intermédiaire : Écrire la fonction qui prend les entiers de 0 à 25 et qui retourne les lettres de A à Z.
2. Écrire la fonction qui prend les entiers de 1 à 26 et qui retourne les lettres de A à Z. La valeur d'entrée `index = 0` est considérée comme une erreur.
3. À partir de la fonction précédente, écrire la fonction `spreadsheet()` pour les indices commençant à 1 et allant au-delà de 26.

```
[3]: # écrivez votre code ici'
def spreadsheet(index):
    ...
```

```
[ ]: # et validez-le ici
exo_spreadsheet.correction(spreadsheet)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

4.14 w4-s3-x4-power

Mise à la puissance

4.14.1 Exercice - niveau basique

```
[1]: # chargement de l'exercice
from corrections.exo_power import exo_power
```

On vous demande d'écrire une fonction qui met un objet `x` à la puissance entière `n`.

Le paramètre `x` doit pouvoir être multiplié par lui même, et le paramètre `n` est un entier ≥ 1 (pas la peine de vérifier).

Naturellement on s'interdit d'utiliser l'opérateur prédéfini `**` ; notamment notre algorithme doit pouvoir s'appliquer à tous les types d'objet qui supportent la multiplication.

Pour optimiser les calculs, on va utiliser un algorithme qui permet d'effectuer un nombre de multiplications en $O(\log_2(n))$.

Pour cela remarquez par exemple que le fait de mettre un nombre au carré nécessite seulement une multiplication ;

Ce qui signifie que la décomposition de `n` en binaire donne une formule pour x^n qui met en jeu de l'ordre de $2 * \log_2 n$ multiplications.

Ainsi par exemple :

— si `n = 21`, c'est-à-dire en base 2 010101, alors

$$n = 2 * 2 * (2 * 2 + 1) + 1$$

$$x^{21} = (((x.x)^{2*2})^2 * x \text{ soit 6 multiplications}$$

— si $n = 42$, c'est-à-dire en base 2 101010, alors

$$n = 2 * (2 * 2 * (2 * 2 + 1) + 1)$$

$$x^{42} = (((x \cdot x)^{2 \cdot x})^2 \cdot x)^2 \text{ soit 7 multiplications}$$

```
[2]: exo_power.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

Commentaires

- on rappelle que $1j$ désigne le nombre complexe i ;
- le jeu de données de test contient des objets (la classe `Number`) pour lesquels l'opérateur `**` ne fonctionne pas;
- le système de correction automatique n'est pas en mesure de vérifier la complexité de votre algorithme, mais sachez que les données de test mettent en jeu des exposants jusqu'à 2^{128} ;

Indice

- on peut être tenté d'écrire une boucle `while` sur la variable n , mais pour commencer une formulation récursive est une approche qui peut sembler beaucoup plus commode à implémenter.

```
[3]: # à vous de jouer
def power(x, n):
    "<votre code>"
```

```
[ ]: # pour vérifier votre code
exo_power.correction(power)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

4.15 w4-s4-c1-scope-builtin

Le module **builtins**

4.15.1 Complément - niveau avancé

Ces noms qui viennent de nulle part

Nous avons vu déjà un certain nombre de fonctions built-in comme par exemple

```
[1]: open, len, zip
```

```
[1]: (<function io.open(file, mode='r', buffering=-1, encoding=None, errors=None
, newline=None, closefd=True, opener=None)>,
<function len(obj, /)>,
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

```
[2]: # 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 built-in (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 :

```
[3]: # nous ne pouvons pas utiliser open puisque
open()
```

ma fonction open

```
[4]: # 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 :

```
[5]: from builtins import open as builtins_open

with builtins_open("builtins.txt", "r", encoding="utf-8") as f:
    print(f.read())
```

quelque chose

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](#) :

```
[6]: dir(builtins)

[6]: ['ArithmeticError',
      'AssertionError',
      'AttributeError',
      'BaseException',
      'BlockingIOError',
      'BrokenPipeError',
      'BufferError',
      'BytesWarning',
      'ChildProcessError',
      'ConnectionAbortedError',
      'ConnectionError',
      'ConnectionRefusedError',
      'ConnectionResetError',
      'DeprecationWarning',
      'EOFError',
      'Ellipsis',
      'EnvironmentError',
      'Exception',
      'False',
      'FileExistsError',
      'FileNotFoundError',
      'FloatingPointError',
      'FutureWarning',
      'GeneratorExit',
      'IOError',
      'ImportError',
      'ImportWarning',
      'IndentationError',
      'IndexError',
      'InterruptedError',
      'IsADirectoryError',
      'KeyError',
      'KeyboardInterrupt',
      'LookupError',
      'MemoryError',
      'ModuleNotFoundError',
      'NameError',
      'None',
```

```
'NotADirectoryError',
'NotImplemented',
'NotImplementedError',
'OSError',
'OverflowError',
'PendingDeprecationWarning',
'PermissionError',
'ProcessLookupError',
'RecursionError',
'ReferenceError',
'ResourceWarning',
'RuntimeError',
'RuntimeWarning',
'StopAsyncIteration',
'StopIteration',
'SyntaxError',
'SyntaxWarning',
'SystemError',
'SystemExit',
'TabError',
'TimeoutError',
'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'ZeroDivisionError',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'breakpoint',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
```

```
'credits',  
'delattr',  
'dict',  
'dir',  
'display',  
'divmod',  
'enumerate',  
'eval',  
'exec',  
'filter',  
'float',  
'format',  
'frozenset',  
'get_ipython',  
'getattr',  
'globals',  
'hasattr',  
'hash',  
'help',  
'hex',  
'id',  
'input',  
'int',  
'isinstance',  
'issubclass',  
'iter',  
'len',  
'license',  
'list',  
'locals',  
'map',  
'max',  
'memoryview',  
'min',  
'next',  
'object',  
'oct',  
'open',  
'ord',  
'pow',  
'print',  
'property',  
'range',  
'repr',  
'reversed',  
'round',  
'set',  
'setattr',  
'slice',  
'sorted',  
'staticmethod',  
'str',  
'sum',  
'super',  
'tuple',  
'type',  
'vars',
```

```
'zip']
```

Vous remarquez que les exceptions (les symboles qui commencent par des majuscules) représentent à elles seules une proportion substantielle de cet espace de noms.

4.16 w4-s4-c2-variable-de-boucle

Visibilité des variables de boucle

4.16.1 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 :

```
[1]: # 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 built-in `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 :

```
[2]: # La variable 'i' n'est pas définie
      try:
          i
      except NameError as e:
          print('OOPS', e)
```

OOPS name 'i' is not defined

```
[3]: # si à présent on fait une boucle
      # avec i comme variable de boucle
      for i in [0]:
          pass

      # alors maintenant i est définie
      i
```

```
[3]: 0
```

On dit que la variable fuit (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 :

```
[4]: # on détruit la variable i si elle existe
     if 'i' in locals():
         del i
```

```
[5]: # 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])
```

[5]: 3

Ça a l'air correct, sauf que :

```
[ ]: # ceci provoque une UnboundLocalError
     length([])

     # NOTE
     # auto-exec-for-latex has skipped execution of this cell
```

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 variable 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 :

```
[6]: # 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
```

```
[7]: # plutôt que de faire ceci
for item in candidates:
    if checks(item):
        break
print('trouvé solution', item)
```

trouvé solution -15

```
[8]: # il vaut mieux faire ceci
solution = None
for item in candidates:
    if checks(item):
        solution = item
        break

print('trouvé solution', solution)
```

trouvé solution -15

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 :

```
[9]: # la fonction length de tout à l'heure
def length1(l):
    for i, x in enumerate(l):
        pass
    return i + 1
```

```
[10]: # 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
```

```
[ ]: # comme ci-dessus: UnboundLocalError
length1([])

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

```
[11]: length2([])
```

```
[11]: 0
```

Les compréhensions

Notez bien que par contre, les variables de compréhension ne fuient pas (contrairement à ce qui se passait en Python 2) :

```
[12]: # on détruit la variable i si elle existe
if 'i' in locals():
    del i
```

```
[13]: # en Python 3, les variables de compréhension ne fuient pas
[i**2 for i in range(3)]
```

```
[13]: [0, 1, 4]
```

```
[14]: # ici i est à nouveau indéfinie
try:
    i
except NameError as e:
    print("OOPS", e)
```

OOPS name 'i' is not defined

4.17 w4-s4-c3-unboundlocalerror

L'exception `UnboundLocalError`

4.17.1 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 :


```
[1]: def ma_fonction1():  
    variable1 = "locale"  
    print(variable1)  
  
ma_fonction1()
```

locale

et ceci que l'on ait ou non une variable globale de même nom

```
[2]: variable2 = "globale"  
  
def ma_fonction2():  
    variable2 = "locale"  
    print(variable2)  
  
ma_fonction2()
```

locale

Variable globale

On peut accéder en lecture à une variable globale sans précaution particulière :

```
[3]: variable3 = "globale"  
  
def ma_fonction3():  
    print(variable3)  
  
ma_fonction3()
```

globale

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 :

```
[4]: # 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}")
```

```
OOPS, exception <class 'UnboundLocalError':  
local variable 'variable4' referenced before assignment
```

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 :

```
[5]: # 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)
```

dans la fonction globale
après la fonction changée localement

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.

4.18 w4-s5-c1-fonctions-globals-et-locals

Les fonctions **globals** et **locals**

4.18.1 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 built-in `globals` et `locals`, que nous allons commencer par essayer sur quelques exemples. Nous avons pour cela écrit un module dédié :

```
[1]: import env_locals_globals
```

dont voici le code

```
[2]: from modtools import show_module
show_module(env_locals_globals)
```

Fichier `/Users/tparment/git/flotpython-course/modules/env_locals_globals.py`

```
-----
01|"""
02|un module pour illustrer les fonctions globals et locals
03|"""
04|
05|globale = "variable globale au module"
06|
07|def display_env(env):
08|    """
09|    affiche un environnement
10|    on affiche juste le nom et le type de chaque variable
11|    """
12|    for variable, valeur in sorted(env.items()):
13|        print("{:>20} → {}".format(variable, type(valeur).__name__))
14|
15|def temoin(x):
16|    "la fonction témoin"
17|    y = x ** 2
18|    print(20 * '-', 'globals:')
19|    display_env(globals())
20|    print(20 * '-', 'locals:')
21|    display_env(locals())
22|
23|class Foo:
24|    "une classe vide"
```

et voici ce qu'on obtient lorsqu'on appelle

```
[3]: env_locals_globals.temoin(10)
```

```
----- globals:
          Foo → type
    __builtins__ → dict
```

```

    __cached__ → str
    __doc__ → str
    __file__ → str
    __loader__ → SourceFileLoader
    __name__ → str
    __package__ → str
    __spec__ → ModuleSpec
display_env → function
    globale → str
    temoin → function
----- locals:
        x → int
        y → int

```

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.

```
[4]: import env_locals
```

```
[5]: # le code de ce module
      show_module(env_locals)
```

Fichier `/Users/tparment/git/flotpython-course/modules/env_locals.py`

```

-----
01|"""
02|un module pour illustrer la fonction locals
03|"""
04|
05|# pour afficher
06|from env_locals_globals import display_env
07|
08|def temoin(x):
09|    "la fonction témoin"
10|    y = x ** 2
11|    print(20*'- ', 'locals - entrée:')
12|    display_env(locals())
13|
14|    for i in (1,):
15|        for j in (1,):
16|            print(20*'- ', 'locals - boucles for:')
17|            display_env(locals())
18|

```

```
[6]: env_locals.temoin(10)
```

```

----- locals - entrée:
        x → int
        y → int
----- locals - boucles for:
        i → int
        j → int
        x → int
        y → int

```

4.18.2 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 :

```
[7]: "{nom}".format(nom="Dupont")
```

```
[7]: 'Dupont'
```

que l'on peut obtenir de manière équivalente, en anticipant sur la prochaine vidéo, avec le passage d'arguments en `**` :

```
[8]: "{nom}".format(**{'nom': 'Dupont'})
```

```
[8]: 'Dupont'
```

En versant la fonction `locals` dans cette formule on obtient une forme relativement élégante

```
[9]: def format_et_locals(nom, prenom, civilite, telephone):
      return "{civilite} {prenom} {nom} : Poste {telephone}".format(**locals())

      format_et_locals('Dupont', 'Jean', 'Mr', '7748')
```

```
[9]: 'Mr Jean Dupont : Poste 7748'
```

Avec l'opérateur `%`

De manière similaire, avec l'opérateur `%` - dont nous rappelons qu'il est obsolète - on peut écrire

```
[10]: 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')
```

```
[10]: 'Mr Jean Dupont : Poste 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 :

```
[11]: # 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')
```

```
[11]: 'Mr Jean Dupont : Poste 7748'
```

4.19 w4-s6-c1-passage-arguments

Passage d'arguments

4.19.1 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.

```
[1]: print("la fonction", "print", "peut", "prendre", "plein", "d'arguments")
```

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 :

```
[2]: # 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()
```

```
ERROR problème dans la fonction foo
ERROR
```

Légère variation

Pour sophistication 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 :

```
[3]: # 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])
```

```
ERROR message d'erreur code 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 :

```
[4]: # 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 :

```
[5]: # 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

```
[6]: # 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')
```

```
la ligne (0, 0) -> (100, 100) en noir
la ligne (0, 100) -> (100, 0) en 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.

4.19.2 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

```
[7]: # 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)
```

la ligne (0, 0) -> (100, 100) en rouge

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

```
[8]: # 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`.

```
[9]: 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

```
[10]: ligne_rouge(0, 100, 100, 0, epaisseur=4)
```

la ligne (0, 100) -> (100, 0) en rouge - ep. 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 positionnels (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)

```
[11]: # une fonction qui prend 4 paramètres simples
def foo(a, b, c, d):
    print(a, b, c, d)
```

```
[12]: # on peut l'appeler par exemple comme ceci
foo(1, c=3, *(2,), **{'d':4})
```

1 2 3 4

```
[13]: # mais pas comme cela
try:
    foo (1, b=3, *(2,), **{'d':4})
except Exception as e:
    print(f"OOPS, {type(e)}, {e}")
```

OOPS, <class 'TypeError'>, foo() got multiple values for argument 'b'

Si le problème ne vous semble pas clair, vous pouvez regarder la [documentation python décrivant ce problème](#).

4.20 w4-s6-c2-pas-de-valeur-par-defaut-mutable

Un piège courant

4.20.1 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

```
[1]: # ne faites SURTOUT PAS ça
def ne_faites_pas_ca(options={}):
    "faire quelque chose"
```

```
[2]: # mais plutôt comme ceci
def mais_plutot_ceci(options=None):
    if options is None:
        options = {}
    "faire quelque chose"
```

4.20.2 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 :

```
[3]: 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

```
[4]: ajouter_un_aleatoire()
```

```
[4]: [9]
```

Sauf que, si on appelle la fonction une deuxième fois, on a une surprise !

```
[5]: ajouter_un_aleatoire()
```

[5]: [9, 10]

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](#).

4.21 w4-s6-c3-keyword-only

Arguments keyword-only

4.21.1 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 reliquat des arguments positionnels
4. paramètres `**kwds` qui attrape dans un dictionnaire le reliquat des arguments nommés

Pour rappel :

```
[1]: # 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}")
```

```
[2]: foo(1)
```

a=1, b=100, args=(), kwds={}

```
[3]: foo(1, 2)
```

a=1, b=2, args=(), kwds={}

```
[4]: foo(1, 2, 3)
```

a=1, b=2, args=(3,), kwds={}

```
[5]: foo(1, 2, 3, bar=1000)
```

```
a=1, b=2, args=(3,), kwds={'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 Python 2, c'est qu'à l'époque on devait impérativement les déclarer dans cet ordre :

positionnels, 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

```
[6]: # 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

```
[7]: # je peux toujours faire ceci
bar(1)
```

```
a=1, b=100, args=(), kwds={}
```

```
[8]: # mais si je fais ceci l'argument 2 va aller dans args
bar(1, 2)
```

```
a=1, b=100, args=(2,), kwds={}
```

```
[9]: # pour passer b=2, je **dois** nommer mon argument
bar(1, b=2)
```

```
a=1, b=2, args=(), kwds={}
```

Ce trait n'est objectivement pas utilisé massivement en Python, mais cela peut être utile de le savoir :

- en tant qu'utilisateur d'une bibliothèque, 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.

4.22

w4-s6-x1-passage-arguments

Passage d'arguments

4.22.1 Exercice - niveau basique

```
[1]: # 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 : $distance(x_1, \dots, x_n) = \sqrt{\sum x_i^2}$

Par convention on fixe que $distance() = 0$

```
[2]: # des exemples
exo_distance.example()
```

GridBox(children=(HTML(value='appel', _dom_classes=('header

```
[3]: # ATTENTION vous devez aussi définir les arguments de la fonction
def distance(votre, signature):
    return "votre code"
```

```
[ ]: # la correction
exo_distance.correction(distance)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

4.22.2 Exercice - niveau intermédiaire

```
[4]: # 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.

```
[5]: # par exemple
     exo_numbers.example()
```

GridBox(children=(HTML(value='appel', _dom_classes=('header

En guise d'indice, je vous invite à regarder les fonctions built-in `sum`, `min` et `max`.

```
[6]: # vous devez définir votre propre signature
     def numbers(votre, signature):
         "<votre_code>"
```

```
[ ]: # pour vérifier votre code
     exo_numbers.correction(numbers)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```


Chapitre 5

Itération, importation et espace de nommage

5.1 w5-s1-c1-break-et-continue

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 boucles **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 :

```
[1]: 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")
```

```
on traite l'entier 0  
on traite l'entier 10  
on traite l'entier 20  
on traite l'entier 30  
on traite l'entier 40  
on traite l'entier 50  
on est sorti de la boucle
```

Pour aller plus loin, vous pouvez lire [cette documentation](#).

5.2 w5-s1-c2-ne-pas-modifier-sujet-boucle-for

Une limite de la boucle **for**

5.2.1 Complément - niveau basique

Pour ceux qui veulent suivre le cours au niveau basique, reprenez 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 :

```
[1]: # on veut enlever de l'ensemble toutes les chaînes
# qui ne contiennent pas 'bert'
ensemble = {'marc', 'albert'}

[ ]: # ceci semble une bonne idée mais ne fonctionne pas
# provoque RuntimeError: Set changed size during iteration

for valeur in ensemble:
    if 'bert' not in valeur:
        ensemble.discard(valeur)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Comment faire alors ?

Première remarque, votre premier réflexe pourrait être de penser à une compréhension d'ensemble :

```
[2]: ensemble2 = {valeur for valeur in ensemble if 'bert' in valeur}
ensemble2
```

```
[2]: {'albert'}
```

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.

```
[3]: from copy import copy
# on veut enlever de l'ensemble toutes les chaînes
# 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)
```

```
{'albert'}
```

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 :

```
[4]: liste = [[1], [2], [3]]
     print('avant', liste)
```

avant [[1], [2], [3]]

```
[5]: for sous_liste in liste:
     sous_liste.append(100)
     print('après', liste)
```

après [[1, 100], [2, 100], [3, 100]]

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 ambiguïté 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 que la boucle 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 théoriquement jamais. En pratique, ce sera le cas quand votre système n'aura plus de mémoire disponible (sauvegardez vos documents avant d'essayer!).

5.3 w5-s1-c3-itertools

Itérateurs

5.3.1 Complément - niveau intermédiaire

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.

```
[1]: 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 :

```
[2]: for x in itertools.chain((1, 2), [3, 4]):  
      print(x)
```

1
2
3
4

— `islice` qui fournit un itérateur sur un slice d'un itérable. On peut le voir comme une généralisation de `range` qui parcourt n'importe quel itérable.

```
[3]: import string  
support = string.ascii_lowercase  
print(f'support={support}')
```

support=abcdefghijklmnopqrstuvwxyz

```
[4]: # range  
for x in range(3, 8):  
    print(x)
```

3
4
5
6
7

```
[5]: # islice  
for x in itertools.islice(support, 3, 8):  
    print(x)
```

d
e
f
g
h

5.4 w5-s2-c1-fonctions

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

$$x_1 \otimes x_2 \dots \otimes x_n$$

De manière un peu moins abstraite, on suppose qu'on dispose d'une fonction binaire `f` qui implémente l'opérateur \otimes , et alors

$$\text{reduce}(f, [x_1, \dots, x_n]) = f(\dots f(f(x_1, x_2), x_3), \dots, x_n)$$

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.

```
[1]: # la fonction reduce dans Python 3 n'est plus une built-in comme en Python 2
# 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)}")
```

0 -> 1

1 -> 1

2 -> 2
3 -> 6
4 -> 24

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** :

```
[2]: entrees = [8, 5, 12, 4, 45, 7]
```

```
print('sum', sum(entrees))  
print('min', min(entrees))  
print('max', max(entrees))
```

```
sum 81  
min 4  
max 45
```

5.5 w5-s2-c2-tris-de-listes-2

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 :

```
[1]: 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 :

```
[2]: def longitude(element):  
    return element[1]  
  
coordonnees.sort(key=longitude)  
print("coordonnées triées par longitude", coordonnees)
```

```
coordonnées triées par longitude [(46, -7), (46, 0), (43, 7)]
```

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 méthode `itemgetter` du module `operator`

```
[3]: # fonction lambda
coordonnees = [(43, 7), (46, -7), (46, 0)]
coordonnees.sort(key=lambda x: x[1])
print("coordonnées triées par longitude", coordonnees)

# méthode operator.itemgetter
import operator
coordonnees = [(43, 7), (46, -7), (46, 0)]
coordonnees.sort(key=operator.itemgetter(1))
print("coordonnées triées par longitude", coordonnees)
```

```
coordonnées triées par longitude [(46, -7), (46, 0), (43, 7)]
coordonnées triées par longitude [(46, -7), (46, 0), (43, 7)]
```

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` :

```
[4]: 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)
```

```
la liste triée est une copie [9, 8, 7, 6, 5, 4, 3, 2, 1]
la liste initiale est intacte [8, 7, 4, 3, 2, 9, 1, 5, 6]
```

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 :

```
[5]: 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)
```

```
la liste triée est une copie [9, 8, 7, 6, 5, 4, 3, 2, 1]
la liste initiale est intacte [8, 7, 4, 3, 2, 9, 1, 5, 6]
```


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

```
[1]: # 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.

```
[2]: # voici un exemple de ce qui est attendu
exo_multi_tri.example()
```

GridBox(children=(HTML(value='appel', _dom_classes=('header

Écrivez votre code ici :

```
[3]: def multi_tri(listes):
      "<votre_code>"
```

```
[ ]: # pour vérifier votre code
exo_multi_tri.correction(multi_tri)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

5.5.3 Exercice - niveau intermédiaire

Tri de plusieurs listes, dans des directions différentes

```
[4]: # 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.

```
[5]: # Pour être un peu plus clair, voici à quoi on s'attend
      exo_multi_tri_reverse.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

À vous de jouer :

```
[6]: def multi_tri_reverse(listes, reverses):
      "<votre_code>"
```

```
[ ]: # et pour vérifier votre code
      exo_multi_tri_reverse.correction(multi_tri_reverse)

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

5.5.4 Exercice - niveau intermédiaire

Tri de listes, avec critère personnalisé

Cet exercice consiste à écrire une fonction de tri selon un critère choisi par le programmeur.

On vous passe en entrée une liste de dictionnaires qui :

- ont tous les clés 'n' et 'p' (pensez nom et prénom, je choisis des noms courts pour que la présentation des données soit plus compacte)
- ont parfois une clé 'p2' (deuxième prénom)

L'exercice consiste à trier la liste en place, selon le critère suivant :

- on trie d'abord les gens selon leur prénom
- en cas de prénoms identiques, on trie selon les entrées concernées selon le nom
- en cas d'homonymes pour le nom et le prénom, on retient d'abord les gens qui n'ont pas de deuxième prénom, et on trie les autres selon leur deuxième prénom.

```
[7]: # voyons cela sur un premier exemple
      from corrections.exo_tri_custom import exo_tri_custom
      exo_tri_custom.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[8]: def tri_custom(liste):
      ... # votre code
      return liste
```

Indice on peut bien sûr utiliser `list.sort()` pour faire ce travail en quelques lignes ; voyez notamment le paramètre `key`.

```
[ ]: # pour corriger votre code
exo_tri_custom.correction(tri_custom)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

5.5.5 Exercice - niveau intermédiaire

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.

```
[1]: # 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 multiplié par deux.

Formellement : `doubler_premier(f, x1, x2, ..., xn) == f(2*x1, x2, ..., xn)`

Voici d'abord quelques exemples de ce qui est attendu. Pour cela on va utiliser comme fonctions :

- `add` et `mul` sont les opérateurs (binaires) du module `operator` ;
- et `distance` est la fonction qu'on a vu dans un exercice précédent ; pour rappel

$$\text{distance}(x_1, \dots, x_n) = \sqrt{\sum x_i^2}$$

```
[2]: # rappel sur la fonction distance:
from corrections.exo_distance import distance
distance(3.0, 4.0)
```

```
[2]: 5.0
```

```
[3]: distance(4.0, 4.0, 4.0, 4.0)
```

```
[3]: 8.0
```

```
[4]: # voici donc quelques exemples de ce qui est attendu.
exo_doubler_premier.example()
```

GridBox(children=(HTML(value='arguments', _dom_classes=('he

```
[5]: # ATTENTION vous devez aussi définir les arguments de la fonction
def doubler_premier(votre, signature):
    return "votre code"
```

```
[ ]: exo_doubler_premier.correction(doubler_premier)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

5.5.6 Exercice - niveau intermédiaire

```
[6]: # 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.

```
[7]: # 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()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;">'>arguments</span>', _dom_classes=('he
```

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.

```
[8]: # ATTENTION vous devez aussi définir les arguments de la fonction
def doubler_premier_kwds(votre, signature):
    "<votre code>"
```

```
[ ]: exo_doubler_premier_kwds.correction(doubler_premier_kwds)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

5.6 w5-s2-x3-compare

Comparaison de fonctions

5.6.1 Exercice - niveau avancé

```
[1]: # 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.

```
[2]: # par exemple
exo_compare_all.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>arguments</span>', _dom_classes=('he
```

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.

```
[3]: # c'est à vous
def compare_all(f, g, entrees):
    "<votre code>"
```

```
[ ]: # pour vérifier votre code
exo_compare_all.correction(compare_all)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

5.6.2 Exercice optionnel - niveau avancé

```
[4]: # 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 $\text{len}(\text{argument_tuples})$ vaut n :

$\forall i \in \{1, \dots, n\}$, si $\text{argument_tuples}[i] == [a_1, \dots, a_j]$, alors

$\text{retour}(i) == \text{True} \iff f(a_1, \dots, a_j) == g(a_1, \dots, a_j)$

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.

```
[5]: exo_compare_args.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>arguments</span>', _dom_classes=('he
```

```
[6]: # ATTENTION vous devez aussi définir les arguments de la fonction
def compare_args(votre, signature):
    "<votre_code>"
```

```
[ ]: exo_compare_args.correction(compare_args)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

5.7 w5-s3-c1-comprehensions

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 :

```
[1]: depart = (-5, -3, 0, 3, 5, 10)
      arrivee = [x**2 for x in depart]
      arrivee
```

```
[1]: [25, 9, 0, 9, 25, 100]
```

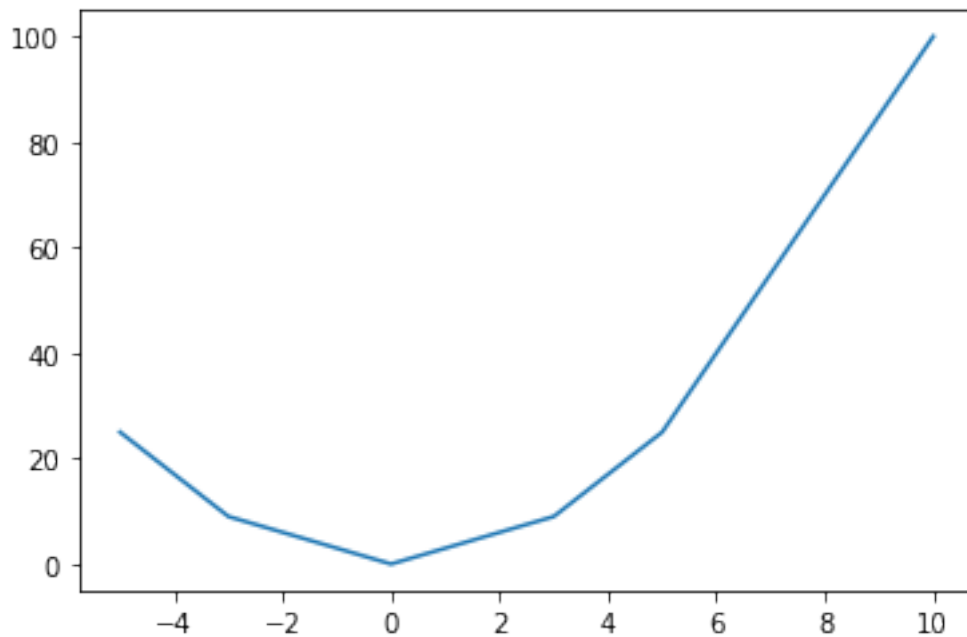
Le résultat de cette expression est donc une liste, dont les éléments sont les résultats de l'expression `x**2` pour `x` prenant toutes les valeurs de `depart`.

Remarque : si on prend un point de vue un peu plus mathématique, ceci revient donc à appliquer une certaine fonction (ici $x \rightarrow x^2$) 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

```
[2]: # profitons de cette occasion pour voir
      # comment tracer une courbe avec matplotlib
      %matplotlib inline
      import matplotlib.pyplot as plt
      plt.ion()
```

```
[3]: # si on met le départ et l'arrivée
      # en abscisse et en ordonnée, 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 :

```
[4]: [x**2 for x in depart if x%2 == 0]
```

```
[4]: [0, 100]
```

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 :

```
[5]: [ord(x) for x in 'abc']
```

```
[5]: [97, 98, 99]
```

```
[6]: [chr(x) for x in (97, 98, 99)]
```

```
[6]: ['a', 'b', 'c']
```


Autres types (2)

On peut également construire par compréhension des dictionnaires et des ensembles :

```
[7]: d = {x: ord(x) for x in 'abc'}  
d
```

```
[7]: {'a': 97, 'b': 98, 'c': 99}
```

```
[8]: e = {x**2 for x in (97, 98, 99) if x % 2 == 0}  
e
```

```
[8]: {9604}
```

Pour en savoir plus

Voyez [la section sur les compréhensions de liste](#) dans la documentation python.

5.8

w5-s3-c2-comprehensions-imbriquees

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 :

```
[1]: [n + p for n in [2, 4] for p in [10, 20, 30]]
```

```
[1]: [12, 22, 32, 14, 24, 34]
```

Bien sûr on peut aussi restreindre ces compréhensions, comme par exemple :

```
[2]: [n + p for n in [2, 4] for p in [10, 20, 30] if n*p >= 40]
```

```
[2]: [22, 32, 14, 24, 34]
```

Observez surtout que le résultat ci-dessus est une liste simple (de profondeur 1), à comparer avec :

```
[3]: [[n + p for n in [2, 4]] for p in [10, 20, 30]]
```

```
[3]: [[12, 14], [22, 24], [32, 34]]
```

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 :

```
[4]: # 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
```

```
[4]: [22, 32, 14, 24, 34]
```

Ordre d'évaluation de `[[.. for ..] .. for ..]`

Pour rappel, on peut imbriquer des compréhensions de compréhensions. Commençons par poser

```
[5]: n = 4
```

On peut alors créer une liste de listes comme ceci :

```
[6]: [(i, j) for i in range(1, j + 1) for j in range(1, n + 1)]
```

```
[6]: [(1, 1),
      (1, 2), (2, 2)],
      [(1, 3), (2, 3), (3, 3)],
      [(1, 4), (2, 4), (3, 4), (4, 4)]]
```

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 :

```
[7]: # 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
```

```
[7]: [(1, 1),
      (1, 2), (2, 2)],
      [(1, 3), (2, 3), (3, 3)],
      [(1, 4), (2, 4), (3, 4), (4, 4)]]
```

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 :

```
[8]: [[(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]
```

```
[8]: [(2, 2)], [(2, 4), (4, 4)]
```

ce qui est équivalent à :

```
[9]: # 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
```

```
[9]: [(2, 2)], [(2, 4), (4, 4)]
```

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 fuient

Nous avons déjà signalé que les variables de boucle restent définies après la sortie de la boucle, ainsi nous pouvons examiner :

```
[10]: i, j
```

```
[10]: (4, 4)
```

C'est pourquoi, afin de comparer les deux formes de compréhension imbriquées nous allons explicitement retirer les variables `i` et `j` de l'environnement

```
[11]: 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 :

```
[12]: [(x, y) for x in [1, 2] for y in [1, 2]]
```

```
[12]: [(1, 1), (1, 2), (2, 1), (2, 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 :

```
[ ]: # ceci ne fonctionne pas
      # NameError: name 'j' is not defined

      [ (i, j) for i in range(1, j + 1) for j in range(1, n + 1) ]

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

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 :

```
[ ]: # 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))

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

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 w5-s3-x1-comprehensions

Compréhensions

5.9.1 Exercice - niveau basique

```
[1]: # 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.

```
[2]: # par exemple
      exo_aplatir.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[3]: def applatir(conteneurs):
      "<votre_code>"
```

```
[ ]: # vérifier votre code
      exo_aplatir.correction(aplatir)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

5.9.2 Exercice - niveau intermédiaire

```
[4]: # 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`.

```
[5]: # exemple
      exo_alternat.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

Indice pour cet exercice il peut être pertinent de recourir à la fonction built-in `zip`.

```
[6]: def alternat(c1, c2):
      "<votre_code>"
```

```
[ ]: # pour vérifier votre code
      exo_alternat.correction(alternat)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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`.

```
[7]: # un exemple
from corrections.exo_intersect import exo_intersect
exo_intersect.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[8]: def intersect(A, B):
      "<votre_code>"
```

```
[ ]: # pour vérifier votre code
exo_intersect.correction(intersect)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

5.10 w5-s3-x2-vigenere

Le code de Vigenère

5.10.1 Exercice, niveau intermédiaire

Le [code ou chiffre de Vigenère](#) est une méthode de chiffrement très rustique, qui est une version un peu améliorée du [chiffre de César](#).

Les deux méthodes fonctionnent par simple décalage dans l'alphabet modulo 26.

Dans le chiffre de César, on se donne une clé constituée d'un seul caractère, disons par exemple C (la 3-ième lettre de l'alphabet), et avec cette clé on chiffre l'alphabet par un décalage de 3, ce qui donne :

```
clé      : C
clair    : ABCDEFGHIJKLMNOPQRSTUVWXYZ
chiffré  : DEFGHIJKLMNOPQRSTUVWXYZABC
```

ou avec d'autres clés

```
clé      : L
clair    : ABCDEFGHIJKLMNOPQRSTUVWXYZ
chiffré  : MNOPQRSTUVWXYZABCDEFGHIJKL
```

```
clé      : E
clair    : ABCDEFGHIJKLMNOPQRSTUVWXYZ
chiffré  : FGHIJKLMNOPQRSTUVWXYZABCDE
```

La méthode de Vigenère consiste à choisir cette fois pour clé un mot, qui est utilisé de manière répétitive.

Ainsi par exemple si je choisis la clé CLE, on va chiffrer un message en appliquant la méthode de César

- avec la clé 'C' sur le 1-er caractère,
- avec la clé 'L' sur le 2-ème caractère,
- avec la clé 'E' sur le 3-ème caractère,
- avec la clé 'C' sur le 4-ème caractère,
- avec la clé 'L' sur le 5-ème caractère,
- ...

Le but de cet exercice est d'écrire une fonction qui implémente la méthode de Vigenère pour, à partir d'une clé connue, coder ou décoder des messages.

5.10.2 Première partie : le code de César

Dans un premier temps on se propose d'implémenter le code de César ; pour rester simple, nous allons nous limiter à ne chiffrer que les caractères alphabétiques dans la plage des caractères ASCII, c'est-à-dire sans accent, cédille ou autre.

Je rappelle par ailleurs l'existence en Python de deux fonctions qui peuvent être très utiles dans ce contexte :

- `ord()` qui projette les caractères vers les entiers (codepoints)
- et `chr()` qui réalise l'opération inverse.

```
[1]: # la fonction ord() retourne le codepoint
      # d'un caractère
      ord('a')
```

```
[1]: 97
```

```
[2]: # et réciproquement avec chr()
      chr(97)
```

```
[2]: 'a'
```

Une fois qu'on a dit ça, il est intéressant de constater que les caractères minuscules et majuscules auxquels nous nous intéressons sont, fort heureusement, contigus dans l'espace des codepoints.

```
[3]: import string
      string.ascii_letters
```

```
[3]: 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```
[4]: string.ascii_lowercase
```

```
[4]: 'abcdefghijklmnopqrstuvwxyz'
```

```
[5]: COLUMNS = 7
```

```
[6]: for index, char in enumerate(string.ascii_uppercase, 1):
      print(f"{char}→{ord(char):3d} ", end=" ")
      if index % COLUMNS == 0:
          print()
```

```
A→ 65 B→ 66 C→ 67 D→ 68 E→ 69 F→ 70 G→ 71
H→ 72 I→ 73 J→ 74 K→ 75 L→ 76 M→ 77 N→ 78
O→ 79 P→ 80 Q→ 81 R→ 82 S→ 83 T→ 84 U→ 85
```

```
[7]: for index, char in enumerate(string.ascii_lowercase, 1):
      print(f"{char}→{ord(char):3d} ", end=" ")
      if index % COLUMNS == 0:
```

```
print()
```

```
a→ 97 b→ 98 c→ 99 d→100 e→101 f→102 g→103
h→104 i→105 j→106 k→107 l→108 m→109 n→110
o→111 p→112 q→113 r→114 s→115 t→116 u→117
```

Forts de ces observations, vous devez pouvoir à présent écrire une première fonction qui implémente le décalage de César.

Comme par ailleurs les opérations d'encodage et de décodage sont symétriques l'une de l'autre, on choisit pour éviter d'avoir à dupliquer du code, d'écrire une fonction dont la signature est :

```
def cesar(clear, key, encode=True):
    # retourne un caractère
```

La fonction en question doit :

- laisser le texte tel quel si ce n'est pas un caractère alphabétique ASCII,
- accepter une clé qui peut être minuscule ou majuscule ; dit autrement, deux clés qui valent 'C' et 'c' produisent toutes les deux le même résultat,
- retourner une majuscule si le texte clair est en majuscule et une minuscule si le texte en clair est une minuscule.

Voici ce que cela donnerait sur quelques exemples :

```
[8]: from corrections.exo_vigenere import exo_cesar
```

```
[9]: exo_cesar.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[10]: # à vous de jouer pour implémenter la fonction cesar
def cesar(clear, key, encode=True):
    pass
```

```
[ ]: # et pour vous corriger
exo_cesar.correction(cesar)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

5.10.3 Deuxième partie : le code de Vigenère

Cette première partie étant acquise, nous pouvons passer à l'amélioration de Vigenère, qui comme on l'a vu dans l'introduction consiste à prendre un mot dont on utilise les lettres successivement, et en boucle, comme clé pour la méthode de César.

Donc pour calculer le chiffrement de ce message avec la clé cle, on va se souvenir que

```
clé      : C
clair    : ABCDEFGHIJKLMNOPQRSTUVWXYZ
chiffré  : DEFGHIJKLMNOPQRSTUVWXYZABC
```



```

clé      : L
clair    : ABCDEFGHIJKLMNOPQRSTUVWXYZ
chiffré  : MNOPQRSTUVWXYZABCDEFGHIJKL

```

```

clé      : E
clair    : ABCDEFGHIJKLMNOPQRSTUVWXYZ
chiffré  : FGHIJKLMNOPQRSTUVWXYZABCDE

```

et du coup faire

```

cesar('c', 'c') → 'f'
cesar('e', 'l') → 'q'
cesar(' ', 'e') → ' '
cesar('m', 'c') → 'p'
cesar('e', 'l') → 'q'
cesar('s', 'e') → 'x'
cesar('s', 'c') → 'v'
cesar('a', 'l') → 'm'
cesar('g', 'e') → 'l'
cesar('e', 'c') → 'h'

```

Voyons cet exemple sous forme de code :

```
[11]: from corrections.exo_vigenere import exo_vigenere
```

```
[12]: exo_vigenere.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

indices

- Bien entendu vous êtes invités à utiliser la fonction `cesar` pour implémenter `vigenere`.
- Par ailleurs, pour cet exercice je vous recommande d’aller voir ou revoir le module `itertools` qui contient des outils qui sont exactement adaptés à ce traitement.
C’est-à-dire, pour être encore plus explicite, qu’il est possible d’écrire cette fonction sans recourir à aucun indice entier sur le texte ni sur la clé.

```
[13]: # à vous de jouer
def vigenere(clear, key, encode=True):
    pass
```

```
[ ]: # et pour corriger
exo_vigenere.correction(vigenere)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

5.11 w5-s4-c1-expressions-generatrices

Expressions génératrices

5.11.1 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 transformer 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.

```
[1]: # c'est une compréhension de liste
comprehension = [x**2 for x in range(100) if x%17 == 0]
print(comprehension)
```

```
[0, 289, 1156, 2601, 4624, 7225]
```

```
[2]: # c'est une expression génératrice
generator = (x**2 for x in range(100) if x%17 == 0)
print(generator)
```

```
<generator object <genexpr> at 0x10f7ec750>
```

Ensuite pour utiliser une expression génératrice, c'est très simple, on l'utilise comme n'importe quel itérateur.

```
[3]: generator is iter(generator) # generator est bien un itérateur
```

```
[3]: True
```

```
[4]: # affiche les premiers carrés des multiples de 17
for count, carre in enumerate(generator, 1):
    print(f'Contenu de generator après {count} itérations : {carre}')
```

```
Contenu de generator après 1 itérations : 0
Contenu de generator après 2 itérations : 289
Contenu de generator après 3 itérations : 1156
Contenu de generator après 4 itérations : 2601
Contenu de generator après 5 itérations : 4624
Contenu de generator après 6 itérations : 7225
```

Avec une expression génératrice on n'est plus limité comme avec les compréhensions par le nombre d'éléments :

```
[5]: # 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)
```

```
{617721316, 4536561316, 3617541316, 311381316}
```

5.11.2 Complément - niveau intermédiaire

Compréhension vs expression génératrice

Digression : liste vs itérateur En Python 3, 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`

```
[6]: # on peut parcourir un range comme si c'était une liste
for i in range(4):
    print(i)
```

```
0
1
2
3
```

mais en réalité le résultat de `range` exhibe un comportement un peu étrange, en ce sens que :

```
[7]: # mais en fait la fonction range ne renvoie PAS une liste (depuis Python 3)
range(4)
```

```
[7]: range(0, 4)
```

```
[8]: # et en effet ce n'est pas une liste
isinstance(range(4), list)
```

```
[8]: False
```

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 successivement aux entrées.

Or dans les cas où, comme pour `range`, on n'a pas réellement besoin de cette liste en tant 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.

```
[9]: 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
```

```
[9]: <generator object <genexpr> at 0x10f7eca98>
```

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 :

```
[10]: for x, y in zip(depart, arrivee2):
       print(f"x={x} => y={y}")
```

```
x=-5 => y=25
x=-3 => y=9
x=0 => y=0
x=3 => y=9
x=5 => y=25
x=10 => y=100
```

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 matière 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 :

```
[11]: 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.

5.12 w5-s4-c2-yield-from

yield from pour cascader deux générateurs

Dans ce notebook nous allons voir comment fabriquer une fonction génératrice qui appelle elle-même une autre fonction génératrice.

5.12.1 Complément - niveau avancé

Une fonction génératrice

Commençons à nous définir une fonction génératrice; par exemple ici nous listons les diviseurs d'un entier, en excluant 1 et l'entier lui-même :

```
[1]: def divs(n, verbose=False):  
    for i in range(2, n):  
        if n % i == 0:  
            if verbose:  
                print(f'trouvé diviseur {i} de {n}')  
            yield i
```

Comme attendu, l'appel direct à cette fonction ne donne rien d'utile :

```
[2]: divs(28)
```

```
[2]: <generator object divs at 0x103f665e8>
```

Mais lorsqu'on l'utilise dans une boucle `for` :

```
[3]: for d in divs(28):  
    print(d)
```

```
2  
4  
7  
14
```

Une fonction génératrice qui appelle une autre fonction génératrice

Bien, jusqu'ici c'est clair. Maintenant supposons que je veuille écrire une fonction génératrice qui énumère tous les diviseurs de tous les diviseurs d'un entier. Il s'agit donc, en sorte, d'écrire une fonction génératrice qui en appelle une autre - ici elle même.

Première idée

Première idée naïve pour faire cela, mais qui ne marche pas :

```
[4]: def divdivs(n):  
    for i in divs(n):  
        divs(i)
```

```
[5]: try:
      for i in divdivs(28):
          print(i)
      except Exception as e:
          print(f"OOPS {e}")
```

OOPS 'NoneType' object is not iterable

Ce qui se passe ici, c'est que `divdivs` est perçue comme une fonction normale, lorsqu'on l'appelle elle ne retourne rien, donc `None`; et c'est sur ce `None` qu'on essaie de faire la boucle `for` (à l'intérieur du `try`), qui donc échoue.

Deuxième idée

Si on utilise juste `yield`, ça ne fait pas du tout ce qu'on veut :

```
[6]: def divdivs(n):
      for i in divs(n):
          yield divs(i)
```

```
[7]: try:
      for i in divdivs(28):
          print(i)
      except Exception as e:
          print(f"OOPS {e}")
```

```
<generator object divs at 0x103f668b8>
<generator object divs at 0x103f66930>
<generator object divs at 0x103f668b8>
<generator object divs at 0x103f66930>
```

En effet, c'est logique, chaque `yield` dans `divdivs()` correspond à une itération de la boucle. Bref, il nous manque quelque chose dans le langage pour arriver à faire ce qu'on veut.

yield from

La construction du langage qui permet de faire ceci s'appelle **yield from**;

```
[8]: def divdivs(n):
      for i in divs(n):
          yield from divs(i, verbose=True)
```

```
[9]: try:
      for i in divdivs(28):
          print(i)
      except Exception as e:
          print(f"OOPS {e}")
```

```
trouvé diviseur 2 de 4
2
trouvé diviseur 2 de 14
2
trouvé diviseur 7 de 14
7
```

Avec `yield from`, on peut indiquer que `divdivs` est une fonction génératrice, et qu'il faut évaluer `divs(..)` comme un générateur; ici l'interpréteur va empiler un second appel à `divdivs`, et énumérer tous les résultats que cette fonction va énumérer avec `yield`.

5.13 w5-s4-x1-produit-scalaire

Les boucles **for**

5.13.1 Exercice - niveau intermédiaire

Produit scalaire

```
[1]: # 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.

NOTE remarquez bien qu'on a dit expression génératrice et pas nécessairement fonction génératrice.

```
[2]: # un petit exemple
      exo_produit_scalaire.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

Vous devez donc écrire :

```
[3]: def produit_scalaire(X, Y):
      """retourne le produit scalaire de deux listes de même taille"""
      "<votre_code>"
```

```
[ ]: # pour vérifier votre code
      exo_produit_scalaire.correction(produit_scalaire)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

5.14 w5-s6-c1-import-once

Précisions sur l'importation

5.14.1 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 :

```
[1]: import multiple_import
```

chargement de multiple_import

Ce module est très simple, comme vous pouvez le voir

```
[2]: from modtools import show_module
      show_module(multiple_import)
```

Fichier /Users/tparment/git/flotpython-course/modules/multiple_import.py

```
-----
1|"""
2|Ce module est conçu pour illustrer le mécanisme de
3|chargement / rechargement
4|"""
5|
6|print("chargement de", __name__)
```

Si on le charge une deuxième fois (peu importe où, dans le même module, un autre module, une fonction..), vous remarquerez qu'il ne produit aucune impression

```
[3]: 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`

```
[4]: # la première fois le chargement a vraiment lieu
      import this
```

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
```


Complex is better than complicated.
 Flat is better than nested.
 Sparse is better than dense.
 Readability counts.
 Special cases aren't special enough to break the rules.
 Although practicality beats purity.
 Errors should never pass silently.
 Unless explicitly silenced.
 In the face of ambiguity, refuse the temptation to guess.
 There should be one-- and preferably only one --obvious way to do it.
 Although that way may not be obvious at first unless you're Dutch.
 Now is better than never.
 Although never is often better than **right** now.
 If the implementation is hard to explain, it's a bad idea.
 If the implementation is easy to explain, it may be a good idea.
 Namespaces are one honking great idea -- let's do more of those!

```
[5]: # 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 :

```
[6]: from importlib import reload
reload(multiple_import)
```

chargement de `multiple_import`

```
[6]: <module 'multiple_import' from '/Users/tparment/git/flotpython-course/modules/multiple_import.py'>
```

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 exclusivement 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 :

```
[7]: # charger le magic 'autoreload'
      %load_ext autoreload
```

The autoreload extension is already loaded. To reload it, use:

```
%reload_ext autoreload
```

```
[8]: # 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.

5.14.2 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.

```
[9]: import sys
      'csv' in sys.modules
```

[9]: False

```
[10]: import csv
       'csv' in sys.modules
```

[10]: True

```
[11]: csv is sys.modules['csv']
```

[11]: True

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`.

```
[12]: del sys.modules['multiple_import']  
import multiple_import
```

chargement de `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.

```
[13]: 'gc' in sys.builtin_module_names
```

```
[13]: True
```

Pour en savoir plus

Pour aller plus loin, vous pouvez lire [la documentation sur l'instruction import](#)

5.15 w5-s6-c2-modules-et-chemins

Où sont cherchés les modules ?

5.15.1 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 se trouve le point d'entrée. Pour rappel, le point d'entrée c'est le nom du fichier que vous passez à l'interpréteur lorsque vous démarrez votre programme.

Lorsque vous lancez l'interpréteur en mode interactif (sans lui donner de point d'entrée), c'est le répertoire courant qui sert alors d'emplacement par défaut pour votre code. Le répertoire courant, c'est celui où vous vous trouvez quand vous lancez la commande `python`. Si vous n'êtes pas sûr de cet emplacement vous pouvez le savoir en faisant :

```
[1]: from pathlib import Path  
Path.cwd()
```

```
[1]: PosixPath('/Users/tparment/git/flotpython-tools/pdf/work')
```

5.15.2 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 un dossier `spam/` s'il s'agit d'un package, éventuellement assorti d'un `__init__.py`); 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
from pathlib import Path

def show_argv_and_path():
    print(f"le répertoire courant est {Path.cwd()}")
    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 ceci dans le fichier `/le/repertoire/du/script/run.py`
- et que vous lancez Python depuis un répertoire différent, disons `/le/repertoire/ou/vous/etes`
- et avec une variable `PYTHONPATH` vide;

```
$ cd /le/repertoire/ou/vous/etes/
/le/repertoire/ou/vous/etes
$ python3 /le/repertoire/du/script/run.py
```

alors vous devriez observer une sortie sur le terminal comme ceci :

```
le répertoire courant est /le/repertoire/ou/vous/etes
le point d'entrée du programme est /le/repertoire/du/script/run.py
la variable sys.path contient
1-ème chemin dans sys.path /le/repertoire/du/script
... <snip> ... le reste dépend de votre installation*
```

C'est-à-dire que :

- la variable `sys.argv[0]` contient - en tous cas ici - le chemin complet `/le/repertoire/du/script/run.py`,
- et le premier terme dans `sys.path` contient `/le/repertoire/du/script/`.

(NB que d'après [cette documentation](#) `sys.argv[0]` peut contenir un chemin complet ou un simple nom de fichier, selon votre OS et comment vous invoquez Python)

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  
from pathlib import Path  
  
directory_installation = Path(__file__).parent  
  
# 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 supprime `distutils` qui lui, fait partie de la distribution standard mais qui est tombé en désuétude 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
```

5.16

w5-s7-c1-import-as

La clause `import as`

5.16.1 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 contrôlées 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 :

```
[1]: # 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 :

```
[2]: from importlib import import_module
```

```
[3]: loaded = import_module(modulename)
     type(loaded)
```

```
[3]: module
```

Nous avons maintenant bien chargé le module `math`, et on l'a rangé dans la variable `loaded`

```
[4]: # loaded référence le même objet module que si on avait fait
     # import math
     import math
     math is loaded
```

```
[4]: True
```

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 :

```
[5]: # un import simple
     import math
```

```
[6]: # peut se récrire
     math = import_module('math')
```

Et :

```
[7]: # et un import from
     from pathlib import Path
```

```
[8]: # 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 écrivez un module dans un fichier `globals.py` et vous l'importez dans votre code

```
import globals
```

Puis un moment après pour déboguer vous voulez utiliser la fonction built-in `globals`. Sauf que, en vertu de la règle de visibilité des variables (rappelez-vous de la règle “LEGB”, que l'on a vue dans une vidéo de la Semaine 4), 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 jouets :

- `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.

```
[9]: # changer le nom du module importé
import un_deux as one_two
one_two.un()
```

la fonction `un` dans le module `un_deux`


```
[10]: # changer le nom d'un symbole importé du module
      from un_deux_trois import un as one
      one()
```

la fonction un dans le module un_deux_trois

```
[11]: # on peut mélanger tout ça
      from un_deux_trois_quatre import un as one, deux, trois as three
```

```
[12]: one()
      deux()
      three()
```

la fonction un dans le module un_deux_trois_quatre
la fonction deux dans le module un_deux_trois_quatre
la fonction trois dans le module un_deux_trois_quatre

Pour en savoir plus

Vous pouvez vous reporter à [la section sur l'instruction import](#) dans la documentation python.

5.17 w5-s7-c2-recapitulatif-import

Récapitulatif sur **import**

5.17.1 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

```
[1]: 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 :

```
[2]: # la fonction elle-même
      print(un_deux.un)

      un_deux.un()
```

```
<function un at 0x10f26b950>
la fonction un dans le module un_deux
```

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` :

```
[3]: # dans l'espace de nommage courant on peut accéder au module lui-même
print(un_deux)
```

```
<module 'un_deux' from '/Users/tparment/git/flotpython-course/modules/un_deux.py'>
```

```
[4]: # mais pas à la variable `un`
try:
    print(un)
except NameError:
    print("La variable 'un' n'est pas définie")
```

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) :

```
[5]: from un_deux_trois import un, deux
```

À présent nous avons deux nouvelles variables dans la portée locale :

```
[6]: un()
      deux()
```

la fonction un dans le module un_deux_trois
la fonction deux dans le module un_deux_trois

Et cette fois, c'est le module lui-même qui n'est pas accessible :

```
[7]: try:
      print(un_deux_trois)
    except NameError:
        print("La variable 'un_deux_trois' n'est pas définie")
```

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 :

```
[8]: import un_deux_trois
```

alors nous avons maintenant deux variables différentes qui désignent la fonction un dans le module :

```
[9]: print(un_deux_trois.un)
      print(un)
      print("ce sont deux façons d'accéder au même objet", un is un_deux_trois.un)
```

```
<function un at 0x10f26b9d8>
<function un at 0x10f26b9d8>
ce sont deux façons d'accéder au même objet True
```

En on peut modifier l'une sans affecter l'autre :

```
[10]: # 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)
```

```
<function un at 0x10f26b9d8>
1
```

5.17.2 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.

```
[11]: # par exemple
      import un_deux as mod12
      mod12.un()
```

la fonction `un` dans le module `un_deux`

```
[12]: from un_deux import deux as m12deux
      m12deux()
```

la fonction `deux` dans le module `un_deux`

```
import *
```

La dernière forme d'import consiste à importer toutes les variables d'un module comme ceci :

```
[13]: from un_deux_trois_quatre import *
```

Cette forme, pratique en apparence, va donc créer dans l'espace de nommage courant les variables

```
[14]: un()
      deux()
```

```
trois()
quatre()
```

```
la fonction un dans le module un_deux_trois_quatre
la fonction deux dans le module un_deux_trois_quatre
la fonction trois dans le module un_deux_trois_quatre
la fonction quatre dans le module 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...

5.17.3 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 :

```
[15]: # 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
```

[15]: True

Imports relatifs

Il existe aussi en python une façon d'importer des modules, non pas directement en cherchant depuis `sys.path`, mais en cherchant à partir du module où se trouve la clause `import`. Nous détaillons ce trait dans un complément ultérieur.

5.18 w5-s7-c3-packages

La notion de package

5.18.1 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 :

```
[1]: from modtools import show_module
```

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" :

```
[2]: import module_simple
```

Chargement du module `module_simple`

Voyons à quoi ressemble ce module :

```
[3]: show_module(module_simple)
```

Fichier /Users/tparment/git/flotpython-course/modules/module_simple.py

```
-----
1|print("Chargement du module", __name__)
2|
3|def spam(n):
4|    "Le polynôme (n+1)*(n-3)"
5|    return n**2 - 2*n - 3
```

On a bien compris maintenant que le module joue le rôle d'espace de nom, dans le sens où :

```
[4]: # on peut définir sans risque une variable globale 'spam'
spam = 'eggs'
print("spam globale", spam)
```

spam globale eggs

```
[5]: # qui est indépendante de celle définie dans le module
print("spam du module", module_simple.spam)
```

spam du module <function spam at 0x10a59af28>

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 le module est aux fichiers.

Nous allons illustrer ceci en créant un package qui contient un module. Pour cela nous créons une arborescence de fichiers comme ceci :

```
package_jouet/
    __init__.py
    module_jouet.py
```

On importe un package exactement comme un module :

```
[6]: import package_jouet
```

```
chargement du package package_jouet
Chargement du module package_jouet.module_jouet dans le package 'package_jo
uet'
```

Voici le contenu de ces deux fichiers :

```
[7]: show_module(package_jouet)
```

```
Fichier /Users/tparment/git/flotpython-course/modules/package_jouet/__init__
_.PY
-----
1|print("chargement du package", __name__)
2|
3|spam = ['a', 'b', 'c']
4|
5|# on peut forcer l'import de modules
6|import package_jouet.module_jouet
7|
8|# et définir des raccourcis
9|jouet = package_jouet.module_jouet.jouet
```

```
[8]: show_module(package_jouet.module_jouet)
```

```
Fichier /Users/tparment/git/flotpython-course/modules/package_jouet/module_
jouet.py
-----
1|print("Chargement du module", __name__, "dans le package 'package_jouet'"
    )
2|
3|jouet = 'une variable définie dans package_jouet.module_jouet'
```

Comme on le voit, le package porte le même nom que le répertoire, c'est-à-dire que, de même que le module `module_simple` correspond au fichier `module_simple.py`, le package python `package_jouet` correspond au répertoire `package_jouet`.

Note historique par le passé, pour définir un package, il fallait obligatoirement créer dans le répertoire (celui, donc, que l'on veut exposer à python), un fichier nommé `__init__.py`; ce n'est plus le cas depuis Python-3.3.

Comme on le voit, importer un package revient essentiellement à charger, lorsqu'il existe, le fichier `__init__.py` dans le répertoire correspondant (et sinon, on obtient un package vide).

On a coutume de faire la différence entre package et module, mais en termes d'implémentation les deux objets sont en fait de même nature, ce sont des modules :

```
[9]: type(package_jouet)
```

```
[9]: module
```

```
[10]: type(package_jouet.module_jouet)
```

```
[10]: module
```

Ainsi, 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 :

```
[11]: package_jouet.spam
```

```
[11]: ['a', 'b', 'c']
```

L'espace de noms du package permet de référencer les packages ou modules qu'il contient, comme on l'a vu ci-dessus, le package référence le module au travers de son attribut `module_jouet` :

```
[12]: package_jouet.module_jouet
```

```
[12]: <module 'package_jouet.module_jouet' from '/Users/tparment/git/flotpython-course/modules/package_jouet/module_jouet.py'>
```

À quoi sert `__init__.py` ?

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 correspond 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

```
[13]: package_jouet.jouet
```

```
[13]: 'une variable définie dans package_jouet.module_jouet'
```

alors qu'en fait il faudrait écrire en toute rigueur

```
[14]: package_jouet.module_jouet.jouet
```

```
[14]: 'une variable définie dans package_jouet.module_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.

De manière générale, `__init__.py` peut contenir n'importe quel code Python chargé d'initialiser le package. Notez que depuis Python-3.3, la présence de `__init__.py` n'est plus strictement nécessaire**.

Lorsqu'il est présent, comme pour les modules usuels, `__init__.py` 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.

Pour en savoir plus

Voir la [section sur les modules](#) dans la documentation python, et notamment la [section sur les packages](#).

5.19 w5-s7-c4-import-advanced

Usage avancés de **import**

5.19.1 Complément - niveau avancé

```
[1]: # notre utilitaire pour afficher le code des modules
from modtools import show_module, find_on_disk
```

Attributs spéciaux

Les objets de type module possèdent des attributs spéciaux; on les reconnaît facilement car leur nom est en `__truc__`, c'est une convention générale dans tous le langage : on en a déjà vu plusieurs exemples avec par exemple les méthodes `__iter__()`.

Voici pour commencer les attributs spéciaux les plus utilisées; pour cela nous reprenons le package d'un notebook précédent :

```
[2]: import package_jouet
```

```
chargement du package package_jouet
Chargement du module package_jouet.module_jouet dans le package 'package_jo
uet'
```

```
__name__
```

Le nom canonique du module :

```
[3]: package_jouet.__name__
```

```
[3]: 'package_jouet'
```

```
[4]: package_jouet.module_jouet.__name__
```

```
[4]: 'package_jouet.module_jouet'
```

```
__file__
```

L'emplacement du fichier duquel a été chargé le module; pour un package ceci dénote un fichier `__init__.py` :

```
[5]: package_jouet.__file__
```

```
[5]: '/Users/tparment/git/flotpython-course/modules/package_jouet/__init__.py'
```

```
[6]: package_jouet.module_jouet.__file__
```

```
[6]: '/Users/tparment/git/flotpython-course/modules/package_jouet/module_jouet.py'
```

```
__all__
```

Il est possible de redéfinir dans un module la variable `__all__`, de façon à définir les symboles qui sont réellement concernés par un `import *`, comme c'est décrit ici.

Je rappelle toutefois que l'usage de `import *` est fortement déconseillé dans du code de production.

Import absolu

La mécanique des imports telle qu'on l'a vue jusqu'ici est ce qui s'appelle un import absolu qui est depuis python-2.5 le mécanisme par défaut : le module importé est systématiquement cherché à partir de `sys.path`.

Dans ce mode de fonctionnement, si on trouve dans le même répertoire deux fichiers `foo.py` et `bar.py`, et que dans le premier on fait :

```
import bar
```

eh bien alors, malgré le fait qu'il existe ici même un fichier `bar.py`, l'import ne réussit pas (sauf si le répertoire courant est dans `sys.path` ; en général ce n'est pas le cas).

Import relatif

Ce mécanisme d'import absolu a l'avantage d'éviter qu'un module local, par exemple `random.py`, ne vienne cacher le module `random` de la bibliothèque standard. Mais comment peut-on faire alors pour charger le module `random.py` local ? C'est à cela que sert l'import relatif.

Voyons cela sur un exemple qui repose sur la hiérarchie suivante :

```
package_relatif/
    __init__.py  (vide)
    main.py
    random.py
```

Le fichier `__init__.py` ici est vide, et voici le code des deux autres modules :

```
[7]: import package_relatif
```

```
[8]: # le code de main.py
code = find_on_disk(package_relatif, "main.py")
!cat $code
```

```
# pour importer un module entier en mode relatif
from . import random as local_random_module
```

```
# la syntaxe pour importer seulement un symbole
```

```
from .random import alea
```

```
print(
    f"""On charge main.py
    __name__={__name__}
    alea={alea()}"""
```

Nous avons illustré dans le point d'entrée `main.py` deux exemples d'import relatif :

Les deux clauses `as` sont bien sûr optionnelles, on les utilise ici uniquement pour bien identifier les différents objets en jeu.

Le module local `random.py` expose une fonction `alea` qui génère un string aléatoire en se basant sur le module standard `random` :

```
[9]: # le code de random.py
code = find_on_disk(package_relatif, "random.py")
!cat $code
```

```
import random

print(f"On charge le module random local {__name__}")

def alea():
    return(f"[[{random.randint(0, 10)}]]")
```

Cet exemple montre comment on peut importer un module local de nom `random` et le module `random` qui provient de la librairie standard :

```
[10]: import package_relatif.main
```

```
On charge le module random local package_relatif.random
On charge main.py
__name__=package_relatif.main
alea=[[9]]
```

```
[11]: print(package_relatif.main.alea())
```

```
[[0]]
```

Pour remonter dans l'arborescence

Il faut savoir également qu'on peut "remonter" dans l'arborescence de fichiers en utilisant plusieurs points `.` consécutifs. Voici un exemple fonctionnel, on part du même contenu que ci-dessus avec un sous-package, comme ceci :

```
package_relatif/
    __init__.py      (vide)
    main.py
    random.py
    subpackage/
        __init__.py  (vide)
        submodule.py
```

```
[12]: # voyons le code de submodule:
import package_relatif.subpackage
```

```
[13]: # le code de submodule/submodule.py
code = find_on_disk(package_relatif.subpackage, "submodule.py")
!cat $code
```

```
# notez ici la présence des deux points pour remonter
from ..random import alea as imported
```

```
print(f"On charge {__name__}")
```

```
def alea():
    return f"<<{imported()}>>"
```

```
[14]: import package_relatif.subpackage.submodule
```

```
On charge package_relatif.subpackage.submodule
```

```
[15]: print(package_relatif.subpackage.submodule.alea())
```

```
<<[[2]]>>
```

Ce qu'il faut retenir

Sur cet exemple, on montre comment un import relatif permet à un module d'importer un module local qui a le même nom qu'un module standard.

Avantages de l'import relatif

Bien sûr ici on aurait pu faire

```
import package_relatif.random
```

au lieu de

```
from . import random
```

Mais l'import relatif présente notamment l'avantage d'être insensible aux renommages divers à l'intérieur d'une bibliothèque.

Dit autrement, lorsque deux modules sont situés dans le même répertoire, il semble naturel que l'import entre eux se fasse par un import relatif, plutôt que de devoir répéter ad nauseam le nom de la bibliothèque - ici `package_relatif` - dans tous les imports.

Frustrations liées à l'import relatif

Se base sur `__name__` et non sur `__file__`. Toutefois, l'import relatif ne fonctionne pas toujours comme on pourrait s'y attendre. Le point important à garder en tête est que lors d'un import relatif, c'est l'attribut `__name__` qui sert à déterminer le point de départ.

Concrètement, lorsque dans `main.py` on fait :

```
from . import random
```

l'interpréteur :

- détermine que dans `main.py`, `__name__` vaut `package_relatif.main`;
- il “oublie” le dernier morceau `main` pour calculer que le package courant est `package_relatif`
- et c'est ce nom qui sert à déterminer le point de départ de l'import relatif.

Aussi cet import est-il retranscrit en

```
from package_relatif import random
```

De la même manière

```
from .random import run
```

devient

```
from package_relatif.random import run
```

Par contre l'attribut `__file__` n'est pas utilisé : ce n'est pas parce que deux fichiers python sont dans le même répertoire que l'import relatif va toujours fonctionner. Avant de voir cela sur un exemple, il nous faut revenir sur l'attribut `__name__`.

Digression sur l'attribut `__name__` Il faut savoir en effet que le point d'entrée du programme - c'est-à-dire le fichier qui est passé directement à l'interpréteur python - est considéré comme un module dont l'attribut `__name__` vaut la chaîne `"__main__"`.

Concrètement, si vous faites

```
python3 tests/montest.py
```

alors la valeur observée dans l'attribut `__name__` n'est pas `"tests.montest"`, mais la constante `"__main__"`.

C'est pourquoi d'ailleurs (et c'est également expliqué ici) vous trouverez parfois à la fin d'un fichier source une phrase comme celle-ci :

```
if __name__ == "__main__":  
    <faire vraiment quelque chose>  
    <comme par exemple tester le module>
```

Cet idiome très répandu permet d'insérer à la fin d'un module du code - souvent un code de test - qui :

- va être exécuté quand on le passe directement à l'interpréteur python, mais
- qui n'est pas exécuté lorsqu'on importe le module.

L'attribut `__package__` Pour résumer :

- le point d'entrée - celui qui est donné à `python` sur la ligne de commande - voit comme valeur pour `__name__` la constante `"__main__"`,
- et le mécanisme d'import relatif se base sur `__name__` pour localiser les modules importés.

Du coup, par construction, il n'est quasiment pas possible d'utiliser les imports relatifs à partir du script de lancement.

Pour pallier à ce type d'inconvénients, il a été introduit ultérieurement (voir PEP 366 ci-dessous) la possibilité pour un module de définir (écrire) l'attribut `__package__`, pour contourner cette difficulté.

Ce qu'il faut retenir On voit que tout ceci est rapidement assez scabreux. Cela explique sans doute l'usage relativement peu répandu des imports relatifs.

De manière générale, une bonne pratique consiste à :

- considérer votre ou vos points d'entrée comme des accessoires ; un point d'entrée typiquement se contente d'importer une classe d'un module, de créer une instance et de lui envoyer une méthode ;
- toujours placer ces points d'entrée dans un répertoire séparé ;
- notamment si vous utilisez `setuptools` pour distribuer votre application via `pypi.org`, vous verrez que ces points d'entrée sont complètement pris en charge par les outils d'installation.

S'agissant des tests :

- la technique qu'on a vue rapidement - de tester si `__name__` vaut `"__main__"` - est extrêmement basique et limitée. Le mieux est de ne pas l'utiliser en fait, en dehors de micro-maquettes.
- en pratique on écrit les tests dans un répertoire séparé - souvent appelé `tests` - et en tirant profit de la librairie `unittest`.
- du coup les tests sont toujours exécutés avec une phrase comme

```
python3 -m unittest tests.jeu_de_tests
```

et dans ce contexte-là, il est possible par exemple pour les tests de recourir à l'import relatif.

Pour en savoir plus

Vous pourrez consulter :

- <https://www.python.org/dev/peps/pep-0328/> qui date du passage de 2.4 à 2.5, dans lequel on décide que tous les imports sans `.` sont absolus - ce n'était pas le cas au préalable.
- <https://www.python.org/dev/peps/pep-0366/> qui introduit la possibilité de définir `__package__` pour contourner les problèmes liés aux imports relatifs dans un script.
- <http://sametmax.com/un-gros-guide-bien-gras-sur-les-tests-unitaires-en-python-partie-1/> qui parle des tests unitaires qui est un tout autre et vaste sujet.

5.20 w5-s7-x1-decode-zen

Décoder le module **this**

5.20.1 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

```
[1]: import this
```

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Il suit du cours qu'une fois cet import effectué nous avons accès à une variable **this**, de type module :

```
[2]: this
```

```
[2]: <module 'this' from '/Users/tparment/miniconda3/envs/flotpython-course/lib/
python3.7/this.py'>
```

But de l'exercice

```
[3]: # 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 :

```
[4]: # on rappelle que dir() renvoie les noms des attributs  
# accessibles à partir de l'objet  
dir(this)
```

```
[4]: ['__builtins__',  
      '__cached__',  
      '__doc__',  
      '__file__',  
      '__loader__',  
      '__name__',  
      '__package__',  
      '__spec__',  
      'c',  
      'd',  
      'i',  
      's']
```

Vous pouvez ignorer `this.c` et `this.i`, les deux autres variables du module sont importantes pour nous.

```
[5]: # 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 :

```
[6]: # ces deux quantités sont égales  
len(this.s) == len(resultat)
```

```
[6]: True
```

À quoi peut bien servir l'autre variable ?

```
[7]: # se pourrait-il que d agisse comme un code simple ?  
this.d[this.s[0]] == resultat[0]
```

```
[7]: True
```

Le texte comporte certes des caractères alphabétiques

```
[8]: # si on ignore les accents,  
# il y a 26 caractères minuscules  
# et 26 caractères majuscules  
len(this.d)
```

```
[8]: 52
```

mais pas seulement ; les autres sont préservés.

À vous de jouer

```
[9]: def decode_zen(this):  
      "<votre code>"
```

Correction

```
[ ]: exo_decode_zen.correction(decode_zen)  
  
# NOTE  
# auto-exec-for-latex has skipped execution of this cell
```


Chapitre 6

Conception des classes

6.1 w6-s1-cl-introduction-classes

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 built-in 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.

```
[1]: 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

```
[2]: help(Matrix2)
```

Help on class Matrix2 in module __main__:

```
class Matrix2(builtins.object)
|   Matrix2(a11, a12, a21, a22)
|
|   Une implémentation sommaire de matrice carrée 2x2
|
|   Methods defined here:
|
|   __init__(self, a11, a12, a21, a22)
|       construit une matrice à partir des 4 coefficients
|
|   determinant(self)
|       renvoie le déterminant de la matrice
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
```

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.

```
[3]: matrice = Matrix2(1, 2, 2, 1)
      print(matrice)
```

```
<__main__.Matrix2 object at 0x10fcb8c50>
```

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 :

```
[4]: matrice.determinant()
```

```
[4]: -3
```

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 :

```
[5]: Matrix2.determinant(matrice)
```

```
[5]: -3
```

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 a 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 dynamique 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.

```
[6]: 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 :

```
[7]: matrice = Matrix2(1, 2, 2, 1)
      print("Determinant =", matrice.determinant())
```

Determinant = -3

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` :

```
[8]: print(matrice)
```

<<mat-2x2 (1, 2, 2, 1)>>

Ce format d'impression reste d'ailleurs valable dans l'impression d'objets plus compliqués, comme par exemple :

```
[9]: # 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}")
```

composite=[<<mat-2x2 (1, 2, 2, 1)>>, None, <<mat-2x2 (1, 0, 0, 1)>>]

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 w6-s1-c2-record-et-classe

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 avons 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 avons alors écrit quelque chose comme :

```
[1]: pierre = {'nom': 'pierre', 'age': 25, 'email': 'pierre@foo.com'}
      print(pierre)
```

```
{'nom': 'pierre', 'age': 25, 'email': 'pierre@foo.com'}
```

Cette fois-ci nous allons implémenter la même abstraction, mais avec une classe **Personne** comme ceci :

```
[2]: 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 de rappel sur le passage d'arguments aux fonctions :


```
[3]: 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)
```

```
<<pierre, 25 ans, email:pierre@foo.com>>
<<paul, 18 ans, email:paul@bar.com>>
<<jacques, 52 ans, email:jacques@cool.com>>
```

Un dictionnaire pour indexer les enregistrements

Nous pouvons appliquer exactement la même technique d'indexation qu'avec les dictionnaires :

```
[4]: # 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 :

```
[5]: pierre = index_par_nom['pierre']
print(pierre)
```

```
<<pierre, 25 ans, email:pierre@foo.com>>
```

Encapsulation

Pour marquer l'anniversaire d'une personne, nous pourrions faire :

```
[6]: pierre.age += 1
pierre
```

```
[6]: <<pierre, 26 ans, email:pierre@foo.com>>
```

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 propriétés, 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'instruction `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 :

```
[7]: # 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 :

```
[8]: help(Personne)
```

Help on class Personne in module __main__:

```
class Personne(builtins.object)
|   Personne(nom, age, email)
|
|   Une personne possède un nom, un âge et une adresse e-mail
|
|   Methods defined here:
|
|   __init__(self, nom, age, email)
|       Initialize self.  See help(type(self)) for accurate signature.
|
|   __repr__(self)
|       Return repr(self).
|
|   sendmail(self, subject, body)
|       Envoie un mail à la personne
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
```

```
| list of weak references to the object (if defined)
```

Et on peut à présent utiliser cette méthode :

```
[9]: pierre.sendmail("Coucou", "Salut ça va ?")
```

```
To: pierre@foo.com
Subject: Coucou
Body: Salut ça va ?
```

6.3 w6-s1-c3-property

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 proposons 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 propriétés :

```
[1]: # 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"
```

```
[2]: # créons une instance
t1 = Temperature1(20)
t1
```

```
[2]: 20K
```

```
[3]: # et pour accéder à la valeur numérique je peux faire
t1.kelvin
```

[3]: 20

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 :

```
[4]: 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 côté, c'est mieux parce que je garantis un invariant, la température est toujours positive :

```
[5]: t2 = Temperature2(-30)
t2
```

[5]: OK

Mais par contre, d'un autre côté, c'est très lourd, parce que chaque fois que je veux utiliser mon objet, je dois faire pour y accéder :

```
[6]: t2.get_kelvin()
```

[6]: 0

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 :

```
[7]: 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"
```

```
[8]: t3 = Temperature3(200)
t3
```

[8]: 200K

```
[9]: # par contre ici on va le mettre à zéro
# à nouveau, une exception serait préférable sans doute
t3.kelvin = -30
t3
```

[9]: OK

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'utilisation, 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`.

```
[10]: 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 propriétés 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 propriété qui définit l'attribut `kelvin`
    kelvin = property(_get_kelvin, _set_kelvin)

    # les deux autres propriétés font la conversion, puis
    # sous-traitent à la propriété 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 :

```
[11]: t = Temperature(celsius=0)
      t
```

```
[11]: <273.16K == 0°C == 32°F>
```

```
[12]: t.fahrenheit
```

```
[12]: 32.0
```

```
[13]: t.celsius += 100
      print(t)
```

```
373.16K
```

```
[14]: try:
      t = Temperature(fahrenheit = -1000)
    except Exception as e:
      print(f"OOPS, {type(e)}, {e}")
```

```
OOPS, <class 'ValueError'>, Kelvin -300.1733333333333 must be positive
```

Pour en savoir plus

Voir aussi [la documentation officielle](#).

Vous pouvez notamment aussi, en option, ajouter un `del` pour intercepter les instructions du type :

```
[15]: # comme on n'a pas défini de delete, on ne peut pas faire ceci
      try:
        del t.kelvin
      except Exception as e:
        print(f"OOPS {type(e)} {e}")
```

```
OOPS <class 'AttributeError'> can't delete attribute
```

6.4 w6-s1-c4-datetime

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 :

```
[1]: 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
```

```
[1]: 1590418620.2493172
```

```
[2]: # 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 :

```
[3]: type(t_later)
```

```
[3]: float
```

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) :

```
[4]: struct_later = time.gmtime(t_later)
print(struct_later)
```

```
time.struct_time(tm_year=2020, tm_mon=5, tm_mday=25, tm_hour=17, tm_min=57,
                  tm_sec=0, tm_wday=0, tm_yday=146, tm_isdst=0)
```

Et on met en forme ce résultat en utilisant des méthodes comme, par exemple, `strftime()` pour afficher l'heure UTC dans 3 heures :

```
[5]: print(f'heure UTC dans trois heures '
          f'{time.strftime("%Y-%m-%d at %H:%M", struct_later)}')
```

```
heure UTC dans trois heures 2020-05-25 at 17:57
```

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` :

```
[6]: 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 :

```
[7]: # on peut imprimer simplement un objet date_time
print(f'maintenant {dt_now}')
```

maintenant 2020-05-25 16:57:00.320770

```
[8]: # 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")}')
```

dans trois heures 2020-05-25 at 19:57

```
[9]: # 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}')
```

dans trois heures 2020-05-25 at 19:57

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 :

```
[10]: 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 :

```
[11]: 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 :

```
[12]: vars(bateau2)
```

```
[12]: {'id': 1000, 'name': 'Toccata', 'country': 'FRA'}
```

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 :

```
[13]: # du côté 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.

6.5 w6-s2-c1-instance-hashable

Manipuler des ensembles d'instances

6.5.1 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 avons 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.

Instance mutable dans un ensemble

Une instance de classe est par défaut un objet mutable. Malgré cela, 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.

Hachage par défaut : basé sur `id()`

```
[1]: # une classe Point
class Point1:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Pt[{self.x}, {self.y}]"
```

Avec ce code, les instances de `Point` sont mutables :

```
[2]: # deux instances
p1 = Point1(2, 2)
p2 = Point1(2, 3)
```

```
[3]: # objets mutables
p1.y = 3
```

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) :

```
[4]: # nos deux objets se ressemblent
p1, p2
```

```
[4]: (Pt[2, 3], Pt[2, 3])
```

```
[5]: # mais peuvent coexister
# dans un ensemble
```

```
# qui a alors 2 éléments
s = { p1, p2 }
len(s)
```

[5]: 2

Si on recherche un de ces deux objets on le trouve :

```
[6]: p1 in s
```

[6]: True

```
[7]: # mais pas un troisième
p3 = Point1(2, 4)
p3 in s
```

[7]: False

Cette possibilité de gérer des ensembles d'objets selon cette stratégie est très commode et peut apporter de gros gains de performance, notamment lorsqu'on a souvent besoin de faire des tests d'appartenance.

En pratique, lorsqu'un modèle de données définit une relation de type "1-n", je vous recommande d'envisager d'utiliser un ensemble plutôt qu'une liste.

Par exemple envisagez :

```
class Animal:
    # blabla

class Zoo:
    def __init__(self):
        self.animals = set()
```

Plutôt que :

```
class Animal:
    # blabla

class Zoo:
    def __init__(self):
        self.animals = []
```

6.5.2 Complément - niveau avancé

Ce n'est pas ce que vous voulez ?

Le comportement qu'on vient de voir pour les instances de `Point1` dans les tables de hachage est raisonnable, si on admet que deux points ne sont égaux que s'ils sont le même objet au sens de `is`.

Mais imaginons que vous voulez au contraire considérer que deux points sont égaux lorsqu'ils coïncident sur le plan. Avec ce modèle de données, vous voudriez que :

- un ensemble dans lequel on insère `p1` et `p2` ne contienne qu'un élément,
- et qu'on trouve `p3` quand on le cherche dans cet ensemble.

Le protocole hashable : `__hash__` et `__eq__`

Le langage nous permet de faire cela, grâce au protocole hashable ; 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 hashes 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 :

```
[8]: 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 :

```
[9]: q1 = Point2(2, 3)
     q2 = Point2(2, 3)
```

Nos deux objets sont distincts pour `id()/is`, mais égaux pour `==` :

```
[10]: print(f"is → {q1 is q2} \n== → {q1 == q2}")
```

```
is → False
== → True
```

Et un ensemble contenant les deux points n'en contient qu'un :

```
[11]: s = {q1, q2}
     len(s)
```

```
[11]: 1
```

```
[12]: q3 = Point2(2, 3)
     q3 in s
```

```
[12]: True
```

Comme les ensembles et les dictionnaires reposent sur le même mécanisme de table de hachage, on peut aussi indifféremment utiliser n'importe lequel de nos 3 points pour indexer un dictionnaire :

```
[13]: d = {}  
      d[q1] = 1  
      d[q2]
```

```
[13]: 1
```

```
[14]: # les clés q1, q2 et q3 sont  
      # les mêmes pour le dictionnaire  
      d[q3] = 10000  
      d
```

```
[14]: {Pt[2, 3]: 10000}
```

Attention !

Tout ceci semble très bien fonctionner ; sauf qu'en fait, il y a une grosse faille, c'est que nos objets `Point2` sont mutables. Du coup on peut maintenant imaginer un scénario comme celui-ci :

```
[15]: t1, t2 = Point2(10, 10), Point2(10, 10)  
      s = {t1, t2}  
      s
```

```
[15]: {Pt[10, 10]}
```

```
[16]: t1 in s, t2 in s
```

```
[16]: (True, True)
```

Mais si maintenant je change un des deux objets :

```
[17]: t1.x = 100
```

```
[18]: s
```

```
[18]: {Pt[100, 10]}
```

```
[19]: t1 in s
```

```
[19]: False
```

```
[20]: t2 in s
```

```
[20]: False
```

Évidemment cela n'est pas correct. Ce qui se passe ici c'est qu'on a

— d'abord inséré `t1` dans `s`, avec un indice de hachage calculé à partir de `10, 10`

- pas inséré `t2` dans `s` parce qu'on a déterminé qu'il existait déjà.

Après avoir modifié `t1` qui est le seul élément de `s` : À ce stade :

- lorsqu'on cherche `t1` dans `s`, on le fait avec un indice de hachage calculé à partir de 100, 10 et du coup on ne le trouve pas,
- lorsqu'on cherche `t2` dans `s`, on utilise le bon indice de hachage, mais ensuite le seul élément qui pourrait faire l'affaire n'est pas égal à `t2`.

Conclusion

La [documentation de Python sur ce sujet](#) indique ceci :

If a class defines mutable objects and implements an `__eq__()` method, it should not implement `__hash__()`, since the implementation of hashable collections requires that a key's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

Notre classe `Point2` illustre bien cette limitation. Pour qu'elle soit utilisable en pratique, il faut rendre ses instances immutables. Cela peut se faire de plusieurs façons, dont deux que nous aborderons dans la prochaine séquence et qui sont - entre autres :

- le `namedtuple`
- et la `dataclass` (nouveau en 3.7).

6.6 w6-s2-c2-speciales-1

Surcharge d'opérateurs (1)

6.6.1 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 expliqués 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 :

```
[1]: class Foo:
      def __repr__(self):
          return 'custom repr'

foo = Foo()
# lorsque vous affichez un objet comme ceci
foo
# en fait vous utilisez repr()
```

```
[1]: custom 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 :

```
[2]: # 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
```

avec print Point(0,100)

[2]: Point(0,100)

```
[3]: # 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
```

avec print (0,100)
avec format (0,100)

[3]: Point2(0,100)

`__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 avons 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.

```
[4]: # 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 à 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 :

```

[5]: 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")

```

```

avant matrice1 [1, 2, 3, 4]
avant matrice2 [10, 20, 30, 40]
somme [11, 22, 33, 44]
après matrice1 [1, 2, 3, 4]

```

```
après matrice2 [10, 20, 30, 40]
[1, 2, 3, 4] n'est pas nulle
[0, 0, 0, 0] 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 built-in `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 une liste de matrices puisque maintenant on sait les additionner, (mais on a dû toutefois passer à `sum` comme élément neutre `zero`) :

```
[6]: sum([matrice1, matrice2, matrice1] , zero)
```

```
[6]: [12, 24, 36, 48]
```

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, s'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.

6.6.2 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.

```
[7]: # remarquez que les opérandes sont apparemment inversés
# dans le sens où pour évaluer
#     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
```

```
[8]: matrice1
```

```
[8]: [1, 2, 3, 4]
```

```
[9]: 12 * matrice1
```

```
[9]: [12, 24, 36, 48]
```

6.7 w6-s2-c3-speciales-2

Méthodes spéciales (2/3)

6.7.1 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.

```
[1]: 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 sortie
        """
        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()
```

```
[2]: # 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)
```

```
<DualQueue, inputs=['trois', 'deux'], outputs=['un', 'zero']>
```

Longueur et appartenance

Avec cette première version de la classe `DualQueue` on peut utiliser `len` et le test d'appartenance :

```
[3]: print(f'len() = {len(queue)}')

print(f"deux appartient-il ? {'deux' in queue}")
print(f"1 appartient-il ? {1 in queue}")
```

```
len() = 4
deux appartient-il ? True
1 appartient-il ? False
```

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 :

```
[4]: # 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 :

```
[5]: print(queue[0])
```

zero

ce qui lève la même exception qu'avec une vraie liste si on utilise un mauvais index :

```
[6]: try:
      print(queue[5])
    except IndexError as e:
      print('ERREUR',e)
```

ERREUR Mauvais indice 5 pour DualQueue

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 :

```
[7]: # 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 :

```
[8]: queue[1:3]
```

```
[8]: ['un', 'deux']
```

Et on reçoit bien une exception si on essaie d'accéder par clé :

```
[9]: try:
      queue['key']
    except KeyError as e:
        print(f"OOPS: {type(e).__name__}: {e}")
```

```
OOPS: KeyError: '[] avec type non reconnu key'
```

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 :

```
[10]: # grâce à __getitem__ on a rendu les
      # objets de type DualQueue itérables
      for item in queue:
          print(item)
```

```

zero
un
deux
trois

```

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__` :

```
[11]: # un test fait directement sur la queue
if queue:
    print(f"La queue {queue} est considérée comme True")
```

La queue <DualQueue, inputs=['trois', 'deux'], outputs=['un', 'zero']> est considérée comme True

```
[12]: # 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")
```

La queue <DualQueue, inputs=[], outputs=[]> 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 :

```
[13]: 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)
```

```
[14]: # on peut maintenant utiliser cet objet
      # comme une fonction qui fait sum(*arg)+2
      plus2()
```

```
[14]: 2
```

```
[15]: plus2(1)
```

```
[15]: 3
```

```
[16]: plus2(1, 2)
```

```
[16]: 5
```

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.

6.8 w6-s2-c4-speciales-3

Méthodes spéciales (3/3)

6.8.1 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 :

```
[1]: # 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 :

```

[2]: # 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') ] )

# réserver un noeud
node_lease = rpc_proxy.BookNode (
    { 'id' : 1002, 'phy_mem' : '32G' } )

```

Envoi à `http://cloud.provider.com/JSONAPI`

de la fonction `GetNodes` -- args= ([('phy_mem', '>=', '32G')],)

Envoi à `http://cloud.provider.com/JSONAPI`

de la fonction `BookNode` -- args= ({'id': 1002, 'phy_mem': '32G'},)

Discussion

Quelques commentaires en vrac au sujet de cette approche :

- l'interface est correcte ; l'objet `rpc_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 :

```

[3]: # 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}...")
        print(f"de 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 :

```

[4]: # 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') ] )

# réserver un noeud
node_lease = rpc_proxy.BookNode (
    { 'id' : 1002, 'phy_mem' : '32G' } )

```

```

Envoi à http://cloud.provider.com/JSONAPI...
de la fonction GetNodes -- args= ([('phy_mem', '>=', '32G')],)
Envoi à http://cloud.provider.com/JSONAPI...
de la fonction BookNode -- args= ({'id': 1002, 'phy_mem': '32G'},)

```

6.9 w6-s3-c1-heritage

Héritage

6.9.1 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 :

```
[1]: # Une classe mère
class Fleur:
    def implicite(self):
        print('Fleur.implicit')
    def redefinie(self):
        print('Fleur.redéfinie')
    def modifiee(self):
        print('Fleur.modifiée')

# Une classe fille
class Rose(Fleur):
    # on ne définit pas implicite
    # on redéfinit complètement 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.

```
[2]: # fille est une instance de Rose
fille = Rose()

fille.implicit()
```

Fleur.implicit

```
[3]: fille.redefinie()
```

Rose.redefinie

S'agissant des deux premières méthodes, le comportement qu'on observe est simplement la conséquence de l'algorithme de recherche d'attributs : `implicit` est trouvée dans la classe `Fleur` et `redefinie` est trouvée dans la classe `Rose`.


```
[4]: fille.modifiee()
```

```
Fleur.modifiée
Rose.modifiée apres Fleur
```

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 built-in `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 :

```
[5]: # Une version allégée de la classe fille, qui utilise super()
class Rose(Fleur):
    def modifiee(self):
        super().modifiee()
        print('Rose.modifiée apres Fleur')
```

```
[6]: fille = Rose()

fille.modifiee()
```

```
Fleur.modifiée
Rose.modifiée apres Fleur
```

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 :

```
[7]: # Une classe avec qui on n'aura pas de relation d'héritage
class Tige:
    def implicite(self):
        print('Tige.implicité')
    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 définir implicite
    def implicite(self):
        self.externe.implicite()

    # on redéfinit complètement redefinie
    def redefinie(self):
        print('Rose.redefinie')

    def modifiee(self):
        self.externe.modifiee()
        print('Rose.modifiee apres Tige')

```

```

[8]: # on obtient ici exactement le même comportement
# pour les trois sortes de méthodes
fille = Rose()

fille.implicite()
fille.redefinie()
fille.modifiee()

```

```

Tige.implicite
Rose.redefinie
Tige.modifiee
Rose.modifiee apres Tige

```

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).

6.9.2 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 :

```
+-----+
| BaseServer |
+-----+
    |
    v
+-----+ +-----+
| TCPServer |----->| UnixStreamServer |
+-----+ +-----+
    |
    v
+-----+ +-----+
| UDPServer |----->| UnixDatagramServer |
+-----+ +-----+
```

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 particulier](#), 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](#).

6.10 w6-s3-c2-namedtuple

Hériter des types built-in ?

6.10.1 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 - en français enregistrement - 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

```
[1]: p1 = {'x': 1, 'y': 2}
      # ou de manière équivalente
      p1 = dict(x=1, y=2)
```

ou une classe

```
[2]: 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.

Pour 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

```
[3]: from collections import namedtuple
```

Techniquement, il s'agit d'une fonction :

```
[4]: type(namedtuple)
```

```
[4]: function
```

qui renvoie une classe - oui les classes sont des objets comme les autres; par exemple pour créer une classe `TuplePoint`, on ferait :

```
[5]: # 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 :

```
[6]: p3 = TuplePoint(1, 2)
```

```
[7]: # cet objet est un tuple
      isinstance(p3, tuple)
```

```
[7]: True
```

```
[8]: # auquel je peux accéder par index
# comme un tuple
p3[0]
```

```
[8]: 1
```

```
[9]: # mais aussi par nom via un attribut
p3.x
```

```
[9]: 1
```

```
[10]: # et comme c'est un tuple il est immuable
try:
    p3.x = 10
except Exception as e:
    print(f"OOPS {type(e)} {e}")
```

```
OOPS <class 'AttributeError'> can't set attribute
```

À quoi ça sert

Les `namedtuple` ne sont 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` :

```
[11]: from pathlib import Path
dot_stat = Path('.').stat()
```

```
[12]: dot_stat
```

```
[12]: os.stat_result(st_mode=16877, st_ino=12922409772, st_dev=16777220, st_nlink
      =128, st_uid=501, st_gid=20, st_size=4096, st_atime=1590418630, st_mtime
      =1590418629, st_ctime=1590418629)
```

```
[13]: isinstance(dot_stat, tuple)
```

```
[13]: True
```

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 :

```
[14]: Foo = namedtuple('Bar', ['spam', 'eggs'])
```

```
[15]: # Foo est le nom de la variable classe
foo = Foo(1, 2)
```

```
[16]: # mais cette classe a son attribut __name__ mal positionné
Foo.__name__
```

```
[16]: 'Bar'
```

Il est donc évidemment 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 :

```
[17]: import sys

# p1 = dict / p2 = instance / p3 = namedtuple

for p in p1, p2, p3:
    print(sys.getsizeof(p))
```

240

56

64

Définir des méthodes sur un **namedtuple**

Dans un des compléments de la séquence précédente, intitulé “Manipuler des ensembles d’instances”, nous avons vu comment redéfinir le protocole hashable sur des instances, en mettant en évidence la nécessité de disposer d’instances non mutables lorsqu’on veut redéfinir `__hash__()`.

Voyons ici comment on pourrait tirer parti d’un **namedtuple** pour refaire proprement notre classe `Point2` - souvenez-vous, il s’agissait de rechercher dans un ensemble de points.

```
[18]: Point2 = namedtuple('Point2', ['x', 'y'])
```

Sans utiliser le mot-clé `class`, il faudrait se livrer à une petite gymnastique pour redéfinir les méthodes spéciales sur la classe `Point2`. Nous allons utiliser l’héritage pour arriver au même résultat :

```
[19]: # ce code est très proche du code utilisé dans le précédent complément
class Point2(namedtuple('Point2', ['x', 'y'])):

    # 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
```

Avec ceci en place on peut maintenant faire :

```
[20]: # trois points égaux au sens de cette classe
q1, q2, q3 = Point2(10, 10), Point2(10, 10), Point2(10, 10)
```

```
[21]: # deux objets distincts
q1 is q2
```

```
[21]: False
```

```
[22]: # mais égaux
q1 == q2
```

```
[22]: True
```

```
[23]: # ne font qu'un dans un ensemble
s = {q1, q2}
len(s)
```

```
[23]: 1
```

```
[24]: # et on peut les trouver
# par le troisième
q3 in s
```

```
[24]: True
```

```
[25]: # et les instances ne sont pas mutables
try:
    q1.x = 100
except Exception as e:
    print(f"OOPS {type(e)}")
```

OOPS <class 'AttributeError'>

6.11 w6-s3-c2-namedtuple

Pour en savoir plus

Vous pouvez vous reporter à [la documentation officielle](#).

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 (cette possibilité a disparu, malheureusement, dans python-3.7).

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.

```
[ ]: # exécuter ceci pour voir le détail de ce que fait `namedtuple`
import sys
major, minor, *_ = sys.version_info
if minor <= 6:
    Point = namedtuple('Point', ['x', 'y'], verbose=True)
else:
    print("désolé, le paramètre verbose a été supprimé en 3.7")

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

6.12 w6-s3-c3-dataclasses

dataclasses

Nouveauté de la version 3.7

Python 3.7 apporte un nouveauté pour simplifier la définition de classes dites “de données” ; ce type de classes s’applique pour des objets qui sont essentiellement un assemblage de quelques champs de données.

Aperçu

La raison d’être de `dataclass` est de fournir - encore un - moyen de définir des classes d’enregistrements.

Voici par exemple comment on pourrait définir une classe `Personne` :

```
[1]: from dataclasses import dataclass

@dataclass
```



```
class Personne:
    nom: str
    age: int
    email: str = ""
```

```
[2]: personne = Personne(nom='jean', age=12)
     print(personne)
```

```
Personne(nom='jean', age=12, email='')
```

Instances non mutables

Le décorateur `dataclass` accepte divers arguments pour choisir le comportement de certains aspects de la classe. Reportez-vous à la documentation pour une liste complète, mais voici un exemple qui utilise `frozen=True` et qui illustre la possibilité de créer des instances non mutables. Nous retrouvons ici le même scénario d'ensemble de points que nous avons déjà rencontré plusieurs fois :

```
[3]: from dataclasses import dataclass

     @dataclass(frozen=True)
     class Point:
         x: float
         y: float

         def __eq__(self, other):
             return self.x == other.x and self.y == other.y

         def __hash__(self):
             return (11 * self.x + self.y) // 16
```

```
[4]: p1, p2, p3 = Point(1, 1), Point(1, 1), Point(1, 1)
```

```
[5]: s = {p1, p2}
     len(s)
```

```
[5]: 1
```

```
[6]: p3 in s
```

```
[6]: True
```

```
[7]: try:
     p1.x = 10
     except Exception as e:
         print(f"OOPS {type(e)}")
```

```
OOPS <class 'dataclasses.FrozenInstanceError'>
```

Pour aller plus loin

Vous pouvez vous rapporter

- [au PEP 557](#) qui a abouti au consensus, et
- [à la documentation officielle du module](#).

6.13 w6-s3-c4-enums

Énumérations

6.13.1 Complément - niveau basique

On trouve dans d'autres langages la notion de types 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 :

```
[1]: from enum import Enum
```

```
[2]: class Flavour(Enum):  
    CHOCOLATE = 1  
    VANILLA = 2  
    PEAR = 3
```

```
[3]: vanilla = Flavour.VANILLA
```

Un premier avantage est que les représentations textuelles sont plus parlantes :

```
[4]: str(vanilla)
```

```
[4]: 'Flavour.VANILLA'
```

```
[5]: repr(vanilla)
```

```
[5]: '<Flavour.VANILLA: 2>'
```

Vous pouvez aussi retrouver une valeur par son nom :

```
[6]: chocolate = Flavour['CHOCOLATE']  
chocolate
```

```
[6]: <Flavour.CHOCOLATE: 1>
```

```
[7]: Flavour.CHOCOLATE
```

```
[7]: <Flavour.CHOCOLATE: 1>
```

Et réciproquement :

```
[8]: chocolate.name
```

```
[8]: 'CHOCOLATE'
```

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 :

```
[9]: 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
```

```
[10]: code = HttpError.REDIRECT_TMP
```

```
[11]: code.is_redirect()
```

```
[11]: True
```

Itération

Un des avantages de cette construction est qu'avec une énumération, l'objet classe (et non une instance) est un itérable :

```
[12]: class Couleur(IntEnum):
    TREFLE = 0
    CARREAU = 1
    COEUR = 2
    PIQUE = 3

    def glyph(self):
        glyphs = {
            Couleur.TREFLE: '\u2663',
            Couleur.CARREAU: '\x1b[31;1m\u2666\x1b[39;0m',
            Couleur.COEUR: '\x1b[31;1m\u2665\x1b[39;0m',
            Couleur.PIQUE: '\u2660',
        }
        return glyphs[self]
```

```
[13]: for couleur in Couleur:
    print(f"Couleur {couleur} -> {couleur.glyph()}")
```

```
Couleur 0 ->  
Couleur 1 ->  
Couleur 2 ->  
Couleur 3 ->
```

Pour en savoir plus

Consultez [la documentation officielle du module enum](#).

6.14 w6-s3-c5-heritage-typage

Héritage, typage

6.14.1 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.

1. 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;
2. 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 seconde 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 that bird a duck.

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** :

```
[1]: str.__bases__
```

```
[1]: (object,)
```

```
[2]: list.__bases__
```

```
[2]: (object,)
```

```
[3]: tuple.__bases__
```

```
[3]: (object,)
```

```
[4]: range.__bases__
```

```
[4]: (object,)
```

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 :

```
[5]: from collections.abc import Iterable
```

```
[6]: isinstance('ab', Iterable)
```

```
[6]: True
```

```
[7]: isinstance([1, 2], Iterable)
```

```
[7]: True
```

```
[8]: # 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
```

```
[9]: foo = Foo()  
     isinstance(foo, Iterable)
```

```
[9]: True
```

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 :

```
[10]: from collections.abc import Hashable
```

```
[11]: # un tuple qui contient une liste ne convient  
# pas comme clé dans un dictionnaire  
# et pourtant
```

```
isinstance ([1], [2]), Hashable)
```

```
[11]: True
```

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.

6.15 w6-s4-c1-heritage-multiple

Héritage multiple

6.15.1 Complément - niveau intermédiaire

La classe `object`

Le symbole `object` est une variable prédéfinie (qui donc fait partie du module `builtins`) :

```
[1]: object
```

```
[1]: object
```

```
[2]: import builtins

builtins.object is object
```

[2]: True

La classe `object` est une classe spéciale ; toutes les classes en Python héritent de la classe `object`, même lorsqu'aucun héritage n'est spécifié :

```
[3]: class Foo:
      pass

Foo.__bases__
```

[3]: (object,)

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 Python 2 entre classes old-style et classes new-style. Nous le mentionnons seulement car on rencontre encore parfois du code qui fait quelque chose comme :

```
[4]: # ceci est du vieux code, on n'a pas besoin
      # de faire hériter Bar de object
      class Bar(object):
          pass
```

qui est un reste de Python 2, et que Python 3 accepte uniquement au titre de la compatibilité.

6.15.2 Complément - niveau avancé

Rappels

L'héritage en Python consiste principalement en l'algorithme de recherche d'un attribut d'une instance ; celui-ci regarde :

1. d'abord dans l'instance ;
2. ensuite dans la classe ;
3. 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 décrivons 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.

6.16 w6-s4-c1-heritage-multiple

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 :

```
[5]: class LeftTop:
      def attribut(self):
```

```

        return "attribut(LeftTop)"

class LeftMiddle(LeftTop):
    pass

class Left(LeftMiddle):
    pass

class Middle:
    pass

class Right:
    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 :

```
[6]: instance.attribut() == 'attribut(LeftTop)'
```

```
[6]: True
```

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 :

```
[7]: 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()
```

```
[7]: 'attribut(Right)'
```

Ou encore bien entendu, si dans votre contexte vous devez appeler les deux méthodes dont vous pourriez hériter et les combiner, vous pouvez le faire aussi, par exemple comme ceci :

```
[8]: class Class(Left, 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, **kwds))

instance3 = Class()
instance3.attribut()
```

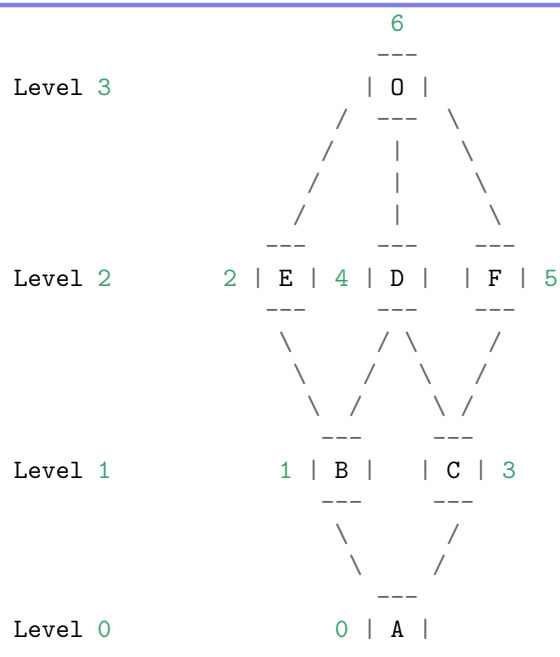
```
[8]: 'attribut(LeftTop) ** attribut(Right)'
```

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”).

```
[9]: 0 = object
      class F(0): pass
      class E(0): pass
      class D(0): 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 :

```
[10]: A.mro()
```

```
[10]: [__main__.A,
      __main__.B,
      __main__.E,
      __main__.C,
      __main__.D,
      __main__.F,
      object]
```

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 :

```
[11]: # 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}")
```

```
OOPS, <class 'TypeError'>, Cannot create a consistent method resolution
order (MRO) for bases X, Y
```

Pour en savoir plus

1. Un [blog de Guido Van Rossum](#) qui retrace l'historique des différents essais qui ont été faits avant de converger sur le modèle actuel.
2. Un [article technique](#) qui décrit le fonctionnement de l'algorithme de calcul de la MRO, et donne des exemples.
3. L'[article de Wikipedia](#) sur l'algorithme C3.

6.17 w6-s5-c1-attributs

Les attributs

6.17.1 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 données.

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 programmatiquement 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

```
[1]: import math
     # nous savons lire un attribut comme ceci
     # qui lit l'attribut de nom 'pi' dans le module math
     math.pi
```

```
[1]: 3.141592653589793
```

La fonction built-in `getattr` permet de lire un attribut programmatiquement :

```
[2]: # si on part d'une chaîne qui désigne le nom de l'attribut
     # la formule équivalente est alors
     getattr(math, 'pi')
```

```
[2]: 3.141592653589793
```

```
[3]: # 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__}`")
```

```
nom=foo, docstring=`une fonction vide`
```

```
[4]: # on peut préciser une valeur par défaut pour le cas où l'attribut
     # n'existe pas
     getattr(foo, "attribut_inexistant", 'valeur_par_defaut')
```

```
[4]: 'valeur_par_defaut'
```

Écrire un attribut

```
[5]: # on peut ajouter un attribut arbitraire (toujours sur l'objet fonction)
foo.hauteur = 100

foo.hauteur
```

```
[5]: 100
```

Comme pour la lecture on peut écrire un attribut programmativement avec la [fonction built-in setattr](#) :

```
[6]: # écrire un attribut avec setattr
setattr(foo, "largeur", 200)

# on peut bien sûr le lire indifféremment
# directement comme ici, ou avec getattr
foo.largeur
```

```
[6]: 200
```

Liste des attributs

La [fonction built-in hasattr](#) permet de savoir si un objet possède ou pas un attribut :

```
[7]: # pour savoir si un attribut existe
hasattr(math, 'pi')
```

```
[7]: True
```

Ce qui peut aussi être retrouvé autrement, avec la [fonction built-in vars](#) :

```
[8]: vars(foo)
```

```
[8]: {'hauteur': 100, 'largeur': 200}
```

Sur quels objets

Il n'est pas possible d'ajouter des attributs sur les types de base, car ce sont des classes immuables :

```
[9]: 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}")
```

```
int → exception <class 'AttributeError'> - 'int' object has no attribute 'foo'
```

```
str → exception <class 'AttributeError'> - 'str' object has no attribute 'foo'
```

```
float → exception <class 'AttributeError'> - 'float' object has no attribute 'foo'
```

```

tribune 'foo'
complex → exception <class 'AttributeError'> - 'complex' object has no attribute 'foo'
tuple → exception <class 'AttributeError'> - 'tuple' object has no attribute 'foo'
dict → exception <class 'AttributeError'> - 'dict' object has no attribute 'foo'
set → exception <class 'AttributeError'> - 'set' object has no attribute 'foo'
frozenset → exception <class 'AttributeError'> - 'frozenset' object has no attribute 'foo'

```

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

6.18 w6-s5-c2-namespaces

Espaces de nommage

6.18.1 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 `=`

```

[1]: class MaClasse:
      pass

      # affectation explicite
      MaClasse.x = 10

      # le symbole x est défini dans l'espace de nommage de MaClasse

```

```
'x' in MaClasse.__dict__
```

```
[1]: True
```

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 éloignée ;
- 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 :

```
[2]: 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()
```

dans la fonction 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.

6.18.2 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 :

```
[3]: 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)
```

```
dans classe A: x du module
dans classe A: x dans A
dans classe A: x du module
```

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.

6.18.3 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 :

```
[1]: from itertools import permutations
```

```
[2]: set = {1, 2, 3}

for p in permutations(set):
    print(p)
```

```
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

Une implémentation

Voici une implémentation possible pour un itérateur de permutations :

```
[3]: 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 itérateur 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-séquence
#
# naïvement 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 longueur 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 :

```
[4]: 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
```

```
[5]: show_first_items(Permutations(12), 20)
```

```
Il y a 479001600 items dans l'itérable
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 11, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 11, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 11, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 11, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 11, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 11, 4, 5, 6, 7, 8, 9, 10]
```

```

[0, 1, 2, 11, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 11, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 11, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[11, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 9, 11]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 11, 10, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 11, 8, 10, 9]
[0, 1, 2, 3, 4, 5, 6, 11, 7, 8, 10, 9]
[0, 1, 2, 3, 4, 5, 11, 6, 7, 8, 10, 9]
[0, 1, 2, 3, 4, 11, 5, 6, 7, 8, 10, 9]
[0, 1, 2, 3, 11, 4, 5, 6, 7, 8, 10, 9]
[0, 1, 2, 11, 3, 4, 5, 6, 7, 8, 10, 9]
...

```

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 logorrhée de commentaires ça redevient presque lisible ;)

```

[6]: 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 insère n-1
    return self.subsequence[0:cutter] + [self.n-1] \
        + self.subsequence[cutter:self.n-1]

# la longueur de cet itérateur est connue
def __len__(self):
    import math
    return math.factorial(self.n)

```

```
[7]: show_first_items(Permutations2(5), 20)
```

Il y a 120 items dans l'itérable

```

[0, 1, 2, 3, 4]
[0, 1, 2, 4, 3]
[0, 1, 4, 2, 3]
[0, 4, 1, 2, 3]
[4, 0, 1, 2, 3]
[0, 1, 3, 2, 4]
[0, 1, 3, 4, 2]
[0, 1, 4, 3, 2]
[0, 4, 1, 3, 2]
[4, 0, 1, 3, 2]
[0, 3, 1, 2, 4]
[0, 3, 1, 4, 2]
[0, 3, 4, 1, 2]
[0, 4, 3, 1, 2]
[4, 0, 3, 1, 2]
[3, 0, 1, 2, 4]
[3, 0, 1, 4, 2]
[3, 0, 4, 1, 2]
[3, 4, 0, 1, 2]
[4, 3, 0, 1, 2]
[0, 2, 1, 3, 4]

```

...

Il me semble intéressant de montrer une autre façon, plus simple, d'écrire un itérateur de permutations, à base cette fois de générateurs; c'est un tout petit peu une digression par rapport au cours qui est sur la conception d'itérateurs et d'itérables. Ça va nous permettre surtout de réviser la notion de `yield from`.

On commence par une version très rustique qui fait des impressions :

```

[8]: # pour simplifier ici on suppose que l'entrée est une vraie liste
      # que l'on va ainsi pouvoir modifier par effets de bord
      def gen_perm1(subject, k=0):

```

```

if k == len(subject):
    # cette version hyper rustique se contente de faire une impression
    print(subject)
else:
    for i in range(k, len(subject)):
        # on échange
        subject[k], subject[i] = subject[i], subject[k]
        gen_perm1(subject, k+1)
        # on remet comme c'était pour le prochain échange
        subject[k], subject[i] = subject[i], subject[k]

```

```
[9]: gen_perm1(['a', 'b', 'c', 'd'])
```

```

['a', 'b', 'c', 'd']
['a', 'b', 'd', 'c']
['a', 'c', 'b', 'd']
['a', 'c', 'd', 'b']
['a', 'd', 'c', 'b']
['a', 'd', 'b', 'c']
['b', 'a', 'c', 'd']
['b', 'a', 'd', 'c']
['b', 'c', 'a', 'd']
['b', 'c', 'd', 'a']
['b', 'd', 'c', 'a']
['b', 'd', 'a', 'c']
['c', 'b', 'a', 'd']
['c', 'b', 'd', 'a']
['c', 'a', 'b', 'd']
['c', 'a', 'd', 'b']
['c', 'd', 'a', 'b']
['c', 'd', 'b', 'a']
['d', 'b', 'c', 'a']
['d', 'b', 'a', 'c']
['d', 'c', 'b', 'a']
['d', 'c', 'a', 'b']
['d', 'a', 'c', 'b']
['d', 'a', 'b', 'c']

```

Très bien, mais on ne veut pas imprimer, on veut itérer. On pourrait se dire, il me suffit de remplacer `print` par `yield`. Essayons cela :

```

[10]: # pour simplifier ici on suppose que l'entrée est une vraie liste
# que l'on va ainsi pouvoir modifier par effets de bord
def gen_perm2(subject, k=0):
    if k == len(subject):
        # cette version hyper rustique se contente de faire une impression
        yield subject
    else:
        for i in range(k, len(subject)):
            # on échange
            subject[k], subject[i] = subject[i], subject[k]
            gen_perm2(subject, k+1)
            # on remet comme c'était pour le prochain échange
            subject[k], subject[i] = subject[i], subject[k]

```

```
[11]: for perm in gen_perm2(['a', 'b', 'c', 'd']):
        print(perm)
```

On est exactement dans le cas où il nous faut utiliser `yield from`. En effet lorsqu'on appelle `gen_perm(subject, k+1)` ici, ce qu'on obtient en retour c'est maintenant un objet générateur. Pour faire ce qu'on cherche à faire il nous faut bien utiliser cet objet générateur, et pour cela on utilise `yield from`.

```
[12]: # pour simplifier ici on suppose que l'entrée est une vraie liste
# que l'on va ainsi pouvoir modifier par effets de bord
def gen_perm3(subject, k=0):
    if k == len(subject):
        # cette version hyper rustique se contente de faire une impression
        yield subject
    else:
        for i in range(k, len(subject)):
            # on échange
            subject[k], subject[i] = subject[i], subject[k]
            yield from gen_perm3(subject, k+1)
            # on remet comme c'était pour le prochain échange
            subject[k], subject[i] = subject[i], subject[k]
```

```
[13]: for perm in gen_perm3(['a', 'b', 'c', 'd']):
        print(perm)
```

```
['a', 'b', 'c', 'd']
['a', 'b', 'd', 'c']
['a', 'c', 'b', 'd']
['a', 'c', 'd', 'b']
['a', 'd', 'c', 'b']
['a', 'd', 'b', 'c']
['b', 'a', 'c', 'd']
['b', 'a', 'd', 'c']
['b', 'c', 'a', 'd']
['b', 'c', 'd', 'a']
['b', 'd', 'c', 'a']
['b', 'd', 'a', 'c']
['c', 'b', 'a', 'd']
['c', 'b', 'd', 'a']
['c', 'a', 'b', 'd']
['c', 'a', 'd', 'b']
['c', 'd', 'a', 'b']
['c', 'd', 'b', 'a']
['d', 'b', 'c', 'a']
['d', 'b', 'a', 'c']
['d', 'c', 'b', 'a']
['d', 'c', 'a', 'b']
['d', 'a', 'c', 'b']
['d', 'a', 'b', 'c']
```


6.19 w6-s8-c1-context-manager-et-exception

Context managers et exceptions

6.19.1 Complément - niveau intermédiaire

On a vu jusqu'ici dans la vidéo comment écrire un context manager; on a vu notamment qu'il était bon pour la méthode `__exit__()` de retourner `False`, de façon à ce que l'exception soit propagée à l'instruction `with` :

```
[1]: import time

class Timer1:
    def __enter__(self):
        print("Entering Timer1")
        self.start = time.time()
        return self

    # en règle générale on se contente de propager l'exception
    # à l'instruction with englobante
    def __exit__(self, *args):
        print(f"Total duration {time.time()-self.start:2f}")

        # et pour cela il suffit que __exit__ retourne False
        return False
```

Ainsi si le corps de l'instruction lève une exception, celle-ci est propagée :

```
[2]: import time
try:
    with Timer1():
        time.sleep(0.5)
        1/0
except Exception as exc:
    # on va bien recevoir cette exception
    print(f"OOPS -> {type(exc)}")
```

```
Entering Timer1
Total duration 0.501005
OOPS -> <class 'ZeroDivisionError'>
```

À la toute première itération de la boucle, on fait une division par 0 qui lève l'exception `ZeroDivisionError`, qui passe bien à l'appelant.

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 par exemple devoir attraper les exceptions liées à la fin du fichier, mais doit par contre laisser passer une exception comme `ZeroDivisionError`.

Les paramètres de `__exit__`

Si on a besoin de filtrer entre les exceptions - c'est-à-dire en laisser passer certaines et pas d'autres - il nous faut quelque chose de plus pour pouvoir faire le tri. Comme [vous pouvez le retrouver ici](#), la méthode `__exit__` reçoit trois arguments :

```
def __exit__(self, exc_type, exc_value, traceback):
```

- si l'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 respectivement au type, à la valeur de l'exception, et à l'état de la pile lorsque l'exception est levée.

Pour illustrer cela, écrivons une nouvelle version de `Timer` qui filtre, disons, l'exception `ZeroDivisionError` que je choisis au hasard, c'est uniquement pour illustrer le mécanisme.

```
[3]: # une deuxième version de Timer
# qui propage toutes les exceptions sauf 'OSError'

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:
            # pas d'exception levée dans le corps du 'with'
            print(f"Total duration {time.time()-self.start:2f}")
            # dans ce cas la valeur de retour n'est pas utilisée
        else:
            # il y a eu une exception de type 'exc_type'
            if exc_type in (ZeroDivisionError,) :
                print("on étouffe")
                # on peut l'étouffer en retournant True
                return True
            else:
                print(f"OOPS : on propage l'exception "
                      f"{exc_type} - {exc_value}")
                # et pour ça il suffit... de ne rien faire du tout
                # ce qui renverra None
```

```
[4]: # commençons avec un code sans souci
try:
    with Timer2():
        time.sleep(0.5)
except Exception as e:
    # on va bien recevoir cette exception
    print(f"OOPS -> {type(e)}")
```

```
Entering Timer1
Total duration 0.501478
```

```
[5]: # avec une exception filtrée
try:
    with Timer2():
        time.sleep(0.5)
        1/0
except Exception as e:
    # on va bien recevoir cette exception
    print(f"OOPS -> {type(e)}")
```

Entering Timer1
on étouffe

```
[6]: # avec une autre exception
try:
    with Timer2():
        time.sleep(0.5)
        raise OSError()
except Exception as e:
    # on va bien recevoir cette exception
    print(f"OOPS -> {type(e)}")
```

Entering Timer1
OOPS : on propage l'exception <class 'OSError'> -
OOPS -> <class 'OSError'>

La bibliothèque **contextlib**

Je vous signale aussi [la bibliothèque contextlib](#) qui offre quelques utilitaires pour se définir un context-manager.

Notamment, elle permet d'implémenter un context manager sous une forme compacte à l'aide d'une fonction génératrice - et du décorateur `contextmanager` :

```
[7]: from contextlib import contextmanager
```

```
[8]: # l'objet compact_timer est un context manager !
@contextmanager
def compact_timer(message):
    start = time.time()
    yield
    print(f"{message}: duration = {time.time() - start}")
```

```
[9]: with compact_timer("Squares sum"):
    print(sum(x**2 for x in range(10**5)))
```

333328333350000
Squares sum: duration = 0.034197092056274414

Un peu comme on peut implémenter un itérateur à partir d'une fonction génératrice qui fait (n'importe quel nombre de) `yield`, ici on implémente un context manager compact sous la forme d'une fonction génératrice.

Comme vous l'avez sans doute deviné sur la base de cet exemple, il faut que la fonction fasse exactement un `yield` : ce qui se passe avant le `yield` est du ressort de `__enter__`, et la fin est du ressort de `__exit__`.

Bien entendu on n'a pas la même puissance d'expression avec cette méthode par rapport à une vraie classe, mais cela permet de créer des context managers avec le minimum de code.

6.20 w6-s8-x1-classes-merger

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 avons 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 :

```
$ python3 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, 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 lesquels 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'FGSA', u'F
```

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.

6.20.1 Niveaux pour l'exercice

Quel que 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 :

```
$ python3 merger.py json/*.json
...
Comparing ALL_SHIPS.txt and ALL_SHIPS.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

```
$ python3 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 susceptibles 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 optionally 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):
    """
    makes a float readable; e.g. transforms 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}''"
```

6.21 w6-s9-c1-organisation-sources

Comment organiser les sources de votre projet Python

Où on va voir que : * c'est bien de grouper son code dans un package * mais à première vue ça casse tout, cependant pas de panique ! * il ne FAUT PAS tripoter la variable **PYTHONPATH** * il faut au contraire créer un **setup.py**, et ensuite lancer une fois
pip install -e .
 pour pouvoir utiliser le code en mode développeur

6.21.1 Complément - niveau intermédiaire

Vous venez d'écrire un super algorithme qui simule le climat de l'an 2100, et vous voulez le publier ? Nous allons voir ici comment organiser les sources de votre projet, pour que ce soit à la fois

- pratique pour vous de tester votre travail pendant le développement
- facile de publier le code pour que d'autres puissent l'installer et l'utiliser
- et éventuellement facile pour d'autres de contribuer à votre projet.

6.21.2 Les infrastructures

En 2020 on ne travaille plus tout seul dans son coin ; il est à la portée de tous d'utiliser et de tirer profit d'infrastructures, ouvertes et gratuites (pour les usages de base au moins) :

Pour ce qui nous concerne ici, voici celles qui vont nous être utiles :

- **PyPI** - (prononcer "paille - pis - ail") pour Python Package Index, est l'endroit où chacun peut publier ses librairies
- **github** - ainsi que ses concurrents **gitlab** et **bitbucket** - sont bien sûr des endroits où l'on peut déposer ses sources pour partage, sous la forme de dépôt **git**

Optionnellement, sachez qu'il existe également des infrastructures pour les deux grandes tâches que sont la documentation et le test, souvent considérées - à tort - comme annexes :

- [readthedocs](#) est une infra qui permet d'exposer la documentation
- [travis](#) est - parmi plein d'autres - une infrastructure permettant d'exécuter une suite de tests

S'agissant de ces deux derniers points : souvent on s'arrange pour que tout soit automatique ; quand tout est en place, il suffit de pousser un nouveau commit auprès de github (par exemple) pour que

- tous les tests soient repassés (d'où le terme de CI* = Continuous Integration) ; du coup on sait en permanence si tel ou tel commit a cassé ou non l'intégrité du code ;
- la documentation soit mise à jour, exposée à tout le monde, et navigable par numéro de version.

Alors bon bien sûr ça c'est le monde idéal ; on ne passe pas d'un seul coup, d'un bout de code qui tient dans un seul module `bidule.py`, à un projet qui utilise tout ceci ; on n'a pas forcément besoin non plus d'utiliser toutes ces ressources (et bien entendu, aucun de ces trucs n'est obligatoire).

Aussi nous allons commencer par le commencement.

6.21.3 Le commencement : créer un package

Le commencement, ça consiste à se préparer à coexister avec d'autres librairies.

Si votre code expose disons une classe `Machine` dans le fichier/module `machine.py`, la première chose consiste à trouver un nom unique ; rien ne vous permet de penser qu'il n'y a pas une autre bibliothèque qui expose un module qui s'appelle aussi `machine` (il y a même fort à parier qu'il y en a plein !). Aussi ce qu'on va commencer par faire c'est d'installer tout notre code dans un package.

Concrètement ça va signifier se mettre dans un sous-dossier, mais surtout d'un point de vue des utilisateurs potentiels de la classe, ça veut dire qu'au lieu de faire juste :

```
from machine import Machine
```

on va décider qu'à partir de maintenant il faut toujours faire

```
from bidule.machine import Machine
```

et de cette façon tous les noms qui sont propres à notre code ne sont accessibles que via l'espace de noms `bidule`, et on évite les conflits avec d'autres bibliothèques.

Choisir le nom du package

Bien sûr ceci ne fonctionne que si je peux être sûr que `bidule` est à moi, de sorte que personne demain ne publie une librairie qui utilise le même nom.

C'est pourquoi je recommande, à ce stade, de s'assurer de prendre un nom qui n'est pas déjà pris ; en toute rigueur c'est optionnel, tant que vous ne prévoyez pas de publier votre appli sur pypi (car bien sûr c'est optionnel de publier sur pypi), mais ça coûte moins cher de le faire très tôt, ça évite des renommages fastidieux plus tard.

Donc pour s'assurer de cela, on va tout simplement demander à `pypi`, qui va jouer le rôle de registrar, et nous garantir l'exclusivité de ce nom. Vous pouvez soit chercher votre nom [directement dans le site pypi](#), ou bien utiliser `pip`

```
pip search bidule
```

Le nom est libre, pour toute la suite je choisis `bidule` comme mon nom de package.

Vous trouverez dans ce repo git <https://github.com/flotpython/bidule> un microscopique petit projet qui illustre notre propos.

Adapter son code

Une fois que j'ai choisi mon nom de package, donc ici `bidule`, je dois :

1. mettre tout mon code dans un répertoire qui s'appelle `bidule`,
2. et modifier mes importations ; maintenant j'importe tout au travers du seul package `bidule`.

Donc je remplace les importations partout ; ce qui avant aurait été simplement

```
from machine import Machine
```

devient

```
from bidule.machine import Machine
```

Remarque : imports relatifs Lorsqu'un fichier a besoin d'en importer dans le même package, on a le choix ; par exemple ici, `machine.py` a besoin d'importer la fonction `helper` du fichier `helpers.py`, il peut faire

```
from bidule.helpers import helper
```

mais aussi plus simplement avec un import relatif :

```
from .helpers import helper
```

remarquez le `.` dans `.helpers`, qui signifie dans le même package que moi.

Je recommande toutefois de ne pas se précipiter avec ces imports relatifs, et notamment de ne pas les utiliser dans un point d'entrée (le fichier qu'on passe à l'interpréteur Python) car ça ne fonctionne pas dans ce cas.

C'est tout cassé

À ce stade précisément, vous constatez .. que plus rien ne marche !

En effet, comme on l'a vu dans le complément sur le chargement des modules, Python recherche vos modules dans l'ordre

- le dossier du point d'entrée
- la variable d'environnement `PYTHONPATH`
- les dossiers système

Et donc si vous m'avez suivi, vous devez avoir quelque chose comme

```
mon-repo-git/  
    bidule/  
        main.py  
        machine.py  
        helpers.py
```

mais alors quand vous faites

```
$ python bidule/main.py  
Traceback (most recent call last):
```

```
File "bidule/main.py", line 1, in <module>
    from bidule.machine import Machine
ModuleNotFoundError: No module named 'bidule'````
```

on va chercher du coup un module `bidule` à partir du répertoire du point d'entrée qui est le dossier `bidule/`, donc on ne trouve pas. ““

Le mauvais réflexe

Du coup naturellement, on se dit, ça n'est pas grave, je vais tirer profit de la variable `PYTHONPATH`. Alors disons-le tout net : Ce n'est pas une bonne idée, ce n'est pas du tout pour ce genre de cas qu'elle a été prévue.

Le fait de modifier une variable d'environnement est un processus tarabiscoté, même sans parler de Windows, et cette approche est une bonne façon de se tirer une balle dans le pied ; un jour ou l'autre la variable ne sera pas positionnée comme il faut, c'est sûr.

Bref, il ne faut pas faire comme ça !!

6.21.4 Le bon réflexe : `setup.py`

Non, le bon réflexe ici c'est d'écrire un fichier `setup.py`, et de l'utiliser pour faire ce qu'on pourrait une installation en mode développeur. Voyons cela :

Je commence donc par créer un fichier `setup.py` à la racine de mon repo git, dans lequel je mets, pour commencer, le minimum :

```
# minimal setup.py to install in develop mode

from setuptools import setup, find_packages

setup(
    name="bidule",
    packages=find_packages(),
)
```

Attention : nous sommes en 2020 et il faut utiliser le package `setuptools`, qui ne fait pas partie de la librairie standard (et non pas le module `distutils` qui, lui, en fait pourtant partie) ; donc comme d'habitude si c'est nécessaire, faites dans le terminal :

```
pip install setuptools
```

Installation en mode développeur : `pip install -e .`

Avec ce fichier en place, et toujours à la racine de mon repo, je peux maintenant faire la formule magique (toujours dans le terminal)

```
$ pip install -e .
Obtaining file:///Users/tparment/git/flotpython-course/w6/mon-repo-git
Installing collected packages: bidule
  Attempting uninstall: bidule
    Found existing installation: bidule 0.0.0
    Uninstalling bidule-0.0.0:
      Successfully uninstalled bidule-0.0.0
```

```
Running setup.py develop for bidule
Successfully installed bidule
```

L'effet de cette commande est de modifier mon environnement pour que le répertoire courant (le `.` dans `pip install -e .`) soit utilisé pour la recherche des modules. Ça signifie que je peux maintenant lancer mon programme sans souci :

```
$ python bidule/main.py
... déroulement normal
```

Et je peux modifier mon code dans le répertoire courant, ce sera bien ce code-là qui sera utilisé ; cette précision pour ceux qui penseraient que, comme on fait une installation, cela pourrait être fait par copie, mais ce n'est pas le cas, donc sauf gros changement dans le contenu, on n'a plus besoin de refaire le `pip install -e .`

Un `setup.py` plus raisonnable

Au delà de cette première utilité, `setup.py` sert à configurer plein d'aspects de votre application ; lorsque votre projet va gagner en maturité, il sera exécuté lorsque vous préparez le packaging, lorsque vous uploadez le package, et au moment d'installer (comme on vient de le voir).

Du coup en pratique, les besoins s'accumulent au fur et à mesure de l'avancement du projet, et on met de plus en plus d'informations dans le `setup.py` ; voici, que j'essaie de mettre dans l'ordre chronologique, quelques ajouts très fréquents [reportez-vous à la doc pour une liste complète](#) :

- `name` est le nom sous lequel votre projet sera rangé dans PyPI
- `packages` est une liste de noms de packages ; tel qu'on l'a écrit, cela sera calculé à partir du contenu de votre dépôt ; dans notre cas on aurait pu aussi bien écrire en dur `['bidule']` ; dans les cas les plus simples on a `packages == [name]`
- `version` est bien entendu important dès que vous commencez à publier sur PyPI (et même avant) pour que PyPI puisse servir la version la plus récente, et/ou satisfaire des exigences précises (les applis qui vous utilisent peuvent par exemple préciser une version minimale, etc...) Cette chaîne devrait être [compatible avec semver \(semantic versioning\)](#) i.e. qu'un numéro de version usuel contient 3 parties (major, minor, patch), comme par ex. "2.1.3" le terme `semantic` signifie ici que toute rupture de compatibilité doit se traduire par une incrémentation du numéro majeur (sauf s'il vaut 0, on a le droit de tâtonner avec une 0.x ; d'où l'importance de la version 1.0)
- `install_requires` : si votre package a besoin d'une librairie non-standard, disons par exemple `numpy`, il est très utile de le préciser ici ; de cette façon, lorsqu'un de vos utilisateurs installera votre appli avec `pip install bidule`, `pip` pourra gérer les dépendances et s'assurer que `numpy` est installé également ; bien sûr on n'en est pas là, mais je vous recommande de maintenir dès le début la liste de vos dépendances ici
- informatifs : `author`, `author_email`, `description`, `keywords`, `url`, `license`, pour affichage sur PyPI ; une mention spéciale à propos de `description_long`, qu'en général on veut afficher à partir de `README.md`, d'où l'idiome fréquent :

```
setup(
    ...
    long_description=open('README.md').read(),
    long_description_content_type = "text/markdown",
    ...
)
```

- etc... beaucoup d'autres réglages et subtilités autour de `setup.py` ; je conseille de prendre les choses comme elles viennent : commencez avec la liste qui est ici, et n'ajoutez d'autres trucs que lorsque ça correspond à un besoin pour vous !

Packager un point d'entrée

Assez fréquemment on package des librairies ; dans ce cas on se soucie d'installer uniquement des modules Python.

Mais imaginez maintenant que votre package contient aussi un point d'entrée - c'est-à-dire en fin de compte une commande que vos utilisateurs vont vouloir lancer depuis le terminal. Ce cas de figure change un peu la donne ; il faut maintenant installer des choses à d'autres endroits du système (pensez par exemple, sur linux/macOS, à quelque chose comme `/usr/bin`).

Dans ce cas surtout n'essayez pas de le faire vous-même ; c'est beaucoup trop compliqué à faire correctement !

Pour illustrer la bonne façon de faire dans ce cas, je vous renvoie pour les détails à un exemple réel, mais pour l'essentiel :

- je vous conseille d'écrire tout le code en question dans une classe habituelle, que vous rangez normalement avec les autres ;
- cette classe expose typiquement une méthode `main()`, qui retourne, pour suivre les conventions usuelles :
 - 0 si tout s'est bien passé
 - 1 sinon
- vous créez un module `__main__.py` qui se contente de créer une instance et de lui envoyer la méthode `main` - voir l'exemple
- vous déclarez cela dans `setup.py` qui se chargera de tout :-)

Voici tout ceci illustré sur un exemple réel. Dans cet exemple, le package (PyPI) s'appelle `apssh`, la commande qu'on veut exposer s'appelle `apssh`, du coup on a * un dossier `apssh` pour matérialiser le package * un module `apssh/apssh.py`, qui définit * une classe `Apssh`, qui expose une méthode `main()`

Voici les différents codes ; le détail de la classe elle-même n'est pas pertinent (c'est très long), c'est pour vous montrer un système de nommage, disons habituel :

- la définition de `entry_points` dans `setup.py`
ici après installation avec `pip`, nos utilisateurs pourront utiliser la commande `apssh`, qui est de cette façon associée au module `__main__.py`
(les termes `entry_points` et `console_scripts` ne doivent pas être modifiés) ;
- le module `__main__.py` ;
- la classe `Apssh` qui fait le travail se trouve dans un module usuel, ici `apssh.py`.

6.21.5 Publier sur PyPI

Pour publier votre application sur PyPI, rien de plus simple :

- il faut naturellement obtenir un login/password
- avant de pouvoir utiliser le nom `bidule`, il faut l'enregistrer :
`python setup.py register`
- aussi il vous faudra installer `twine`
`pip install twine`

Ensuite à chaque version, une fois que les tests sont passés et tout :

- préparer le packaging
`python setup.py sdist bdist_wheel`
- pousser sur PyPI
`twine upload dist/*`

Signalons enfin qu’il existe une infra PyPI “de test” sur <https://test.pypi.org> utile quand on ne veut pas polluer l’index officiel.

6.21.6 Utiliser **pip** pour installer

Ensuite une fois que c’est fait, le monde entier peut profiter de votre magnifique contribution en faisant bien sûr

```
pip install bidule
```

Remarquez que l’on conseille parfois, pour éviter d’éventuels soucis de divergence entre les commandes `python/python3` et `pip/pip3`, * de remplacer tous les appels à `pip` * par plutôt `python -m pip`, qui permet d’être sûr qu’on installe dans le bon environnement.

D’autres formes utiles de `pip` :

- `pip show bidule` : pour avoir des détails sur un module précis
- `pip freeze` : pour une liste complète des modules installés dans l’environnement, avec leur numéro de version
- `pip list` : sans grand intérêt, si ce n’est dans sa forme
`pip list -o` qui permet de lister les modules qui pourraient être mis à jour
- `pip install -r requirements.txt` : pour installer les modules dont la liste est dans le fichier `requirements.txt`

6.21.7 Packages et `__init__.py`

Historiquement avant la version 3.3 pour qu’un dossier se comporte comme un package il était obligatoire d’y créer un fichier de nom `__init__.py` - même vide au besoin.

Ce n’est plus le cas depuis cette version. Toutefois, il peut s’avérer utile de créer ce fichier, et si vous lisez du code vous le trouverez très fréquemment.

L’intérêt de ce fichier est de pouvoir agir sur : * le contenu du package lui-même, c’est-à-dire les attributs attachés à l’objet module associé à ce dossier, * et accessoirement d’exécuter du code supplémentaire.

Un usage particulièrement fréquent consiste à “remonter” au niveau du package les symboles définis dans les sous-modules. Voyons ça sur un exemple.

Dans notre repo de démonstration, nous avons une classe `Machine` définie dans le module `bidule.machine`. Donc de l’extérieur pour me servir de cette classe je dois faire

```
from bidule.machine import Machine
```

C’est très bien, mais dès que le contenu va grossir, je vais couper mon code en de plus en plus de modules. Ce n’est pas tellement aux utilisateurs de devoir suivre ce genre de détails. Donc si je veux pouvoir changer mon découpage interne sans impacter les utilisateurs, je vais vouloir qu’on puisse faire plutôt, simplement

```
from bidule.machine import Machine
```

pour y arriver il me suffit d’ajouter cette ligne dans le `__init__.py` du package `bidule` :

```
import Machine from .machine
```

qui du coup va définir le symbole `Machine` directement dans l'objet package.

6.21.8 Environnements virtuels

Terminons ce tour d'horizon pour dire un mot des environnements virtuels.

Par le passé, on installait python une seule fois dans le système ; en 2020, c'est une approche qui n'a que des inconvénients :

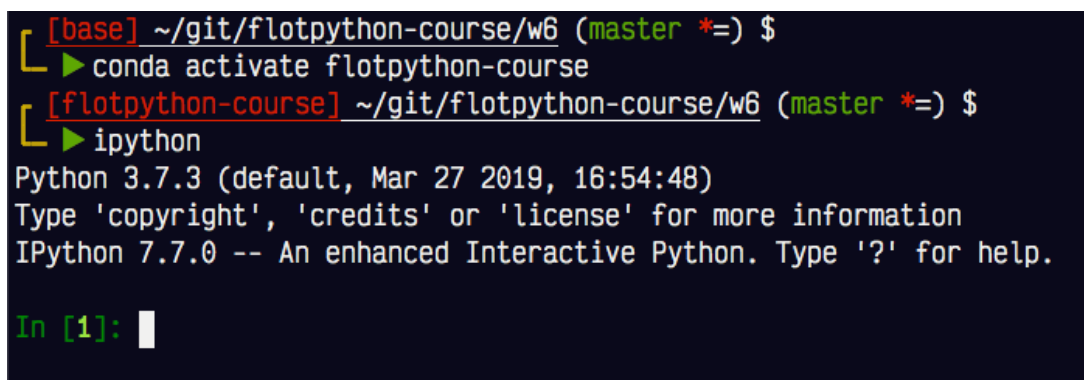
- quand on travaille sur plusieurs projets, on peut avoir besoin de Python-3.6 sur l'un et Python-3.8 sur un autre ;
- ou alors on peut avoir un projet qui a besoin de `Django==2.2` et un autre qui ne marche qu'avec `Django>=3.0` ;
- en plus par dessus le marché, dans certains cas il faut être super utilisateur pour modifier l'installation ; typiquement on passe son temps à faire `sudo pip` au lieu de `pip`...

et le seul avantage, c'est que tous les utilisateurs de l'ordi peuvent partager l'installation ; sauf que, plus de 99 fois sur 100, il n'y a qu'un utilisateur pour un ordi ! Bref, c'est une pratique totalement dépassée.

La création et la gestion d'environnements virtuels sont très faciles aujourd'hui. Aussi c'est une pratique recommandée de se créer un `virtualenv` par projet. C'est tellement pratique qu'on n'hésite pas une seconde à repartir d'un environnement vide à la moindre occasion, par exemple lorsqu'on a un doute sur les dépendances.

Le seul point sur lequel il faut être attentif, c'est de trouver un moyen de savoir en permanence dans quel environnement on se trouve. Notamment :

- une pratique très répandue consiste à s'arranger pour que le prompt dans le terminal indique cela,
- dans vs-code, dans la bannière inférieure, on nous montre toujours l'environnement courant.



```
[base] ~/git/flotpython-course/w6 (master *) $
└─▶ conda activate flotpython-course
[flotpython-course] ~/git/flotpython-course/w6 (master *) $
└─▶ ipython
Python 3.7.3 (default, Mar 27 2019, 16:54:48)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.7.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

figure : le prompt dans le terminal nous montre le venv courant

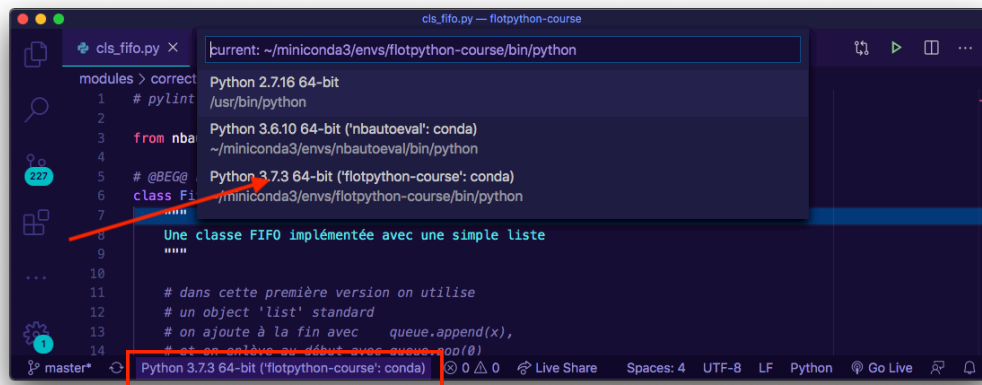


figure : vs-code nous montre le venv courant et nous permet de le changer

Les outils

Par contre il reste le choix entre plusieurs outils, que j'essaie de lister ici :

- **venv** un module de la librairie standard
- **virtualenv** un module externe, qui préexistait à **venv** et qui a fourni la base des fonctionnalités de **venv**
- **miniconda** un sous-produit de anaconda

Actuellement j'utilise quant à moi **miniconda**.

Voici à titre indicatif une session sous MacOS en guise de rapide introduction.

Vous remarquerez comme le prompt reflète l'environnement dans lequel on se trouve, ça semble relativement impératif si on ne veut pas s'emmêler les pinceaux.

La liste de mes environnements

```
[base] ~ $ conda env list
# conda environments:
#
base                  * /Users/tparment/miniconda3
<snip ...>
```

j'en crée un nouveau avec Python-3.8

```
[base] ~ $ conda create -n demo-py38 python=3.8
Collecting package metadata (current_repodata.json): done
Solving environment: done
<snip ...>
```

on le voit

```
[base] ~ $ conda env list
# conda environments:
#
base                  * /Users/tparment/miniconda3
```

```
demo-py38 /Users/tparment/miniconda3/envs/demo-py38
<snip...>
```

pour entrer dans le nouvel environnement

```
[base] ~ $ conda activate demo-py38
[demo-py38] ~ $
```

les packages installés

très peu de choses

```
[demo-py38] ~ $ pip list
Package      Version
-----
certifi      2020.4.5.1
pip          20.0.2
setuptools   46.2.0.post20200511
wheel        0.34.2
```

on y installe ce qu'on veut

```
[demo-py38] ~ $ pip install numpy==1.15.3
```

la version de python

```
[demo-py38] ~ $ python --version
Python 3.8.2
```

sortir

```
[demo-py38] ~ $ conda deactivate
[base] ~ $
```

la version de python

```
[base] ~ $ python --version
Python 3.7.6
```

on n'a pas perturbé l'environnement de départ

```
[base] ~ $ pip show numpy
Name: numpy
Version: 1.18.1
```

pour détruire l'environnement en question

```
[base] ~ $ conda env remove -n demo-py38
```

```
Remove all packages in environment /Users/tparment/miniconda3/envs/demo-py38:
```

6.22 w6-s9-c2-outils-annexes

Outils périphériques

6.22.1 Compléments - niveau intermédiaire

Pour conclure le tronc commun de ce cours Python, nous allons très rapidement citer quelques outils qui ne sont pas nécessairement dans la bibliothèque standard, mais qui sont très largement utilisés dans l'écosystème python.

Il s'agit d'une liste non exhaustive bien entendu.

Debugging

Pour le debugging, la bibliothèque standard s'appelle `pdb`. Typiquement pour mettre un breakpoint on écrit :

```
def foo(n):  
    n = n ** 2  
    # pour mettre un point d'arrêt  
    import pdb  
    pdb.set_trace()  
    # la suite de foo()  
    return n / 10
```

Je vous signale d'ailleurs qu'à partir de Python 3.7, il est recommandé d'utiliser la nouvelle fonction built-in `breakpoint()` qui rend le même service.

Une fois qu'on a dit ça, votre IDE dispose certainement d'une fonctionnalité pour faire ça avec la souris de manière plus flexible.

Tests

Le module `unittest` de la bibliothèque standard fournit des fonctionnalités de base pour écrire des tests unitaires.

Je vous signale également des outils comme `pytest` (et `nosetests`), qui ne sont pas dans la distribution standard, mais qui enrichissent les capacités de `unittest` pour en rendre l'utilisation quotidienne (beaucoup) plus fluide.

Parmi les fonctionnalités fournies par un framework de test comme `pytest` :

- découverte automatique des tests : en respectant les règles de nommage, il vous suffit de lancer `pytest` à la racine de votre projet pour exécuter tous les tests. Vous pouvez en une commande lancer tous les tests, tous ceux d'un dossier ou d'un fichier, ou juste un test-case
- exécution automatique des tests : vous vous concentrez sur le fait d'écrire ce qui doit se passer, le framework se charge du reste
- fixtures pour expliciter plus simplement comment mettre le système dans un état contrôlé
- et beaucoup d'autres choses...

Pour approfondir le sujet, [cette note très courte](#) explicite les conventions pour la découverte des tests, et les bonnes pratiques.

Personnellement je préfère mettre les tests dans un dossier séparé du package, mais bon, tous les goûts sont dans la nature apparemment :)

Documentation

Le standard de fait dans ce domaine est clairement une combinaison basée sur

- l'outil **sphinx**, qui permet de générer la documentation à partir du source, avec
 - des plugins pour divers sous-formats dans les docstrings,
 - un système de templating,
 - et de nombreuses autres possibilités ;
- **readthedocs.io** qui est une plateforme ouverte pour l'hébergement des documentations, elle-même facilement intégrable avec un repository type **github.io**,

Pour vous donner une idée du résultat, je vous invite à consulter un module de ma facture :

- les sources sur github sur <https://github.com/parmentelat/asynciojobs>, et notamment le sous-répertoire **sphinx**,
- et la documentation en ligne sur <http://asynciojobs.readthedocs.io/>.

Lint

Au delà de la vérification automatique de la présentation du code (PEP8), il existe un outil **pylint** qui fait de l'analyse de code source en vue de détecter des erreurs le plus tôt possible dans le cycle de développement.

En quelques mots, ma recommandation à ce sujet est que :

- tout d'abord, et comme dans tous les langages en fait, il est très utile de faire passer systématiquement son code dans un linter de ce genre ;
- idéalement on ne devrait commiter que du code qui passe cette étape ;
- cependant, il y a un petit travail de filtrage à faire au démarrage, car pylint détecte plusieurs centaines de sortes d'erreurs, du coup il convient de passer un moment à configurer l'outil pour qu'il en ignore certaines.

Dès que vous commencez à travailler sur des projets sérieux, vous devez utiliser un éditeur qui intègre et exécute automatiquement **pylint**. On peut notamment recommander **PyCharm**.

Type hints

Je voudrais citer enfin l'outil **mypy** qui est un complément crucial dans la mise en oeuvre des type hints.

Comme on l'a vu en Semaine 4 dans la séquence consacrée aux type hints, et en tous cas jusque Python-3.6, les annotations de typage que vous insérez éventuellement dans votre code sont complètement ignorées de l'interpréteur.

Elles sont par contre analysées par l'outil **mypy** qui fournit une sorte de couche supplémentaire de linter et permet de détecter, ici encore, les éventuelles erreurs qui peuvent résulter notamment d'une mauvaise utilisation de telle ou telle librairie.

Conclusion

À nouveau cette liste n'est pas exhaustive, elle s'efforce simplement de guider vos premiers pas dans cet écosystème.

Je vous invite à creuser de votre côté les différents aspects qui, parmi cette liste, vous semblent les plus intéressants pour votre usage.

6.23

w6-s9-x1-exos-en-vrac

Quelques sujets d'exercice en vrac

```
[1]: # ceci permet de recharger les modules
      # lorsqu'ils ont été modifiés en dehors du notebook

      # pour commodité lors du développement des exercices

      %load_ext autoreload
      %autoreload 2
```

The autoreload extension is already loaded. To reload it, use:

```
%reload_ext autoreload
```

Niveaux de difficulté La plupart de (tous ?) ces exercices sont inspirés d'énoncés trouvés sur <https://codewars.com/>; dans ces cas-là j'indique la référence, ainsi que la difficulté affichée sur codewars; [l'échelle est à la japonaise](#), 1 kyu c'est très difficile et 8 kyu c'est très simple.

6.23.1 Trouver la somme

inspiré de <https://www.codewars.com/kata/52c31f8e6605bcc646000082> (6 kyu)

On cherche dans une liste deux nombres (à des indices différents) dont la somme est fixée

- en entrée : une liste de nombres, et une valeur cible
- en sortie : les index dans la liste de deux nombres dont la somme est égale à la valeur cible; on doit retourner un tuple, trié dans l'ordre croissant.

Hypothèses : on admet (pas besoin de le vérifier donc) que les entrées sont correctes, c'est-à-dire ne contiennent que des nombres et qu'il existe une solution.

Unicité : n'importe quelle solution est valable en cas de solutions multiples; toutefois pour des raisons techniques, la correction automatique ne teste votre code que sur des entrées où la solution est unique.

```
[2]: # charger l'exercice et afficher un exemple

      from corrections.exo_two_sum import exo_two_sum
      exo_two_sum.example()
```

GridBox(children=(HTML(value='appel', _dom_classes=('header

```
[3]: # indice : il y a peut-être des choses utiles dans ce module
      # import itertools

      def two_sum(data, target):
          ...
```

```
[ ]: exo_two_sum.correction(two_sum)

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

6.23.2 Plus grande distance

inspiré de <https://www.codewars.com/kata/5442e4fc7fc447653a0000d5> (6 kyu)

- en entrée : une liste d'objets (vous pouvez vous restreindre à des entiers pour commencer)
- en sortie : un entier qui décrit la plus grande distance (en termes d'indices) entre deux occurrences du même objet dans la liste ;
- si aucun objet n'est présent en double, retournez 0.

```
[4]: from corrections.exo_longest_gap import exo_longest_gap

      exo_longest_gap.example()
```

GridBox(children=(HTML(value='appel', _dom_classes=('header

```
[5]: # votre code
      def longest_gap(liste):
          ...
```

```
[ ]: exo_longest_gap.correction(longest_gap)

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

6.23.3 Meeting

inspiré de <https://www.codewars.com/kata/59df2f8f08c6cec835000012> (6 kyu)

Je vous invite à lire l'énoncé directement sur codewars.

```
[6]: from corrections.exo_meeting import exo_meeting

      exo_meeting.example()
```

GridBox(children=(HTML(value='appel', _dom_classes=('header


```
[7]: def meeting(string):
      ...
```

```
[ ]: exo_meeting.correction(meeting)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

6.23.4 Évaluateur d'expression postfix (6 kyu)

Une fonction `postfix_eval` prend en entrée une chaîne qui décrit une opération à faire sur des entiers, en utilisant la notation polonaise postfixée, où on écrit par exemple `10 20 +` pour ajouter 10 et 20 ; cette notation est aussi appelée la notation polonaise inverse.

Les opérandes sont tous des entiers ; on demande de supporter les 4 opérations `+` `-` `*` et `/` (division entière), la calculatrice ne manipule donc que des entiers.

Lorsque la chaîne est mal formée, vous devez renvoyer une des trois chaînes suivantes :

- `error-syntax` si on ne peut pas comprendre l'entrée,
- `error-empty-stack`, si on essaie de faire une opération mais que l'on n'a pas les deux opérandes nécessaires,
- `error-unfinished`, si on détecte des opérandes non utilisés.

```
[8]: # charger l'exercice et afficher un exemple
      from corrections.exo_postfix_eval import exo_postfix_eval
      exo_postfix_eval.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[9]: def postfix_eval(chaine):
      ...
```

```
[ ]: exo_postfix_eval.correction(postfix_eval)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

6.23.5 Évaluateur d'expression postfix typé (5 kyu)

```
[10]: from fractions import Fraction
```

Écrire une variante de `postfix_eval` qui accepte en deuxième argument un type de nombre parmi `int`, `float`, ou `Fraction`, de sorte que la calculatrice utilise ce type pour faire ses calculs.

indice : attention au cas de la division, qui doit se comporter selon le type comme une division entière (comme dans `postfix_eval`), ou comme une division usuelle si le type le permet.

```
[11]: # charger l'exercice et afficher un exemple

from corrections.exo_postfix_eval import exo_postfix_eval_typed
exo_postfix_eval_typed.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[12]: # votre code
def postfix_eval_typed(chaine, result_type):
    ...
```

```
[ ]: exo_postfix_eval_typed.correction(postfix_eval_typed)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

6.23.6 exercice - niveau avancé

On se propose d'écrire une classe pour représenter les polynômes :

- avec un constructeur qui prend en argument les coefficients en commençant par les degrés les plus élevés; ainsi par exemple
 - `Polynomial()` aussi bien que `Polynomial(0)` représentent le polynôme nul,
 - `Polynomial(3, 2, 1)` représente $3X^2 + 2X + 1$, et
 - `Polynomial(3, 0, 1, 0, 0)` représente $3X^4 + X^2$
- avec un attribut **degree** pour accéder au degré
- avec une méthode **derivative()** pour calculer le polynôme dérivé
- qui sait s'additionner, se multiplier et se comparer avec `==`
- et qu'on peut appeler (autrement dit qui est un callable)
ce qui signifie qu'on peut écrire par exemple

```
P = Polynomial(3, 2, 1)
P(10) == 321
```

Note importante

Le système de correction automatique a besoin également que votre classe définisse son comportement vis-à-vis de `repr()` ; regardez les exemples pour voir la représentation choisie.

```
[1]: from corrections.cls_polynomial import exo_polynomial
exo_polynomial.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:small;"\>scénario 1</span>', _dom_classes=('he
```

```
[2]: # votre code
```

```
class Polynomial:

    def __init__(self, *coefs):
        ...
```

```
[ ]: # correction
exo_polynomial.correction(Polynomial)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

```
[3]: # peut-être utile pour debugger ?
P00 = Polynomial()
P0 = Polynomial(0)
P1 = Polynomial(1)
P = Polynomial(3, 2, 1)
Q = Polynomial(1, 2)
R = Polynomial(3, 8, 5, 2)
```

```
[4]: P0 == P00 == P0 * P1
```

```
[4]: False
```

```
[5]: P * Q == R
```

```
-----

TypeError                                Traceback (most recent call last)

<ipython-input-5-51ebae22f908> in <module>
----> 1 P * Q == R

TypeError: unsupported operand type(s) for *: 'Polynomial' and 'Polynomial'
```

```
[6]: P(10)
```

```
-----

TypeError                                Traceback (most recent call last)

<ipython-input-6-69a867584a90> in <module>
----> 1 P(10)

TypeError: 'Polynomial' object is not callable
```

6.23.7 exercice - niveau avancé

On se propose d'écrire une classe pour représenter les températures :

- avec un constructeur qui prend exactement un paramètre nommé
 - `Temperature(kelvin=0)`
 - aussi bien que `Temperature(celsius=0)`
- avec un attribut `kelvin` et un attribut `celsius` pour accéder en lecture ou en écriture à la valeur actuelle de la température, dans l'échelle choisie.

Note importante

Le système de correction automatique a besoin également que votre classe définisse son comportement vis-à-vis de `repr()` ; regardez les exemples pour voir la représentation choisie.

Pour simplifier cet aspect de l'exercice, on a choisi d'arrondir à $0^{\circ}\text{C} = 273^{\circ}\text{K}$, et de ne manipuler que des valeurs entières.

```
[1]: from corrections.cls_temperature import exo_temperature
     exo_temperature.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>scénario 1</span>', _dom_classes=('h
```

```
[2]: # votre code

class Temperature:

    K = 273

    def __init__(self, kelvin=None, celsius=None):
        ...

    def __repr__(self):
        return f"xxx"
```

```
[ ]: # correction
     exo_temperature.correction(Temperature)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

```
[3]: # peut-être utile pour debugger ?
     K00 = Temperature()
     K0 = Temperature(kelvin=0)
```

```
[4]: K0 == K00
```

```
[4]: False
```

```
[5]: C0 = Temperature(celsius=0)
     C00 = Temperature(kelvin=Temperature.K)
```

```
[6]: C0 == C00
```

```
[6]: False
```

```
[7]: C0
```

```
[7]: xxx
```

```
[8]: C00
```

```
[8]: xxx
```

6.24 w6-s9-x4-primes

Exercice - niveau avancé

6.24.1 itérateurs et générateurs

Tous les exercices de ce notebook vous demandent d'écrire des fonctions qui construisent des itérateurs.

```
[1]: import itertools
```

6.24.2 1. Nombres premiers

On vous demande d'écrire un générateur qui énumère les nombres premiers.

Naturellement il existe de nombreuses bibliothèques pour cela, mais on vous demande ici d'écrire votre propre algorithme, même s'il est naïf.

```
[2]: from corrections.gen_primes import exo_primes
     exo_primes.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

Le générateur ne s'arrête donc jamais, c'est un générateur infini comme `itertools.count()`. Le système de correction automatique est capable d'extraire certaines parties du flux du générateur, avec une convention voisine de `range()` et/ou du slicing.

Ainsi par exemple le deuxième jeu de test, sous-titré $1 \rightarrow 5 / 2$, va retenir les éléments énumérés par le générateur aux itérations 1, 3 et 5 - en commençant bien sûr à compter à 0.

NOTES

- Évidemment, il vous faut retourner un itérateur, et la correction automatique vérifiera ce point.
- Notez aussi que, lorsqu'on cherche à déterminer si n est entier, on a nécessairement déjà fait ce travail sur tous les entiers plus petits que n . Il est donc tentant, et fortement recommandé, de profiter de cette information pour accélérer l'algorithme.

- Si votre algorithme est très lent ou faux, vous pouvez perdre le kernel (en français noyau), c'est-à-dire qu'il calcule pendant très longtemps (ou pour toujours); dans ces cas-là, la marge gauche indique `In [*]:` et l'étoile n'est jamais remplacée par un chiffre. Il vous faut alors interrompre votre kernel; pour cela utilisez le menu Kernel qui a des options pour interrompre ou redémarrer le kernel courant; les raccourcis clavier `i i` et `o o` permettent aussi d'interrompre et redémarrer le noyau.

```
[3]: # à vous de jouer

def primes():
    # vous DEVEZ retourner un itérateur
    # bien sûr count() n'est pas une bonne réponse...
    return itertools.count()
```

```
[ ]: # pour corriger votre code
exo_primes.correction(primes)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

zone de debug

```
[4]: # à toutes fins utiles

MAX = 10

iterator = primes()

for index, prime in enumerate(itertools.islice(iterator, MAX)):
    print(f"{index} -> {prime}")
```

```
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
6 -> 6
7 -> 7
8 -> 8
9 -> 9
```

6.24.3 2. Les carrés des nombres premiers

On veut à présent énumérer les carrés des nombres premiers

NOTE il y a au moins deux façons triviales de parvenir au résultat.

```
[5]: from corrections.gen_primes import exo_prime_squares
     exo_prime_squares.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[6]: # à vous

     def prime_squares():
         ...
```

```
[ ]: exo_prime_squares.correction(prime_squares)

     # NOTE
     # auto-exec-for-latex has skipped execution of this cell
```

6.24.4 3. Combinaisons d'itérateurs

On vous demande d'écrire un itérateur qui énumère des couples :

- en première position, on veut trouver les nombres premiers, mais avec un décalage : les cinq premiers tuples contiennent 1, puis le sixième contient 2, et à partir de là les nombres premiers ;
- en deuxième position, les carrés des nombres premiers, sans décalage :

NOTE

Il peut être tentant de créer deux instances de l'itérateur `primes()` ; toutefois c'est cet objet qui demande le plus de temps de calcul, aussi on vous suggère de réfléchir, en option, à une solution qui ne crée qu'un seul exemplaire de cet itérateur.

```
[7]: from corrections.gen_primes import exo_prime_legos
     exo_prime_legos.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[8]: # à vous de jouer

     def prime_legos():
         ...
```

```
[ ]: exo_prime_legos.correction(prime_legos)

     # NOTE
     # auto-exec-for-latex has skipped execution of this cell
```

6.24.5 4. Les n -ièmes nombres premiers, avec n premier

On vous demande d'implémenter un itérateur qui renvoie les n -ièmes nombres premiers, mais seulement pour n premier.

Ainsi comme `primes()` retourne la suite

indice	premier
0	2
1	3
2	5
3	7
4	11
5	13
6	17
7	19

on veut que `prime_th_primes` retourne la suite

indice	premier
0	5
1	7
2	13
3	19

```
[9]: # ce qui est illustré sur cet exemple calculé, qui va un peu plus loin

from corrections.gen_primes import exo_prime_th_primes
exo_prime_th_primes.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[10]: # À vous de jouer

def prime_th_primes():
    # souvenez-vous que vous devez retourner un itérateur
    return itertools.count()
```

```
[ ]: # pour corriger votre code
exo_prime_th_primes.correction(prime_th_primes)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```


6.24.6 Exercice - niveau intermédiaire

On se propose d'écrire une classe de redirection ; typiquement, une instance de cette classe est capable de résoudre n'importe quel attribut car son travail est de sous-traiter le travail à quelqu'un d'autre.

L'exercice vient en deux versions ; dans la première on considère seulement des attributs de données, dans la seconde, un attribut inconnu se comporte comme une méthode.

Première version : attributs de donnée

Dans cette première version, on veut créer une instance à partir de laquelle on peut accéder à n'importe quel attribut dont le nom est un nom de variable légal en Python.

Et la valeur de l'attribut est dérivé de son nom en :

- le mettant en minuscules, et
- en remplaçant les éventuels `_` par un signe `-`

```
[1]: from corrections.cls_redirectors import exo_redirector1
     exo_redirector1.example()
```

GridBox(children=(HTML(value='scénario 1', _dom_classes=('h

```
[2]: # votre code pour la classe Redirector1
     class Redirector1:

         def __repr__(self):
             return "redirector"

         ...
```

```
[ ]: # pour la corriger
     #
     exo_redirector1.correction(Redirector1)

     # NOTE
     # auto-exec-for-latex has skipped execution of this cell
```

6.24.7 Exercice - niveau avancé

deuxième version : méthodes

Cette fois-ci on considère qu'un attribut manquant est une méthode ; pour fixer les idées on décide que :

- ces méthodes prennent toutes un seul argument - en sus de l'objet qui les reçoit ;
- chacune de ces méthodes retourne une simple chaîne, qui contient des morceaux provenant de :
 - l'objet redirecteur lui-même,
 - le nom de la méthode,
 - l'argument passé à la méthode, qui donc est unique.

Ce mécanisme est illustré sur les exemples suivants, avec deux méthodes `foo` et `bar` ; par contre le test automatique exercera votre code avec des noms de méthodes aléatoires.

```
[3]: from corrections.cls_redirectors import exo_redirector2
     exo_redirector2.example()
```

GridBox(children=(HTML(value='scénario 1', _dom_classes=('h

```
[4]: # à vous de jouer
     class Redirector2:
         def __init__(self, id):
             self.id = id
         def __repr__(self):
             return f"Redirector2({self.id})"
```

```
[ ]: # pour corriger
     exo_redirector2.correction(Redirector2)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

6.24.8 application

Un exemple en vraie grandeur de ce type de mécanisme est proposé dans la librairie standard Python : <https://docs.python.org/3/library/xmlrpc.client.html#serverproxy-objects>.

Typiquement la classe générique `ServerProxy` ressemble à notre deuxième version de l'exercice, en ce sens qu'elle n'a aucune connaissance de l'API distante à laquelle elle est connectée ; autrement dit l'ensemble des méthodes effectivement proposées par la classe `ServerProxy` est totalement inconnu au moment où on écrit le code de cette classe.

Chapitre 7

L'écosystème data science Python

7.1 w7-s01-c1-installation

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 dans votre interpréteur Python (par exemple, IPython) sans erreur :

```
[1]: import numpy as np
import matplotlib.pyplot as plt
```

```
[2]: 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 (sous Windows, `cmd.exe` avec les droits d'administrateur) 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
```

7.2 w7-s02-c1-dimension1

numpy en dimension 1

7.2.1 Complément - niveau basique

Comme on l'a vu dans la vidéo, **numpy** est une bibliothèque qui offre un type supplémentaire par rapport aux types de base Python : le tableau, qui s'appelle en anglais **array** (en fait techniquement, **ndarray**, pour n-dimension array).

Bien que techniquement ce type ne fasse pas partie des types de base de Python, il est extrêmement puissant, et surtout beaucoup plus efficace que les types de base, dès lors qu'on manipule des données qui ont la bonne forme, ce qui est le cas dans un grand nombre de domaines.

Aussi, si vous utilisez une bibliothèque de calcul scientifique, la quasi totalité des objets que vous serez amenés à manipuler seront des tableaux **numpy**.

Dans cette première partie nous allons commencer avec des tableaux à une dimension, et voir comment les créer et les manipuler.

```
[1]: import numpy as np
```

Création à partir de données

np.array

On peut créer un tableau numpy à partir d'une liste - ou plus généralement un itérable - avec la fonction **np.array** comme ceci :

```
[2]: array = np.array([12, 25, 32, 55])  
array
```

```
[2]: array([12, 25, 32, 55])
```

Attention : une erreur commune au début consiste à faire ceci, qui ne marche pas :

```
[3]: try:
      array = np.array(1, 2, 3, 4)
    except Exception as e:
      print(f"OOPS, {type(e)}, {e}")
```

OOPS, <class 'ValueError'>, only 2 non-keyword arguments accepted

Ça marche aussi à partir d'un itérable :

```
[4]: builtin_range = np.array(range(10))
      builtin_range
```

```
[4]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Création d'intervalles

np.arange

Sauf que dans ce cas précis on préférera utiliser directement la méthode **arange** de **numpy** :

```
[5]: numpy_range = np.arange(10)
      numpy_range
```

```
[5]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Avec l'avantage qu'avec cette méthode on peut donner des bornes et un pas d'incrément qui ne sont pas entiers :

```
[6]: numpy_range_f = np.arange(1.0, 2.0, 0.1)
      numpy_range_f
```

```
[6]: array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
```

np.linspace

Aussi et surtout, lorsqu'on veut créer un intervalle dont on connaît les bornes, il est souvent plus facile d'utiliser **linspace**, qui crée un intervalle un peu comme **arange**, mais on lui précise un nombre de points plutôt qu'un pas :

```
[7]: X = np.linspace(0., 10., 50)
      X
```

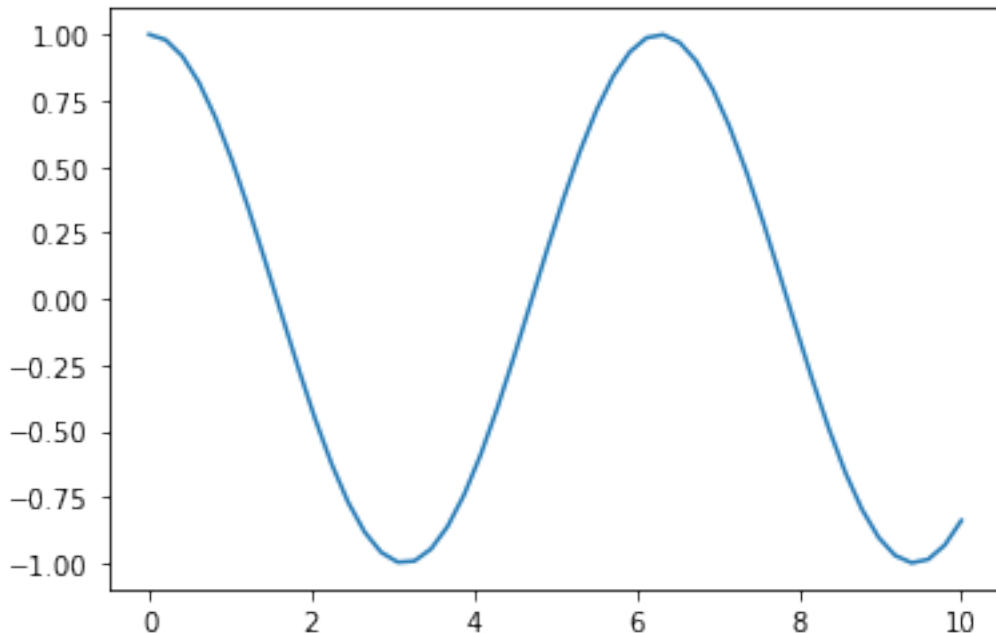
```
[7]: array([ 0.         ,  0.20408163,  0.40816327,  0.6122449 ,  0.81632653,
            1.02040816,  1.2244898 ,  1.42857143,  1.63265306,  1.83673469,
            2.04081633,  2.24489796,  2.44897959,  2.65306122,  2.85714286,
            3.06122449,  3.26530612,  3.46938776,  3.67346939,  3.87755102,
            4.08163265,  4.28571429,  4.48979592,  4.69387755,  4.89795918,
            5.10204082,  5.30612245,  5.51020408,  5.71428571,  5.91836735,
            6.12244898,  6.32653061,  6.53061224,  6.73469388,  6.93877551,
            7.14285714,  7.34693878,  7.55102041,  7.75510204,  7.95918367,
```

```
8.16326531, 8.36734694, 8.57142857, 8.7755102 , 8.97959184,
9.18367347, 9.3877551 , 9.59183673, 9.79591837, 10. ])
```

Vous remarquez que les 50 points couvrent à intervalles réguliers l'espace compris entre 0 et 10 inclusivement. Notons que 50 est aussi le nombre de points par défaut. Cette fonction est très utilisée lorsqu'on veut dessiner une fonction entre deux bornes, on a déjà eu l'occasion de le faire :

```
[8]: import matplotlib.pyplot as plt
     %matplotlib inline
     plt.ion()
```

```
[9]: # il est d'usage d'ajouter un point-virgule à la fin de la dernière ligne
     # si on ne le fait pas (essayez..), on obtient l'affichage d'une ligne
     # de bruit qui n'apporte rien
     Y = np.cos(X)
     plt.plot(X, Y);
```



Programmation vectorielle

Attardons-nous un petit peu :

- nous avons créé un tableau X de 50 points qui couvrent l'intervalle [0..10] de manière uniforme,
- et nous avons calculé un tableau Y de 50 valeurs qui correspondent aux cosinus des valeurs de X.

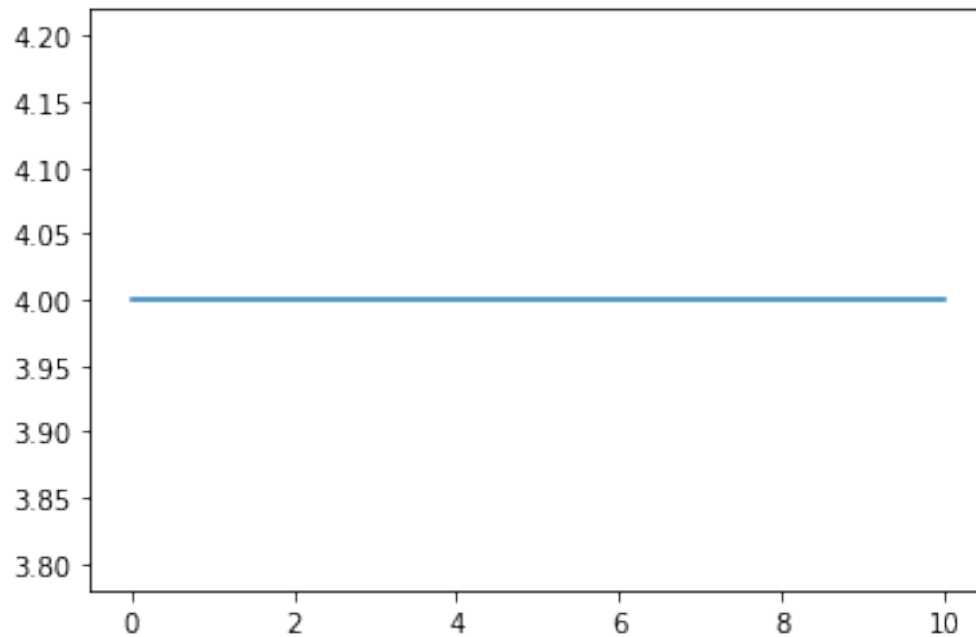
Remarquez qu'on a fait ce premier calcul sans même savoir comment accéder aux éléments d'un tableau. Vous vous doutez bien qu'on va accéder aux éléments d'un tableau à base d'index, on le verra bien sûr, mais on n'en a pas eu besoin ici.

En fait en `numpy` on passe son temps à écrire des expressions dont les éléments sont des tableaux, et cela produit des opérations membre à membre, comme on vient de le voir avec cosinus.

Ainsi pour tracer la fonction $x \rightarrow \cos^2(x) + \sin^2(x) + 3$ on fera tout simplement :

```
[10]: # l'énorme majorité du temps, on écrit avec numpy
# des expressions qui impliquent des tableaux
# exactement comme si c'était des nombres
Z = np.cos(X)**2 + np.sin(X)**2 + 3

plt.plot(X, Z);
```



C'est le premier réflexe qu'il faut avoir avec les tableaux numpy : on a vu que les compréhensions et les expressions génératrices permettent de s'affranchir des boucles du genre :

```
out_data = []
for x in in_data:
    out_data.append(une_fonction(x))
```

on a vu en python natif qu'on ferait plutôt :

```
out_data = (une_fonction(x) for x in in_data)
```

Eh bien en fait, en numpy, on doit penser encore plus court :

```
out_data = une_fonction(in_data)
```

ou en tous les cas une expression qui fait intervenir `in_data` comme un tout, sans avoir besoin d'accéder à ses éléments.

ufunc

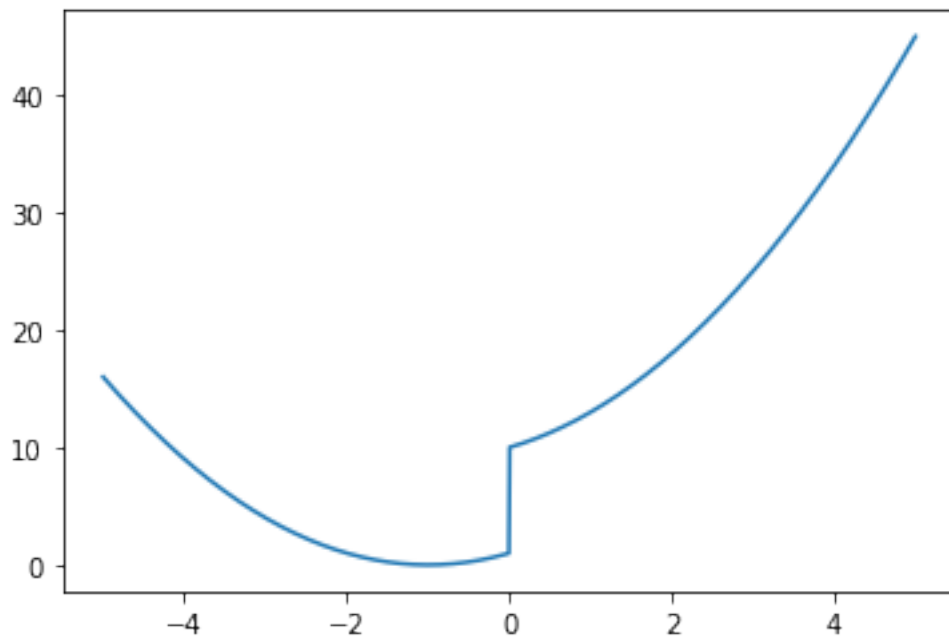
Le mécanisme général qui applique une fonction à un tableau est connu sous le terme de Universal function, ou `ufunc`, ça peut vous être utile avec les moteurs de recherche.

Voyez notamment la liste des [fonctionnalités disponibles sous cette forme dans numpy](#).

Je vous signale également un utilitaire qui permet, sous forme de décorateur, de passer d'une fonction scalaire à une `ufunc` :

```
[11]: # le décorateur np.vectorize vous permet
      # de facilement transformer une opération scalaire
      # en opération vectorielle
      # je choisis à dessein une fonction définie par morceaux
      @np.vectorize
      def scalar_function(x):
          return x**2 + 2*x + (1 if x <= 0 else 10)
```

```
[12]: # je choisis de prendre beaucoup de points
      # à cause de la discontinuité
      X = np.linspace(-5, 5, 1000)
      Y = scalar_function(X)
      plt.plot(X, Y);
```



Conclusion

Pour conclure ce complément d'introduction, ce style de programmation - que je vais décider d'appeler programmation vectorielle de manière un peu impropre - est au cœur de `numpy`, et n'est bien entendu pas limitée aux tableaux de dimension 1, comme on va le voir dans la suite.

7.3 w7-s02-c2-dtype

Type d'un tableau **numpy**

7.3.1 Complément - niveau intermédiaire

Nous allons voir dans ce complément ce qu'il faut savoir sur le type d'un tableau **numpy**.

```
[1]: import numpy as np
```

Dans ce complément nous allons rester en dimension 1 :

```
[2]: a = np.array([1, 2, 4, 8])
```

Toutes les cellules ont le même type

Comme on l'a vu dans la vidéo, les très bonnes performances que l'on peut obtenir en utilisant un tableau **numpy** sont liées au fait que le tableau est homogène : toutes les cellules du tableau possèdent le même type :

```
[3]: # pour accéder au type d'un tableau
a.dtype
```

```
[3]: dtype('int64')
```

Vous voyez que dans notre cas, le système a choisi pour nous un type entier ; selon les entrées on peut obtenir :

```
[4]: # si je mets au moins un flottant
f = np.array([1, 2, 4, 8.])
f.dtype
```

```
[4]: dtype('float64')
```

```
[5]: # et avec un complexe
c = np.array([1, 2, 4, 8j])
c.dtype
```

```
[5]: dtype('complex128')
```

Et on peut préciser le type que l'on veut si cette heuristique ne nous convient pas :

```
[6]: # je choisis explicitement mon dtype
c2 = np.array([1, 2, 4, 8], dtype=np.complex64)
c2.dtype
```

```
[6]: dtype('complex64')
```

Pertes de précision

Une fois que le type est déterminé, on s'expose à de possibles pertes de précision, comme d'habitude :

```
[7]: a, a.dtype
```

```
[7]: (array([1, 2, 4, 8]), dtype('int64'))
```

```
[8]: # a est de type entier  
# je vais perdre le 0.14  
a[0] = 3.14  
a
```

```
[8]: array([3, 2, 4, 8])
```

Types disponibles

Voyez la liste complète <https://docs.scipy.org/doc/numpy/user/basics.types.html>.

Ce qu'il faut en retenir :

- vous pouvez choisir entre `bool`, `int`, `uint` (entier non signé), `float` et `complex`;
- ces types ont diverses tailles pour vous permettre d'optimiser la mémoire réellement utilisée;
- ces types existent en tant que tels (hors de tableaux).

```
[9]: # un entier sur 1 seul octet, c'est possible !  
np_1 = np.int8(1)  
# l'équivalent en Python natif  
py_1 = 1
```

```
[10]: # il y a bien égalité  
np_1 == py_1
```

```
[10]: True
```

```
[11]: # mais bien entendu ce ne sont pas les mêmes objets  
np_1 is py_1
```

```
[11]: False
```

Du coup, on peut commencer à faire de très substantielles économies de place; imaginez que vous souhaitez manipuler une image d'un million de pixels en noir et blanc sur 256 niveaux de gris; j'en profite pour vous montrer `np.zeros` (qui fait ce que vous pensez) :

```
[12]: # pur Python  
from sys import getsizeof  
pure_py = [0 for i in range(10**6)]  
getsizeof(pure_py)
```

```
[12]: 8697464
```

```
[13]: # numpy
      num_py = np.zeros(10**6, dtype=np.int8)
      getsizeof(num_py)
```

```
[13]: 1000096
```

Je vous signale enfin l'attribut `itemsize` qui vous permet d'obtenir la taille en octets occupée par chacune des cellules, et qui correspond donc en gros au nombre qui apparaît dans `dtype`, mais divisé par huit :

```
[14]: a.dtype
```

```
[14]: dtype('int64')
```

```
[15]: a.itemsize
```

```
[15]: 8
```

```
[16]: c.dtype
```

```
[16]: dtype('complex128')
```

```
[17]: c.itemsize
```

```
[17]: 16
```

7.4 w7-s03-c1-shape

Forme d'un tableau **numpy**

Nous allons voir dans ce complément comment créer des tableaux en plusieurs dimensions et manipuler la forme (`shape`) des tableaux.

```
[1]: import numpy as np
```

Un exemple

Nous avons vu précédemment comment créer un tableau **numpy** de dimension 1 à partir d'un simple itérable, nous allons à présent créer un tableau à 2 dimensions, et pour cela nous allons utiliser une liste imbriquée :

```
[2]: d2 = np.array([[11, 12, 13], [21, 22, 23]])
      d2
```

```
[2]: array([[11, 12, 13],
          [21, 22, 23]])
```

Ce premier exemple va nous permettre de voir les différents attributs de tous les tableaux **numpy**.

L'attribut **shape**

Tous les tableaux **numpy** possèdent un attribut **shape** qui retourne, sous la forme d'un tuple, les dimensions du tableau :

```
[3]: # la forme (les dimensions) du tableau  
d2.shape
```

```
[3]: (2, 3)
```

Dans le cas d'un tableau en 2 dimensions, cela correspond donc à lignes x colonnes.

On peut facilement changer de forme

Comme on l'a vu dans la vidéo, un tableau est en fait une vue vers un bloc de données. Aussi il est facile de changer la dimension d'un tableau - ou plutôt, de créer une autre vue vers les mêmes données :

```
[4]: # l'argument qu'on passe à reshape est le tuple  
# qui décrit la nouvelle *shape*  
v2 = d2.reshape((3, 2))  
v2
```

```
[4]: array([[11, 12],  
          [13, 21],  
          [22, 23]])
```

Et donc, ces deux tableaux sont deux vues vers la même zone de données ; ce qui fait qu'une modification sur l'un se répercute dans l'autre :

```
[5]: # je change un tableau  
d2[0][0] = 100  
d2
```

```
[5]: array([[100, 12, 13],  
          [ 21, 22, 23]])
```

```
[6]: # ça se répercute dans l'autre  
v2
```

```
[6]: array([[100, 12],  
          [ 13, 21],  
          [ 22, 23]])
```

Les attributs liés à la forme

Signalons par commodité les attributs suivants, qui se dérivent de **shape** :

```
[7]: # le nombre de dimensions  
d2.ndim
```

```
[7]: 2
```

```
[8]: # vrai pour tous les tableaux
len(d2.shape) == d2.ndim
```

```
[8]: True
```

```
[9]: # le nombre de cellules
d2.size
```

```
[9]: 6
```

```
[10]: # vrai pour tous les tableaux
# une façon compliquée de dire
# une chose toute simple :
# la taille est le produit
# des dimensions
from operator import mul
from functools import reduce
d2.size == reduce(mul, d2.shape, 1)
```

```
[10]: True
```

Lorsqu'on utilise `reshape`, il faut bien sûr que la nouvelle forme soit compatible :

```
[11]: try:
        d2.reshape((3, 4))
    except Exception as e:
        print(f"OOPS {type(e)} {e}")
```

```
OOPS <class 'ValueError'> cannot reshape array of size 6 into shape (3,4)
```

Dimensions supérieures

Vous pouvez donc deviner comment on construit des tableaux en dimensions supérieures à 2, il suffit d'utiliser un attribut `shape` plus élaboré :

```
[12]: shape = (2, 3, 4)
size = reduce(mul, shape)

# vous vous souvenez de arange
data = np.arange(size)
```

```
[13]: d3 = data.reshape(shape)
d3
```

```
[13]: array([[[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]],

            [[12, 13, 14, 15],
              [16, 17, 18, 19],
              [20, 21, 22, 23]]])
```

Cet exemple vous permet de voir qu'en dimensions supérieures la forme est toujours :

$n_1 \times n_2 \times \dots \times \text{lignes} \times \text{colonnes}$

Enfin, ce que je viens de dire est arbitraire, dans le sens où, bien entendu, vous pouvez décider d'interpréter les tableaux comme vous voulez.

Mais en termes au moins de l'impression par `print`, il est logique de voir que l'algorithme d'impression balaye le tableau de manière mécanique comme ceci :

```
for i in range(2):
    for j in range(3):
        for k in range(4):
            array[i][j][k]
```

Et c'est pourquoi vous obtenez la présentation suivante avec des tableaux de dimensions plus grandes :

```
[14]: # la même chose avec plus de dimensions
shape = (2, 3, 4, 5)
size = reduce(mul, shape) # le produit des 4 nombres dans shape
size
```

[14]: 120

```
[15]: data = np.arange(size)

# ce tableau est visualisé
# à base de briques de base
# de 4 lignes et 5 colonnes
d4 = data.reshape(shape)
d4
```

```
[15]: array([[[[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14],
               [15, 16, 17, 18, 19]],

              [[ 20, 21, 22, 23, 24],
               [ 25, 26, 27, 28, 29],
               [ 30, 31, 32, 33, 34],
               [ 35, 36, 37, 38, 39]],

              [[ 40, 41, 42, 43, 44],
               [ 45, 46, 47, 48, 49],
               [ 50, 51, 52, 53, 54],
               [ 55, 56, 57, 58, 59]]],

           [[[ 60, 61, 62, 63, 64],
               [ 65, 66, 67, 68, 69],
               [ 70, 71, 72, 73, 74],
               [ 75, 76, 77, 78, 79]],

              [[ 80, 81, 82, 83, 84],
               [ 85, 86, 87, 88, 89],
               [ 90, 91, 92, 93, 94]]],

           ...])
```

```
[ 95, 96, 97, 98, 99],
[[100, 101, 102, 103, 104],
 [105, 106, 107, 108, 109],
 [110, 111, 112, 113, 114],
 [115, 116, 117, 118, 119]]])
```

Vous voyez donc qu'avec la forme :

```
2, 3, 4, 5
```

cela vous donne l'impression que vous avez comme brique de base des tableaux qui ont :

```
4 lignes
5 colonnes
```

Et souvenez-vous que vous pouvez toujours insérer un 1 n'importe où dans la forme, puisque ça ne modifie pas la taille qui est le produit des dimensions :

```
[16]: d2.shape
```

```
[16]: (2, 3)
```

```
[17]: d2
```

```
[17]: array([[100, 12, 13],
           [ 21, 22, 23]])
```

```
[18]: d2.reshape(2, 1, 3)
```

```
[18]: array([[[100, 12, 13],
              [ 21, 22, 23]])])
```

```
[19]: d2.reshape(2, 3, 1)
```

```
[19]: array([[[100],
              [ 12],
              [ 13]],
            [[ 21],
              [ 22],
              [ 23]])])
```

Ou même :

```
[20]: d2.reshape((1, 2, 3))
```

```
[20]: array([[[100, 12, 13],
              [ 21, 22, 23]])])
```

```
[21]: d2.reshape((1, 1, 1, 1, 2, 3))
```

```
[21]: array([[[[[[100, 12, 13],
                [ 21, 22, 23]]]]]])
```

Résumé des attributs

Voici un résumé des attributs des tableaux `numpy` :

attribut	signification	exemple
<code>shape</code>	tuple des dimensions	<code>(3, 5, 7)</code>
<code>ndim</code>	nombre dimensions	<code>3</code>
<code>size</code>	nombre d'éléments	<code>3 * 5 * 7</code>
<code>dtype</code>	type de chaque élément	<code>np.float64</code>
<code>itemsize</code>	taille en octets d'un élément	<code>8</code>

Divers

Je vous signale enfin, à titre totalement anecdotique cette fois, l'existence de la méthode `ravel` qui vous permet d'aplatir n'importe quel tableau :

```
[22]: d2
```

```
[22]: array([[100, 12, 13],
            [ 21, 22, 23]])
```

```
[23]: d2.ravel()
```

```
[23]: array([100, 12, 13, 21, 22, 23])
```

```
[24]: # il y a d'ailleurs aussi flatten qui fait
      # quelque chose de semblable
      d2.flatten()
```

```
[24]: array([100, 12, 13, 21, 22, 23])
```

7.5 w7-s03-c2-initialisation

Création de tableaux

7.5.1 Complément - niveau basique

Passons rapidement en revue quelques méthodes pour créer des tableaux `numpy`.

```
[1]: import numpy as np
```


Non initialisé : `np.empty`

La méthode la plus efficace pour créer un tableau `numpy` consiste à faire l'allocation de la mémoire mais sans l'initialiser :

```
[2]: memory = np.empty(dtype=np.int8,
                        shape=(1_000, 1_000))
```

J'en profite pour attirer votre attention sur l'impression des gros tableaux où l'on s'efforce de vous montrer les coins :

```
[3]: print(memory)
```

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

Il se peut que vous voyiez ici des valeurs particulières ; selon votre OS, il y a une probabilité non nulle que vous ne voyiez ici que des zéros. C'est un peu comme avec les dictionnaires qui, depuis la version 3.6, peuvent donner l'impression de conserver l'ordre dans lequel les clés ont été créées. Ici c'est un peu la même chose, vous ne devez pas écrire un programme qui repose sur le fait que `np.empty` retourne un tableau garni de zéros (utilisez alors `np.zeros`, que l'on va voir tout de suite).

Tableaux constants

On peut aussi créer et initialiser un tableau avec `np.zeros` et `np.ones` :

```
[4]: zeros = np.zeros(dtype=np.complex128, shape=(1_000, 100))
      print(zeros)
```

```
[[0.+0.j 0.+0.j 0.+0.j ... 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j ... 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j ... 0.+0.j 0.+0.j 0.+0.j]
 ...
 [0.+0.j 0.+0.j 0.+0.j ... 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j ... 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j ... 0.+0.j 0.+0.j 0.+0.j]]
```

```
[5]: fours = 4 * np.ones(dtype=float, shape=(8, 8))
      fours
```

```
[5]: array([[4., 4., 4., 4., 4., 4., 4., 4.],
           [4., 4., 4., 4., 4., 4., 4., 4.],
           [4., 4., 4., 4., 4., 4., 4., 4.],
           [4., 4., 4., 4., 4., 4., 4., 4.],
           [4., 4., 4., 4., 4., 4., 4., 4.],
           [4., 4., 4., 4., 4., 4., 4., 4.],
           [4., 4., 4., 4., 4., 4., 4., 4.],
           [4., 4., 4., 4., 4., 4., 4., 4.]])
```

Progression arithmétique : **arange**

En guise de rappel, avec **arange** on peut créer des tableaux de valeurs espacées d'une valeur constante. Ça ressemble donc un peu au **range** de Python natif :

```
[6]: np.arange(4)
```

```
[6]: array([0, 1, 2, 3])
```

```
[7]: np.arange(1, 5)
```

```
[7]: array([1, 2, 3, 4])
```

Sauf qu'on peut y passer un pas qui n'est pas entier :

```
[8]: np.arange(5, 7, .5)
```

```
[8]: array([5. , 5.5, 6. , 6.5])
```

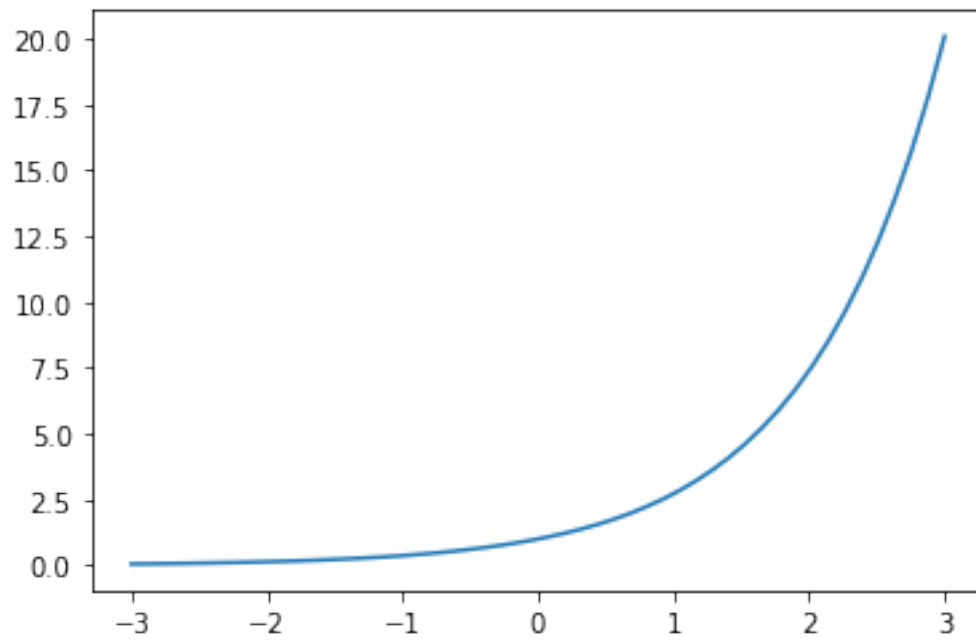
Progression arithmétique : **linspace**

Mais bien souvent, plutôt que de préciser le pas entre deux valeurs, on préfère préciser le nombre de points ; et aussi inclure la deuxième borne. C'est ce que fait **linspace**, c'est très utile pour modéliser une fonction sur un intervalle ; on a déjà vu des exemples de ce genre :

```
[9]: %matplotlib inline
import matplotlib.pyplot as plt
plt.ion()
```

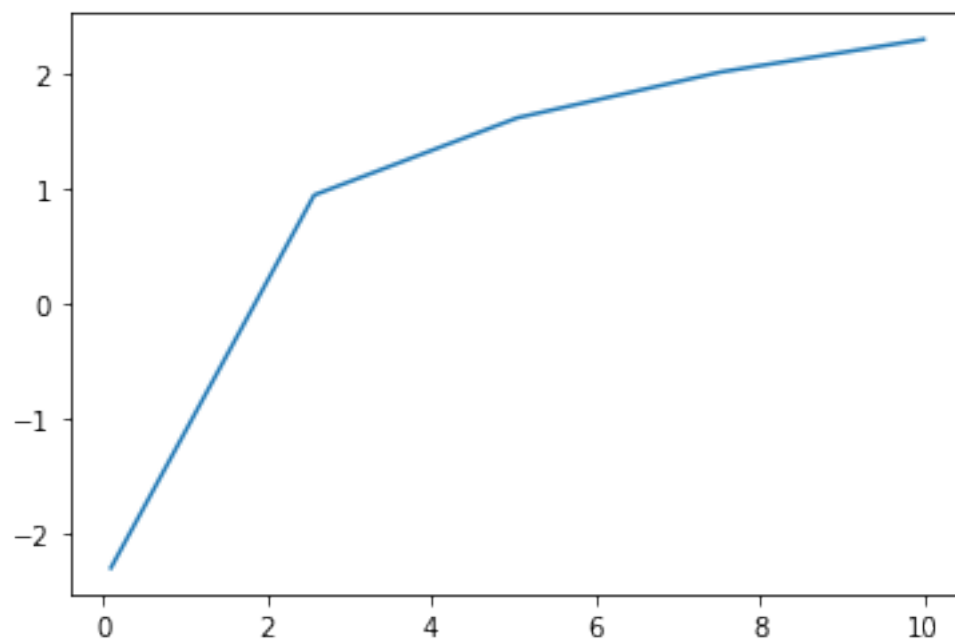
```
[10]: X = np.linspace(-3., +3.)
      Y = np.exp(X)

      plt.plot(X, Y);
```



```
[11]: # par défaut linspace crée 50 points  
      # avec moins de points
```

```
X = np.linspace(1/10, 10, num = 5)  
plt.plot(X, np.log(X));
```



Pour des intervalles en progression géométrique, voyez `np.geomspace`.

Multi-dimensions : **indices**

La méthode `np.indices` se comporte un peu comme `arange` mais pour plusieurs directions ; voyons ça sur un exemple :

```
[12]: ix, iy = np.indices((3, 5))
```

```
[13]: ix
```

```
[13]: array([[0, 0, 0, 0, 0],
            [1, 1, 1, 1, 1],
            [2, 2, 2, 2, 2]])
```

```
[14]: iy
```

```
[14]: array([[0, 1, 2, 3, 4],
            [0, 1, 2, 3, 4],
            [0, 1, 2, 3, 4]])
```

Cette fonction s'appelle **indices** parce qu'elle produit des tableaux (ici 2 car on lui a passé une **shape** à deux dimensions) qui contiennent, à la case (i, j) , i (pour le premier tableau) ou j pour le second.

Ainsi, si vous voulez construire un tableau de taille $(2, 4)$ dans lequel, par exemple :

```
tab[i, j] = 200*i + 2*j + 50
```

Vous n'avez qu'à faire :

```
[15]: ix, iy = np.indices((2, 4))
      tab = 200*ix + 2*iy + 50
      tab
```

```
[15]: array([[ 50,  52,  54,  56],
            [250, 252, 254, 256]])
```

Multi-dimensions : **meshgrid**

Si vous voulez créer un tableau un peu comme avec `linspace`, mais en plusieurs dimensions : imaginez par exemple que vous voulez tracer une fonction à deux entrées :

$$f : (x, y) \longrightarrow \cos(x) + \cos^2(y)$$

Sur un pavé délimité par :

$$x \in [-\pi, +\pi], y \in [3\pi, 5\pi]$$

Il vous faut donc créer un tableau, disons de 50 x 50 points, qui réalise un maillage uniforme de ce pavé, et pour ça vous pouvez utiliser **meshgrid**. Pour commencer :

```
[16]: # on fabrique deux tableaux qui échantillonnent
      # de manière uniforme les intervalles en X et en Y
      # on prend un pas de 10 dans les deux sens, ça nous donnera
      # 100 points pour couvrir l'espace carré qui nous intéresse
```

```
Xticks, Yticks = (np.linspace(-np.pi, np.pi, num=10),
                  np.linspace(3*np.pi, 5*np.pi, num=10))
```

Avec `meshgrid`, on va créer deux tableaux, qui sont respectivement les (100) X et les (100) Y de notre maillage :

```
[17]: # avec meshgrid on les croise
      # ça fait comme un produit cartésien,
      # en extrayant les X et les Y du résultat

      X, Y = np.meshgrid(Xticks, Yticks)

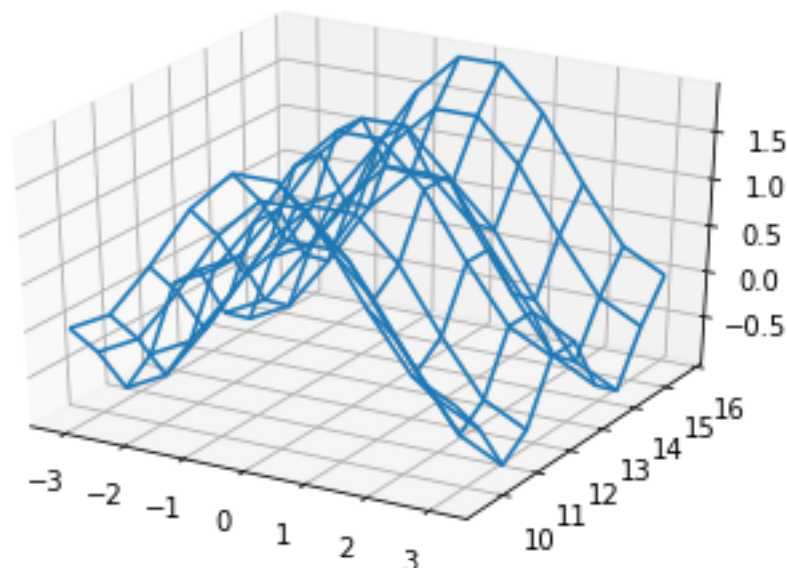
      # chacun des deux est donc de taille 10 x 10
      X.shape, Y.shape
```

```
[17]: ((10, 10), (10, 10))
```

Que peut-on faire avec ça ? Eh bien, en fait, on a tout ce qu'il nous faut pour afficher notre fonction :

```
[18]: # un tableau 10 x 10 qui contient les images de f()
      # sur les points de la grille
      Z = np.cos(X) + np.cos(Y)**2
```

```
[19]: from mpl_toolkits.mplot3d import Axes3D
      fig = plt.figure()
      ax = fig.add_subplot(111, projection='3d')
      ax.plot_wireframe(X, Y, Z);
```



Je vous laisse vous convaincre qu'il est facile d'écrire `np.indices` à partir de `np.meshgrid` et `np.arange`.

7.6 w7-s05-c1-broadcasting

Le broadcasting

```
[1]: import numpy as np
```

7.6.1 Complément - niveau intermédiaire

Lorsque l'on a parlé de programmation vectorielle, on a vu que l'on pouvait écrire quelque chose comme ceci :

```
[2]: X = np.linspace(0, 2 * np.pi)
      Y = np.cos(X) + np.sin(X) + 2
```

Je vous fais remarquer que dans cette dernière ligne on combine :

- deux tableaux de mêmes tailles - quand on ajoute `np.cos(X)` avec `np.sin(X)` ;
- un tableau avec un scalaire - quand on ajoute 2 au résultat.

En fait, le broadcasting est ce qui permet :

- d'unifier le sens de ces deux opérations ;
- de donner du sens à des cas plus généraux, où on fait des opérations entre des tableaux qui ont des tailles différentes, mais assez semblables pour que l'on puisse tout de même les combiner.

7.6.2 Exemples en 2D

Nous allons commencer par quelques exemples simples, avant de généraliser le mécanisme. Pour commencer, nous nous donnons un tableau de base :

```
[3]: a = 100 * np.ones((3, 5), dtype=np.int32)
      print(a)
```

```
[[100 100 100 100 100]
 [100 100 100 100 100]
 [100 100 100 100 100]]
```

Je vais illustrer le broadcasting avec l'opération `+`, mais bien entendu ce mécanisme est à l'œuvre dès que vous faites des opérations entre deux tableaux qui n'ont pas les mêmes dimensions.

Pour commencer, je vais donc ajouter à mon tableau de base un scalaire :

Broadcasting entre les dimensions `(3, 5)` et `(1,)`

```
[4]: print(a)
```

```
[[100 100 100 100 100]
 [100 100 100 100 100]
 [100 100 100 100 100]]
```

```
[5]: b = 3
      print(b)
```

3

Lorsque j'ajoute ces deux tableaux, c'est comme si j'avais ajouté à **a** la différence :

```
[6]: # pour élaborer c
      c = a + b
      print(c)
```

```
[[103 103 103 103 103]
 [103 103 103 103 103]
 [103 103 103 103 103]]
```

```
[7]: # c'est comme si j'avais
      # ajouté à a ce terme-ci
      print(c - a)
```

```
[[3 3 3 3 3]
 [3 3 3 3 3]
 [3 3 3 3 3]]
```

C'est un premier cas particulier de broadcasting dans sa version extrême.

Le scalaire **b**, qui est en l'occurrence considéré comme un tableau dont le **shape** vaut (1,), est dupliqué dans les deux directions jusqu'à obtenir ce tableau uniforme de taille (5, 3) et qui contient un 3 partout.

Et c'est ce tableau, qui est maintenant de la même taille que **a**, qui est ajouté à **a**.

Je précise que cette explication est du domaine du modèle pédagogique ; je ne dis pas que l'implémentation va réellement allouer un second tableau, bien évidemment on peut optimiser pour éviter cette construction inutile.

Broadcasting (3, 5) et (5,)

Voyons maintenant un cas un peu moins évident. Je peux ajouter à mon tableau de base une ligne, c'est-à-dire un tableau de taille (5,). Voyons cela :

```
[8]: print(a)
```

```
[[100 100 100 100 100]
 [100 100 100 100 100]
 [100 100 100 100 100]]
```

```
[9]: b = np.arange(1, 6)
      print(b)
```

```
[1 2 3 4 5]
```

```
[10]: b.shape
```

[10]: (5,)

Ici encore, je peux ajouter les deux termes :

```
[11]: # je peux ici encore
      # ajouter les tableaux
      c = a + b
      print(c)
```

```
[[101 102 103 104 105]
 [101 102 103 104 105]
 [101 102 103 104 105]]
```

```
[12]: # et c'est comme si j'avais
      # ajouté à a ce terme-ci
      print(c - a)
```

```
[[1 2 3 4 5]
 [1 2 3 4 5]
 [1 2 3 4 5]]
```

Avec le même point de vue que tout à l'heure, on peut se dire qu'on a d'abord transformé (broadcasté) le tableau `b` :

depuis la dimension (5,)

vers la dimension (3, 5)

```
[13]: # départ
      print(b)
```

```
[1 2 3 4 5]
```

```
[14]: # arrivée
      print(c - a)
```

```
[[1 2 3 4 5]
 [1 2 3 4 5]
 [1 2 3 4 5]]
```

Vous commencez à mieux voir comment ça fonctionne ; s'il existe une direction dans laquelle on peut "tirer" les données pour faire coïncider les formes, on peut faire du broadcasting. Et ça marche dans toutes les directions, comme on va le voir tout de suite.

Broadcasting (3, 5) et (3, 1)

Au lieu d'ajouter à `a` une ligne, on peut lui ajouter une colonne, pourvu qu'elle ait la même taille que les colonnes de `a` :

```
[15]: print(a)
```

```
[[100 100 100 100 100]
 [100 100 100 100 100]]
```



```
[100 100 100 100 100]]
```

```
[16]: b = np.arange(1, 4).reshape(3, 1)
      print(b)
```

```
[[1]
 [2]
 [3]]
```

Voyons comment se passe le broadcasting dans ce cas-là :

```
[17]: c = a + b
      print(c)
```

```
[[101 101 101 101 101]
 [102 102 102 102 102]
 [103 103 103 103 103]]
```

```
[18]: print(c - a)
```

```
[[1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]]
```

Vous voyez que tout se passe exactement de la même façon que lorsqu'on avait ajouté une simple ligne, on a cette fois “tiré” la colonne dans la direction des lignes, pour passer :

depuis la dimension (3, 1)

vers la dimension (3, 5)

```
[19]: # départ
      print(b)
```

```
[[1]
 [2]
 [3]]
```

```
[20]: # arrivée
      print(c - a)
```

```
[[1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]]
```

Broadcasting (3, 1) et (1, 5)

Nous avons maintenant tous les éléments en main pour comprendre un exemple plus intéressant, où les deux tableaux ont des formes pas vraiment compatibles à première vue :

```
[21]: col = np.arange(1, 4).reshape((3, 1))
      print(col)
```

```
[[1]
 [2]
 [3]]
```

```
[22]: line = 100 * np.arange(1, 6)
      print(line)
```

```
[100 200 300 400 500]
```

Grâce au broadcasting, on peut additionner ces deux tableaux pour obtenir ceci :

```
[23]: m = col + line
      print(m)
```

```
[[101 201 301 401 501]
 [102 202 302 402 502]
 [103 203 303 403 503]]
```

Remarquez qu'ici les deux entrées ont été étirées pour atteindre une dimension commune.

Et donc pour illustrer le broadcasting dans ce cas, tout se passe comme si on avait :

transformé la colonne (3, 1)

en tableau (3, 5)

```
[24]: print(col)
```

```
[[1]
 [2]
 [3]]
```

```
[25]: print(col + np.zeros(5, dtype=np.int))
```

```
[[1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]]
```

et transformé la ligne (1, 5)

en tableau (3, 5)

```
[26]: print(line)
```

```
[100 200 300 400 500]
```

```
[27]: print(line + np.zeros(3, dtype=np.int).reshape((3, 1)))
```

```
[[100 200 300 400 500]
 [100 200 300 400 500]
 [100 200 300 400 500]]
```

avant d'additionner terme à terme ces deux tableaux 3 x 5.

7.6.3 En dimensions supérieures

Pour savoir si deux tableaux peuvent être compatibles via broadcasting, il faut comparer leurs formes. Je commence par vous donner des exemples. Ici encore quand on mentionne l'addition, cela vaut pour n'importe quel opérateur binaire.

Exemples de dimensions compatibles

```
A  15 x 3 x 5
B  15 x 1 x 5
A+B 15 x 3 x 5
```

Cas de l'ajout d'un scalaire :

```
A  15 x 3 x 5
B              1
A+B 15 x 3 x 5
```

```
A  15 x 3 x 5
B      3 x 5
A+B 15 x 3 x 5
```

```
A  15 x 3 x 5
B      3 x 1
A+B 15 x 3 x 5
```

Exemples de dimensions non compatibles

Deux lignes de longueurs différentes :

```
A  3
B  4
```

Un cas plus douteux :

```
A      2 x 1
B  8 x 4 x 3
```

Comme vous le voyez sur tous ces exemples :

- on peut ajouter A et B lorsqu'il existe une dimension C qui "étire" à la fois celle de A et celle de B ;
- on le voit sur le dernier exemple, mais on ne peut broadcaster que de 1 vers n ; lorsque $p > 1$ divise n , on ne peut pas broadcaster de p vers n , comme on pourrait peut-être l'imaginer.

Comme c'est un cours de Python, plutôt que de formaliser ça sous une forme mathématique - je vous le laisse en exercice - je vais vous proposer plutôt une fonction Python qui détermine si deux tuples sont des `shape` compatibles de ce point de vue.

```
[28]: # le module broadcasting n'est pas standard
      # c'est moi qui l'ai écrit pour illustrer le cours
      from broadcasting import compatible, compatible2
```

```
[29]: # on peut dupliquer selon un axe
      compatible((15, 3, 5), (15, 1, 5))
```

```
[29]: (15, 3, 5)
```

```
[30]: # ou selon deux axes
      compatible((15, 3, 5), (5,))
```

```
[30]: (15, 3, 5)
```

```
[31]: # c'est bien clair que non
      compatible((2,), (3,))
```

```
[31]: False
```

```
[32]: # on ne peut pas passer de 2 à 4
      compatible((1, 2), (2, 4))
```

```
[32]: False
```

7.7 w7-s05-c2-indexing-slicing

Index et slices

7.7.1 Complément - niveau basique

```
[1]: import numpy as np
      import matplotlib.pyplot as plt
      %matplotlib inline
      plt.ion()
```

J'espère que vous êtes à présent convaincus qu'il est possible de faire énormément de choses avec **numpy** en faisant des opérations entre tableaux, et sans aller référencer un par un les éléments des tableaux, ni faire de boucle **for**.

Il est temps maintenant de voir que l'on peut aussi manipuler les tableaux **numpy** avec des index.

Indexation par des entiers et tuples

La façon la plus naturelle d'utiliser un tableau est habituellement à l'aide des indices. On peut aussi bien sûr accéder aux éléments d'un tableau **numpy** par des indices :

```
[2]: # une fonction qui crée un tableau
      # tab[i, j] = i + 10 * j
      def background(n):
```

```
i = np.arange(n)
j = i.reshape((n, 1))
return i + 10 * j
```

```
[3]: a5 = background(5)
      print(a5)
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
```

Avec un seul index on obtient naturellement une ligne :

```
[4]: a5[1]
```

```
[4]: array([10, 11, 12, 13, 14])
```

```
[5]: # que l'on peut à nouveau indexer
      a5[1][2]
```

```
[5]: 12
```

```
[6]: # ou plus simplement indexer par un tuple
      a5[1, 2]
```

```
[6]: 12
```

```
[7]: # naturellement on peut affecter une case
      # individuellement
      a5[2][1] = 221
      a5[3, 2] += 300
      print(a5)
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 221 22 23 24]
 [30 31 332 33 34]
 [40 41 42 43 44]]
```

```
[8]: # ou toute une ligne
      a5[1] = np.arange(100, 105)
      print(a5)
```

```
[[ 0  1  2  3  4]
 [100 101 102 103 104]
 [20 221 22 23 24]
 [30 31 332 33 34]
 [40 41 42 43 44]]
```

```
[9]: # et on on peut aussi changer
      # toute une ligne par broadcasting
```

```
a5[4] = 400
print(a5)
```

```
[[ 0  1  2  3  4]
 [100 101 102 103 104]
 [ 20 221  22  23  24]
 [ 30  31 332  33  34]
 [400 400 400 400 400]]
```

7.8 w7-s05-c2-indexing-slicing

Slicing

Grâce au slicing on peut aussi référencer une colonne :

```
[10]: a5 = background(5)
      print(a5)
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
```

```
[11]: a5[:, 3]
```

```
[11]: array([ 3, 13, 23, 33, 43])
```

C'est un tableau à une dimension, mais vous pouvez tout de même modifier la colonne par une affectation :

```
[12]: a5[:, 3] = range(300, 305)
      print(a5)
```

```
[[ 0  1  2 300  4]
 [10 11 12 301 14]
 [20 21 22 302 24]
 [30 31 32 303 34]
 [40 41 42 304 44]]
```

Ou, ici également bien sûr, par broadcasting :

```
[13]: # on affecte un scalaire à une colonne
      a5[:, 2] = 200
      print(a5)
```

```
[[ 0  1 200 300  4]
 [10 11 200 301 14]
 [20 21 200 302 24]
 [30 31 200 303 34]
 [40 41 200 304 44]]
```

```
[14]: # ou on ajoute un scalaire à une colonne
a5[:, 4] += 400
print(a5)
```

```
[[ 0  1 200 300 404]
 [10 11 200 301 414]
 [20 21 200 302 424]
 [30 31 200 303 434]
 [40 41 200 304 444]]
```

Les slices peuvent prendre une forme générale :

```
[15]: a8 = background(8)
print(a8)
```

```
[[ 0  1  2  3  4  5  6  7]
 [10 11 12 13 14 15 16 17]
 [20 21 22 23 24 25 26 27]
 [30 31 32 33 34 35 36 37]
 [40 41 42 43 44 45 46 47]
 [50 51 52 53 54 55 56 57]
 [60 61 62 63 64 65 66 67]
 [70 71 72 73 74 75 76 77]]
```

```
[16]: # toutes les lignes de rang 1, 4, 7
a8[1::3]
```

```
[16]: array([[10, 11, 12, 13, 14, 15, 16, 17],
           [40, 41, 42, 43, 44, 45, 46, 47],
           [70, 71, 72, 73, 74, 75, 76, 77]])
```

```
[17]: # toutes les colonnes de rang 1, 5, 9
a8[:, 1::4]
```

```
[17]: array([[ 1,  5],
           [11, 15],
           [21, 25],
           [31, 35],
           [41, 45],
           [51, 55],
           [61, 65],
           [71, 75]])
```

```
[18]: # et on peut bien sûr les modifier
a8[:, 1::4] = 0
print(a8)
```

```
[[ 0  0  2  3  4  0  6  7]
 [10  0 12 13 14  0 16 17]
 [20  0 22 23 24  0 26 27]
 [30  0 32 33 34  0 36 37]
 [40  0 42 43 44  0 46 47]
 [50  0 52 53 54  0 56 57]
 [60  0 62 63 64  0 66 67]
 [70  0 72 73 74  0 76 77]]
```

Du coup, le slicing peut servir à extraire des blocs :

```
[19]: # un bloc au hasard dans a8
      print(a8[5:8, 2:5])
```

```
[[52 53 54]
 [62 63 64]
 [72 73 74]]
```

newaxis

On peut utiliser également le symbole spécial `np.newaxis` en conjonction avec un slice pour “décaler” les dimensions :

```
[20]: X = np.arange(1, 7)
      print(X)
```

```
[1 2 3 4 5 6]
```

```
[21]: X.shape
```

```
[21]: (6,)
```

```
[22]: Y = X[:, np.newaxis]
      print(Y)
```

```
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
```

```
[23]: Y.shape
```

```
[23]: (6, 1)
```

Et ainsi de suite :

```
[24]: Z = Y[:, np.newaxis]
      Z
```

```
[24]: array([[1],
           [2],
           [3],
           [4],
           [5],
           [6]])
```



```
[25]: Z.shape
```

```
[25]: (6, 1, 1)
```

De cette façon, par exemple, en combinant le slicing pour créer X et Y, et le broadcasting pour créer leur somme, je peux créer facilement la table de tous les tirages de 2 dés à 6 faces :

```
[26]: dice2 = X + Y
print(dice2)
```

```
[[ 2  3  4  5  6  7]
 [ 3  4  5  6  7  8]
 [ 4  5  6  7  8  9]
 [ 5  6  7  8  9 10]
 [ 6  7  8  9 10 11]
 [ 7  8  9 10 11 12]]
```

Ou tous les tirages à trois dés :

```
[27]: dice3 = X + Y + Z
print(dice3)
```

```
[[[ 3  4  5  6  7  8]
  [ 4  5  6  7  8  9]
  [ 5  6  7  8  9 10]
  [ 6  7  8  9 10 11]
  [ 7  8  9 10 11 12]
  [ 8  9 10 11 12 13]]

 [[ 4  5  6  7  8  9]
  [ 5  6  7  8  9 10]
  [ 6  7  8  9 10 11]
  [ 7  8  9 10 11 12]
  [ 8  9 10 11 12 13]
  [ 9 10 11 12 13 14]]

 [[ 5  6  7  8  9 10]
  [ 6  7  8  9 10 11]
  [ 7  8  9 10 11 12]
  [ 8  9 10 11 12 13]
  [ 9 10 11 12 13 14]
  [10 11 12 13 14 15]]

 [[ 6  7  8  9 10 11]
  [ 7  8  9 10 11 12]
  [ 8  9 10 11 12 13]
  [ 9 10 11 12 13 14]
  [10 11 12 13 14 15]
  [11 12 13 14 15 16]]

 [[ 7  8  9 10 11 12]
  [ 8  9 10 11 12 13]
  [ 9 10 11 12 13 14]
  [10 11 12 13 14 15]
  [11 12 13 14 15 16]
  [12 13 14 15 16 17]]
```

```
[[ 8  9 10 11 12 13]
 [ 9 10 11 12 13 14]
 [10 11 12 13 14 15]
 [11 12 13 14 15 16]
 [12 13 14 15 16 17]
 [13 14 15 16 17 18]]]
```

J'en profite pour introduire un utilitaire qui n'a rien à voir, mais avec `np.unique`, vous pourriez calculer le nombre d'occurrences dans le tableau, et ainsi calculer les probabilités d'apparition de tous les nombres entre 3 et 18 :

```
[28]: np.unique(dice3, return_counts=True)
```

```
[28]: (array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18]),
       array([ 1,  3,  6, 10, 15, 21, 25, 27, 27, 25, 21, 15, 10,  6,  3,  1]))
```

Différences avec les listes

Avec l'indexation et le slicing, on peut créer des tableaux qui sont des vues sur des fragments d'un tableau; on peut également déformer leur dimension grâce à `newaxis`; on peut modifier ces fragments, en utilisant un scalaire, un tableau, ou une slice sur un autre tableau. Les possibilités sont infinies.

Il est cependant utile de souligner quelques différences entre les tableaux `numpy` et, les listes natives, pour ce qui concerne les indexations et le slicing.

On ne peut pas changer la taille d'un tableau avec le slicing. La taille d'un objet `numpy` est par définition constante; cela signifie qu'on ne peut pas, par exemple, modifier sa taille totale avec du slicing; c'est à mettre en contraste avec, si vous vous souvenez :

Listes

```
[29]: # on peut faire ceci
liste = [0, 1, 2]
liste[1:2] = [100, 102, 102]
liste
```

```
[29]: [0, 100, 102, 102, 2]
```

Tableaux

```
[30]: # on ne peut pas faire cela
array = np.array([0, 1, 2])
try:
    array[1:2] = np.array([100, 102, 102])
except Exception as e:
    print(f"OOPS, {type(e)}, {e}")
```

```
OOPS, <class 'ValueError'>, could not broadcast input array from shape (3)
into shape (1)
```

On peut modifier un tableau en modifiant une slice

Une slice sur un objet `numpy` renvoie une vue sur un extrait du tableau, et en changeant la vue on change le tableau ; ici encore c'est à mettre en contraste avec ce qui se passe sur les listes :

Listes

```
[31]: # une slice d'une liste est une shallow copy  
liste = [0, 1, 2]  
liste[1:2]
```

```
[31]: [1]
```

```
[32]: # en modifiant la slice,  
# on ne modifie pas la liste  
liste[1:2][0] = 999999  
liste
```

```
[32]: [0, 1, 2]
```

Tableaux

```
[33]: # une slice d'un tableau numpy est un extrait du tableau  
array = np.array([0, 1, 2])  
array[1:2]
```

```
[33]: array([1])
```

```
[34]: array[1:2][0] = 100  
array
```

```
[34]: array([ 0, 100,  2])
```

7.9

w7-s05-c3-operations-logiques

Opérations logiques

7.9.1 Complément - niveau basique

Même si les tableaux contiennent habituellement des nombres, on peut être amenés à faire des opérations logiques et du coup à manipuler des tableaux de booléens. Nous allons voir quelques éléments à ce sujet.

```
[1]: import numpy as np
```

Opérations logiques

On peut faire des opérations logiques entre tableaux exactement comme on fait des opérations arithmétiques.

On va partir de deux tableaux presque identiques. J'en profite pour vous signaler qu'on peut copier un tableau avec, tout simplement, `np.copy` :

```
[2]: a = np.arange(25).reshape(5, 5)
      print(a)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

```
[3]: b = np.copy(a)
      b[2, 2] = 1000
      print(b)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 1000 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

Dans la lignée de ce qu'on a vu jusqu'ici en matière de programmation vectorielle, une opération logique va ici aussi nous retourner un tableau de la même taille :

```
[4]: # la comparaison par == ne nous
      # retourne pas directement un booléen
      # mais un tableau de la même taille que a et b
      print(a == b)
```

```
[[ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True False  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]]
```

all et any

Si votre intention est de vérifier que les deux tableaux sont entièrement identiques, utilisez `np.all` - et non pas le built-in natif `all` de Python - qui va vérifier que tous les éléments du tableau sont vrais :

```
[5]: # oui
      np.all(a == a)
```

```
[5]: True
```

```
[6]: # oui
      np.all(a == b)
```

[6]: False

```
[7]: # oui
      # on peut faire aussi bien
      # np.all(x)
      # ou
      # x.all()
      (a == a).all()
```

[7]: True

```
[8]: # par contre : non !
      # ceci n'est pas conseillé
      # même si ça peut parfois fonctionner
      try:
          all(a == a)
      except Exception as e:
          print(f'OOPS {type(e)} {e}')
```

OOPS <class 'ValueError'> The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()

C'est bien sûr la même chose pour **any** qui va vérifier qu'il y a au moins un élément vrai. Comme en Python natif, un nombre qui est nul est considéré comme faux :

```
[9]: np.zeros(5).any()
```

[9]: False

```
[10]: np.ones(5).any()
```

[10]: True

Masques

Mais en général, c'est rare qu'on ait besoin de consolider de la sorte un booléen sur tout un tableau, on utilise plutôt les tableaux logiques comme des masques, pour faire ou non des opérations sur un autre tableau.

J'en profite pour introduire une fonction de **matplotlib** qui s'appelle **imshow** et qui permet d'afficher une image :

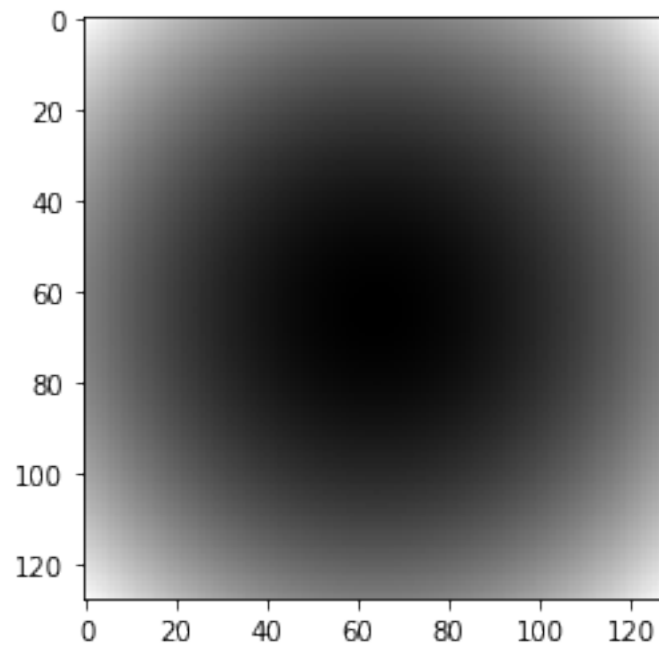
```
[11]: import matplotlib.pyplot as plt
      %matplotlib inline
      plt.ion()
```

```
[12]: # construisons un disque centré au milieu de l'image

width = 128
center = width / 2

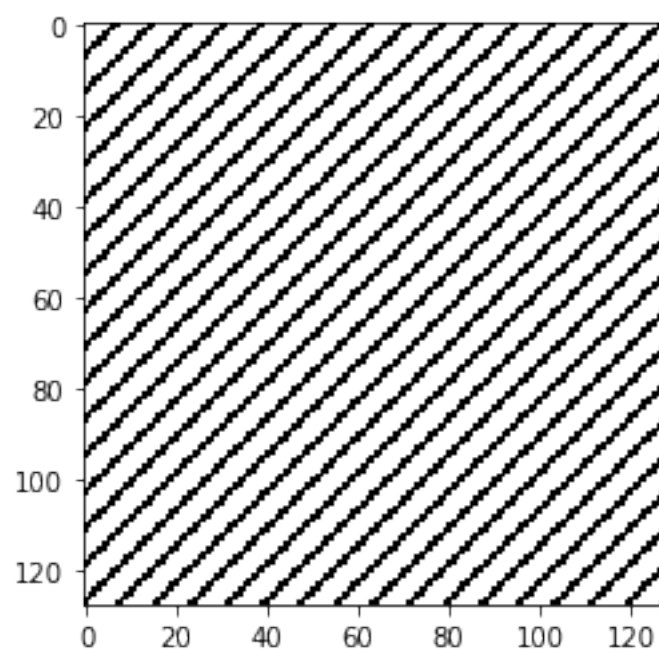
ix, iy = np.indices((width, width))
image = (ix-center)**2 + (iy-center)**2
```

```
# pour afficher l'image en niveaux de gris  
plt.imshow(image, cmap='gray');
```



Maintenant je peux créer un masque qui produise des rayures en diagonale, donc selon la valeur de $(i+j)$.
Par exemple :

```
[13]: # pour faire des rayures  
# de 6 pixels de large  
rayures = (ix + iy) % 8 <= 5  
plt.imshow(rayures, cmap='gray');
```



```
[14]: # en fait c'est bien sûr
      # un tableau de booléens
      print(rayures)
```

```
[[ True  True  True ...  True False False]
 [ True  True  True ... False False  True]
 [ True  True  True ... False  True  True]
 ...
 [ True False False ...  True  True  True]
 [False False  True ...  True  True  True]
 [False  True  True ...  True  True False]]
```

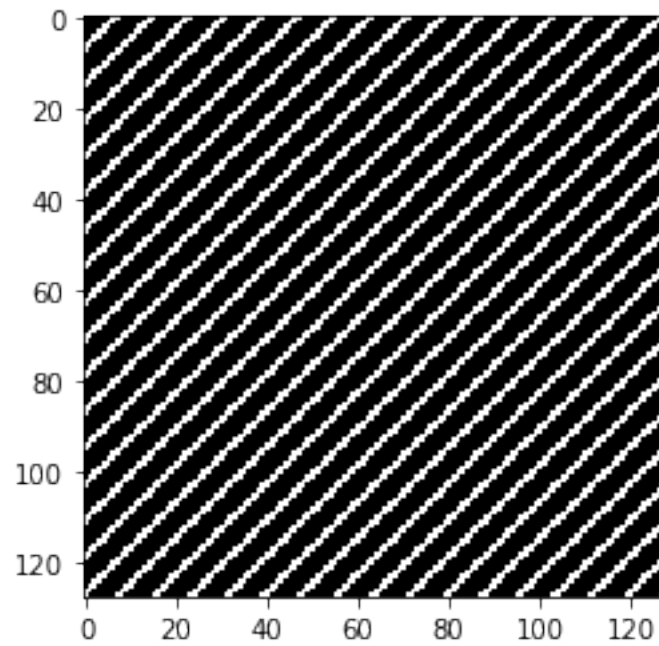
je vous montre aussi comment inverser un masque parce que c'est un peu abscons :

```
[15]: # on ne peut pas faire
      try:
          anti_rayures = not rayures
      except Exception as e:
          print(f"OOPS - {type(e)} - {e}")
```

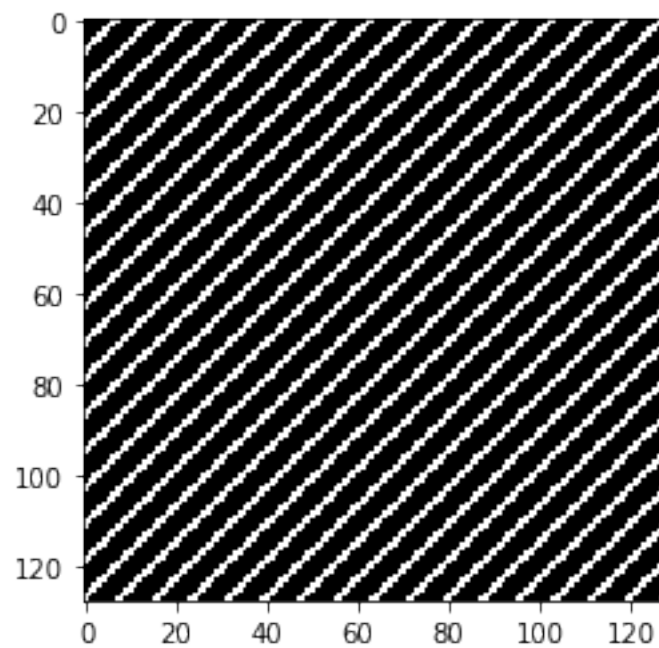
OOPS - <class 'ValueError'> - The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()

```
[16]: # on ne peut pas non plus faire
      # rayures.not()
      # parce not est un mot clé
      # et on ne peut pas non plus faire
      # rayures.logical_not()
      # et ça c'est plutôt un défaut

      anti_rayures = np.logical_not(rayures)
      plt.imshow(anti_rayures,
                  cmap='gray');
```

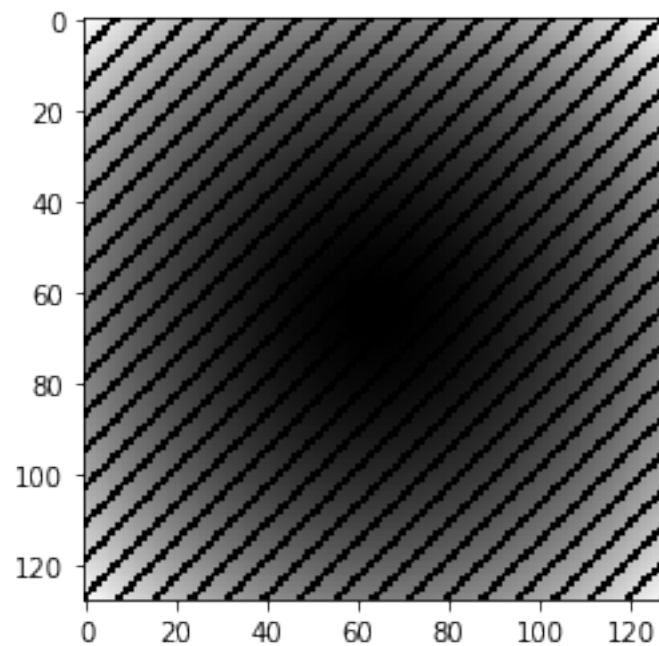


```
[17]: # lorsque vous avez de vrais  
# booléens, vous pouvez  
# utiliser l'opérateur ~  
# qui fait un not bitwise  
anti_rayures = ~rayures  
plt.imshow(anti_rayures,  
           cmap='gray');
```

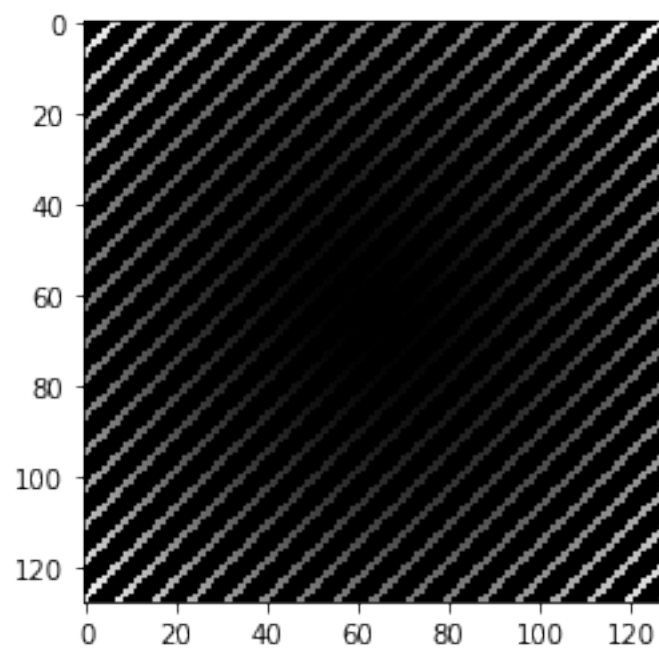


Maintenant je peux utiliser le masque `rayures` pour faire des choses sur l'image. Par exemple simplement :

```
[18]: # pour effacer les rayures  
plt.imshow(image*rayures, cmap='gray');
```



```
[19]: # ou garder l'autre moitié  
plt.imshow(image*anti_rayures, cmap='gray');
```



```
[20]: image
```

```
[20]: array([[8192., 8065., 7940., ..., 7817., 7940., 8065.],
           [8065., 7938., 7813., ..., 7690., 7813., 7938.],
           [7940., 7813., 7688., ..., 7565., 7688., 7813.],
           ...,
           [7817., 7690., 7565., ..., 7442., 7565., 7690.],
           [7940., 7813., 7688., ..., 7565., 7688., 7813.],
           [8065., 7938., 7813., ..., 7690., 7813., 7938.]])
```

```
[21]: np.logical_not(image)
```

```
[21]: array([[False, False, False, ..., False, False, False],
           [False, False, False, ..., False, False, False],
           [False, False, False, ..., False, False, False],
           ...,
           [False, False, False, ..., False, False, False],
           [False, False, False, ..., False, False, False],
           [False, False, False, ..., False, False, False]])
```

Expression conditionnelle et **np.where**

Nous avons vu en Python natif l'expression conditionnelle :

```
[22]: 3 if True else 2
```

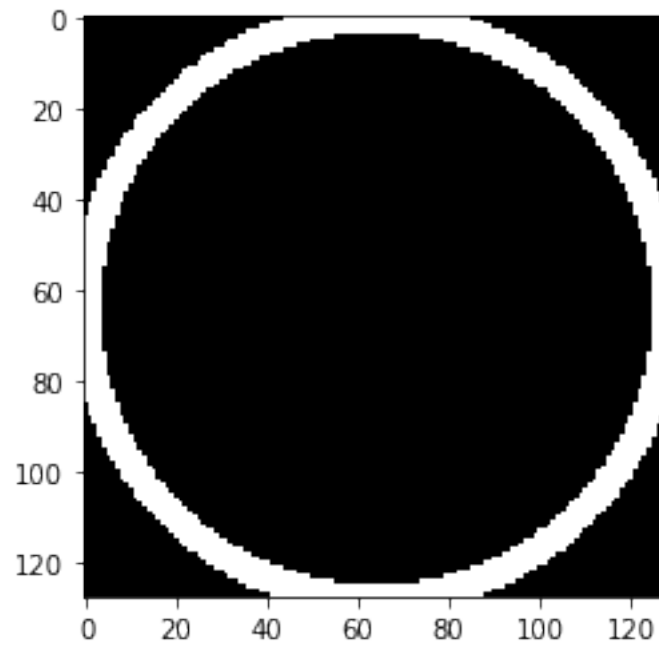
```
[22]: 3
```

Pour reproduire cette construction en **numpy** vous avez à votre disposition **np.where**. Pour l'illustrer nous allons construire deux images facilement discernables. Et, pour cela, on va utiliser **np.isclose**, qui est très utile pour comparer que deux nombres sont suffisamment proches, surtout pour les calculs flottants en fait, mais ça nous convient très bien ici aussi :

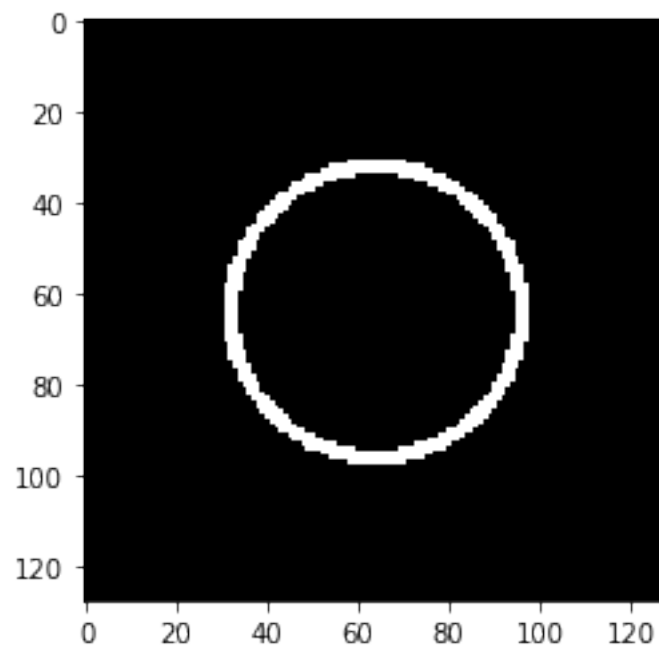
```
[23]: np.isclose?
```

Pour élaborer une image qui contient un grand cercle, je vais dire que la distance au centre (je rappelle que c'est le contenu de **image**) est suffisamment proche de 64^2 , ce que vaut **image** au milieu de chaque bord :

```
[24]: big_circle = np.isclose(image, 64 **2, 10/100)
      plt.imshow(big_circle, cmap='gray');
```



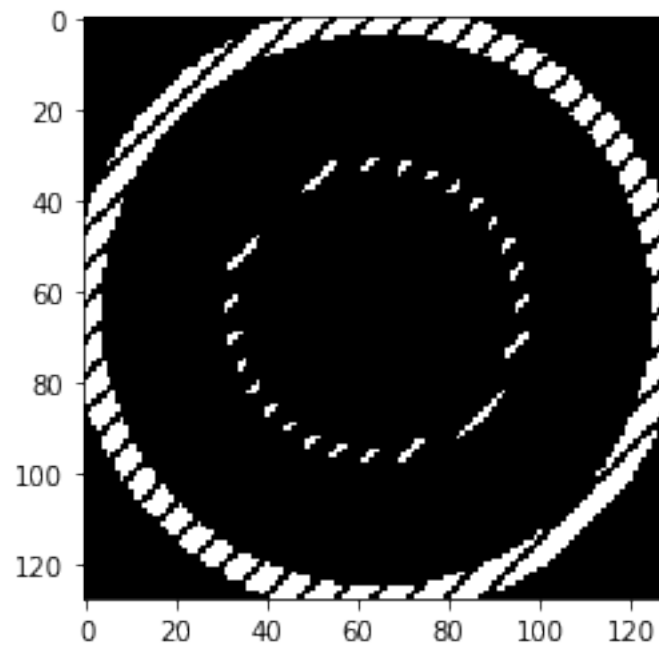
```
[25]: small_circle = np.isclose(image, 32 ** 2, 10/100)
      plt.imshow(small_circle, cmap='gray');
```



En utilisant `np.where`, je peux simuler quelque chose comme ceci :

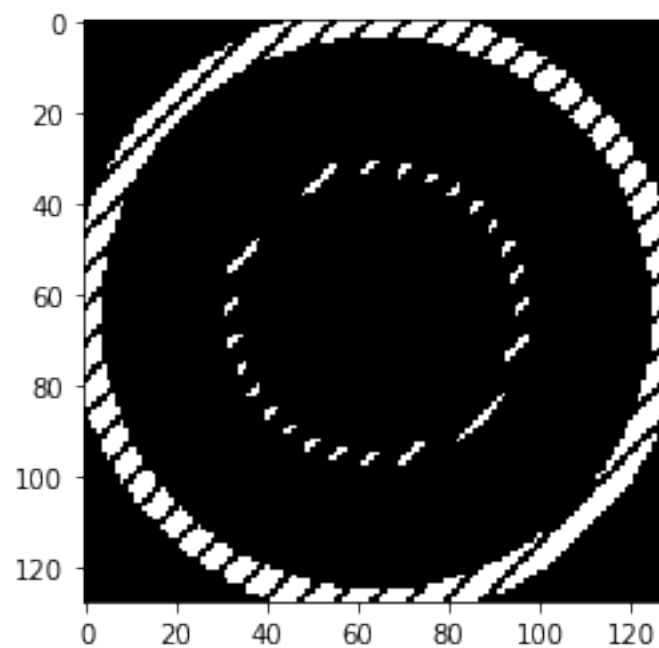
```
mixed = big_circle if rayures else small_circle
```

```
[26]: # sauf que ça se présente en fait comme ceci :  
mixed = np.where(rayures, big_circle, small_circle)  
plt.imshow(mixed, cmap='gray');
```



Remarquez enfin qu'on peut aussi faire la même chose en tirant profit que `True == 1` et `False == 0` :

```
[27]: mixed2 = rayures * big_circle + (1-rayures) * small_circle  
plt.imshow(mixed2, cmap='gray');
```



7.10

w7-s05-c4-algebre-lineaire

Algèbre linéaire

7.10.1 Complément - niveau basique

```
[1]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
```

Un aspect important de l'utilisation de **numpy** consiste à manipuler des matrices et vecteurs. Voici une rapide introduction à ces fonctionnalités.

Produit matriciel - **np.dot**

Rappel : On a déjà vu que ***** entre deux tableaux faisait une multiplication terme à terme.

```
[2]: ligne = 1 + np.arange(3)
print(ligne)
```

```
[1 2 3]
```

```
[3]: colonne = 1 + np.arange(3).reshape(3, 1)
print(colonne)
```

```
[[1]
 [2]
 [3]]
```

Ce n'est pas ce que l'on veut ici !

```
[4]: # avec le broadcasting, numpy me laisse écrire ceci
# mais **ce n'est pas** un produit matriciel
print(ligne * colonne)
```

```
[[1 2 3]
 [2 4 6]
 [3 6 9]]
```

L'opération de produit matriciel s'appelle **np.dot** :

```
[5]: m1 = np.array([[1, 1],
                   [2, 2]])
print(m1)
```

```
[[1 1]
```

```
[2 2]]
```

```
[6]: m2 = np.array([[10, 20],
                  [30, 40]])
      print(m2)
```

```
[[10 20]
 [30 40]]
```

```
[7]: # comme fonction
      np.dot(m1, m2)
```

```
[7]: array([[ 40,  60],
           [ 80, 120]])
```

```
[8]: # comme méthode
      m1.dot(m2)
```

```
[8]: array([[ 40,  60],
           [ 80, 120]])
```

Je vous signale aussi un opérateur spécifique, noté @, qui permet également de faire le produit matriciel.

```
[9]: m1 @ m2
```

```
[9]: array([[ 40,  60],
           [ 80, 120]])
```

```
[10]: m2 @ m1
```

```
[10]: array([[ 50,  50],
           [110, 110]])
```

C'est un opérateur un peu ad hoc pour `numpy`, puisqu'il ne fait pas de sens avec les types usuels de Python :

```
[11]: for x, y in ( (10, 20), (10., 20.), ([10], [20]), ((10,), (20,))):
      try:
          x @ y
      except Exception as e:
          print(f"OOPS - {type(e)} - {e}")
```

```
OOPS - <class 'TypeError'> - unsupported operand type(s) for @: 'int' and '
int'
```

```
OOPS - <class 'TypeError'> - unsupported operand type(s) for @: 'float' and
'float'
```

```
OOPS - <class 'TypeError'> - unsupported operand type(s) for @: 'list' and
'list'
```

```
OOPS - <class 'TypeError'> - unsupported operand type(s) for @: 'tuple' and
'tuple'
```

Produit scalaire - `np.dot` ou `@`

Ici encore, vous pouvez utiliser `dot` qui va intelligemment transposer le second argument :

```
[12]: v1 = np.array([1, 2, 3])  
      print(v1)
```

```
[1 2 3]
```

```
[13]: v2 = np.array([4, 5, 6])  
      print(v2)
```

```
[4 5 6]
```

```
[14]: np.dot(v1, v2)
```

```
[14]: 32
```

```
[15]: v1 @ v2
```

```
[15]: 32
```

Transposée

Vous pouvez accéder à une matrice transposée de deux façons :

— soit sous la forme d'un attribut `m.T` :

```
[16]: m = np.arange(4).reshape(2, 2)  
      print(m)
```

```
[[0 1]  
 [2 3]]
```

```
[17]: print(m.T)
```

```
[[0 2]  
 [1 3]]
```

— soit par la méthode `transpose()` :

```
[18]: print(m)
```

```
[[0 1]  
 [2 3]]
```

```
[19]: m.transpose()
```

```
[19]: array([[0, 2],  
            [1, 3]])
```

Matrice identité - `np.eye`

```
[20]: np.eye(4, dtype=np.int)
```

```
[20]: array([[1, 0, 0, 0],
           [0, 1, 0, 0],
           [0, 0, 1, 0],
           [0, 0, 0, 1]])
```

Matrices diagonales - `np.diag`

Avec `np.diag`, vous pouvez dans les deux sens :

- extraire la diagonale d'une matrice;
- construire une matrice à partir de sa diagonale.

```
[21]: M = np.arange(4) + 10 * np.arange(4)[:, np.newaxis]
      print(M)
```

```
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]]
```

```
[22]: D = np.diag(M)
      print(D)
```

```
[ 0 11 22 33]
```

```
[23]: M2 = np.diag(D)
      print(M2)
```

```
[[ 0  0  0  0]
 [ 0 11  0  0]
 [ 0  0 22  0]
 [ 0  0  0 33]]
```

Déterminant - `np.linalg.det`

Avec la fonction `np.linalg.det` :

```
[24]: # une isométrie
      M = np.array([[0, -1], [1, 0]])
      print(M)
```

```
[[ 0 -1]
 [ 1  0]]
```

```
[25]: # et donc
      np.linalg.det(M) == 1
```


[25]: True

Valeurs propres - `np.linalg.eig`

Vous pouvez obtenir valeurs propres et vecteurs propres d'une matrice avec `np.eig` :

```
[26]: # la symétrie par rapport à x=y  
S = np.array([[0, 1], [1, 0]])
```

```
[27]: values, vectors = np.linalg.eig(S)
```

```
[28]: # pas de déformation  
values
```

[28]: array([1., -1.])

```
[29]: # les deux diagonales  
vectors
```

```
[29]: array([[ 0.70710678, -0.70710678],  
           [ 0.70710678,  0.70710678]])
```

Systèmes d'équations - `np.linalg.solve`

Fabriquons-nous un système d'équations :

```
[30]: x, y, z = 1, 2, 3
```

```
[31]: 3*x + 2*y + z
```

[31]: 10

```
[32]: 2*x + 3*y + 4*z
```

[32]: 20

```
[33]: 5*x + 2*y + 6*z
```

[33]: 27

On peut le résoudre tout simplement comme ceci :

```
[34]: coefficients= np.array([  
    [3, 2, 1],  
    [2, 3, 4],  
    [5, 2, 6],  
])
```

```
[35]: constants = [
    10,
    20,
    27,
]
```

```
[36]: X, Y, Z = np.linalg.solve(coefficients, constants)
```

Par contre bien sûr on est passé par les flottants, et donc on a le souci habituel avec la précision des arrondis :

```
[37]: Z
```

```
[37]: 3.0000000000000004
```

Résumé

En résumé, ce qu'on vient de voir :

outil	propos
<code>np.dot</code>	produit matriciel
<code>np.dot</code>	produit scalaire
<code>np.transpose</code>	transposée
<code>np.eye</code>	matrice identité
<code>np.diag</code>	extrait la diagonale
<code>np.diag</code>	ou construit une matrice diagonale
<code>np.linalg.det</code>	déterminant
<code>np.linalg.eig</code>	valeurs propres
<code>np.linalg.solve</code>	résout système équations

Pour en savoir plus

Voyez la [documentation complète](#) sur l'algèbre linéaire.

7.11 w7-s05-c5-indexation-evoluee

Indexation évoluée

7.11.1 Complément - niveau avancé

Nous allons maintenant voir qu'il est possible d'indexer un tableau **numpy** avec, non pas des entiers ou des tuples comme on l'a vu dans un complément précédent, mais aussi avec d'autres types d'objets qui permettent des manipulations très puissantes :

- indexation par une liste ;
- indexation par un tableau ;
- indexation multiple (par un tuple) ;
- indexation par un tableau de booléens.

```
[1]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
```

Pour illustrer ceci, on va réutiliser la fonction `background` que l'on avait vue pour les indexations simples :

```
[2]: # une fonction qui crée un tableau
# tab[i, j] = i + 10 * j
def background(n):
    i = np.arange(n)
    j = i.reshape((n, 1))
    return i + 10 * j
```

Indexation par une liste

On peut indexer par une liste d'entiers, cela constitue une généralisation des slices.

```
[3]: b = background(6)
print(b)
```

```
[[ 0  1  2  3  4  5]
 [10 11 12 13 14 15]
 [20 21 22 23 24 25]
 [30 31 32 33 34 35]
 [40 41 42 43 44 45]
 [50 51 52 53 54 55]]
```

Si je veux référencer les lignes 1, 3 et 4, je ne peux pas utiliser un slice ; mais je peux utiliser une liste à la place :

```
[4]: # il faut lire ceci comme
# j'indexe b, avec comme indice la liste [1, 3, 4]
b[[1, 3, 4]]
```

```
[4]: array([[10, 11, 12, 13, 14, 15],
          [30, 31, 32, 33, 34, 35],
          [40, 41, 42, 43, 44, 45]])
```

```
[5]: # pareil pour les colonnes, en combinant avec un slice
b[:, [1, 3, 4]]
```

```
[5]: array([[ 1,  3,  4],
          [11, 13, 14],
          [21, 23, 24],
          [31, 33, 34],
          [41, 43, 44],
          [51, 53, 54]])
```

```
[6]: # et comme toujours on peut faire du broadcasting
b[:, [1, 3, 4]] = np.arange(1000, 1006).reshape((6, 1))
print(b)
```

```
[[ 0 1000  2 1000 1000  5]
 [ 10 1001 12 1001 1001 15]
 [ 20 1002 22 1002 1002 25]
 [ 30 1003 32 1003 1003 35]
 [ 40 1004 42 1004 1004 45]
 [ 50 1005 52 1005 1005 55]]
```

Indexation par un tableau

On peut aussi indexer un tableau A ... par un tableau ! Pour que cela ait un sens :

- le tableau d'index doit contenir des entiers ;
- ces derniers doivent être tous plus petits que la première dimension de A.

Le cas simple : l'entrée et l'index sont de dimension 1.

```
[7]: # le tableau qu'on va indexer
cubes = np.arange(10) ** 3
print(cubes)
```

```
[ 0  1  8 27 64 125 216 343 512 729]
```

```
[8]: # et un index qui est un tableau numpy
# doit contenir des entiers entre 0 et 9
tab = np.array([1, 7, 2])
print(cubes[tab])
```

```
[ 1 343  8]
```

```
[9]: # donne - logiquement - le même résultat que
# si l'index était une liste Python
lis = [1, 7, 2]
print(cubes[lis])
```

```
[ 1 343  8]
```

De manière générale Dans le cas général, le résultat de A[index] :

- a la même forme “externe” que index ;
- où l'on a remplacé i par A[i] ;
- qui peut donc être un tableau si A est de dimension > 1

```
[10]: A = np.array([[0, 'zero'], [1, 'un'], [2, 'deux'], [3, 'trois']])
print(A)
```

```
[[0 'zero']
 [1 'un']
 [2 'deux']
 [3 'trois']]
```

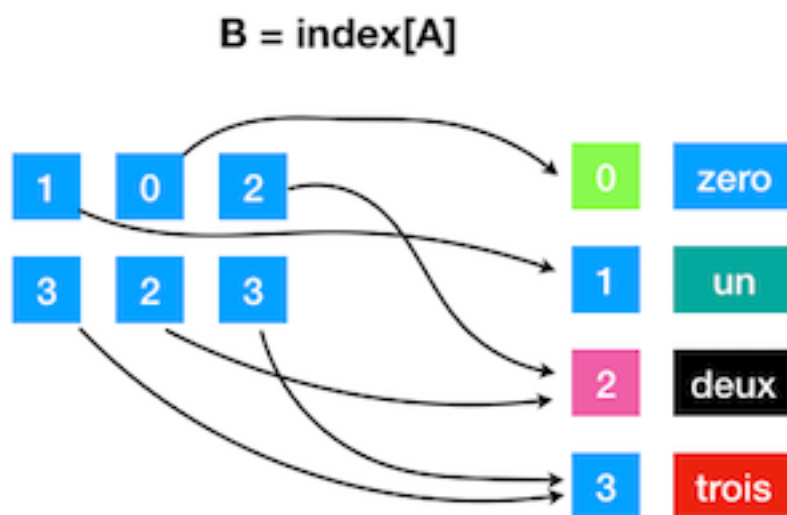
```
[11]: index = np.array([[1, 0, 2], [3, 2, 3]])
print(index)
```

```
[[1 0 2]  
 [3 2 3]]
```



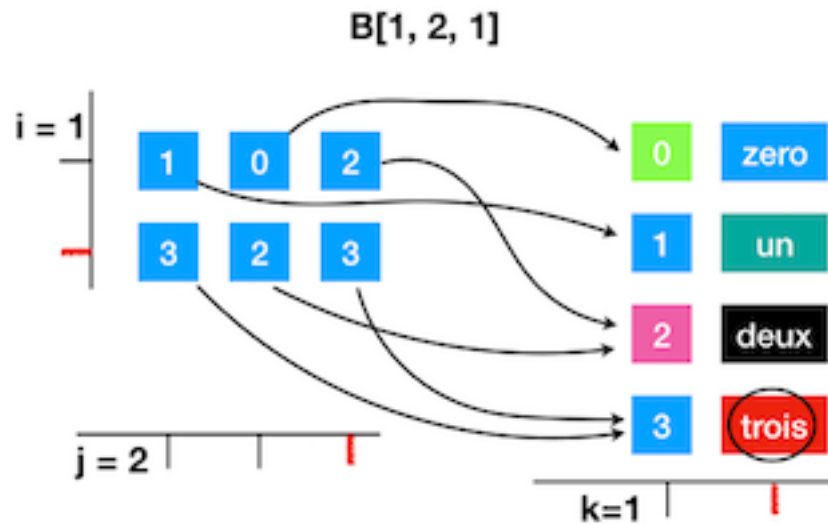
```
[12]: B = A[index]  
      print(B)
```

```
[[['1' 'un']  
  ['0' 'zero']  
  ['2' 'deux']]  
  
[['3' 'trois']  
  ['2' 'deux']  
  ['3' 'trois']]]
```



```
[13]: B[1, 2, 1]
```

```
[13]: 'trois'
```



Et donc si :

- `index` est de dimension (i, j, k) ;
- `A` est de dimension (a, b) .

Alors :

- `A[index]` est de dimension (i, j, k, b) ;
- il faut que les éléments dans `index` soient dans $[0 \dots a[$.

Ce que l'on vérifie ici :

```
[14]: # l'entrée
      print(A.shape)
```

```
(4, 2)
```

```
[15]: # l'index
      print(index.shape)
```

```
(2, 3)
```

```
[16]: # le résultat
      print(A[index].shape)
```

```
(2, 3, 2)
```

Cas particulier : entrée de dimension 1, `index` de dim. > 1 Lorsque l'entrée `A` est de dimension 1, alors la sortie a exactement la même forme que l'`index`.

C'est comme si `A` était une fonction que l'on applique aux indices dans `index`.

```
[17]: print(cubes)
```

```
[ 0  1  8 27 64 125 216 343 512 729]
```

```
[18]: i2 = np.array([[2, 4], [8, 9]])
      print(i2)
```

```
[[2 4]
 [8 9]]
```

```
[19]: print(cubes[i2])
```

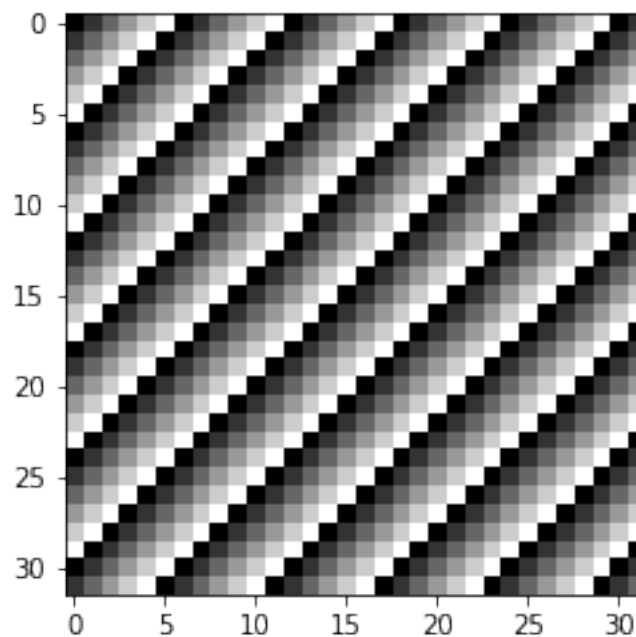
```
[[ 8 64]
 [512 729]]
```

Application au codage des couleurs dans une image

```
[20]: # je crée une image avec 6 valeurs disposées en diagonale
      N = 32
      colors = 6

      image = np.empty((N, N), dtype = np.int32)
      for i in range(N):
          for j in range(N):
              image[i, j] = (i+j) % colors
```

```
[21]: plt.imshow(image, cmap='gray');
```

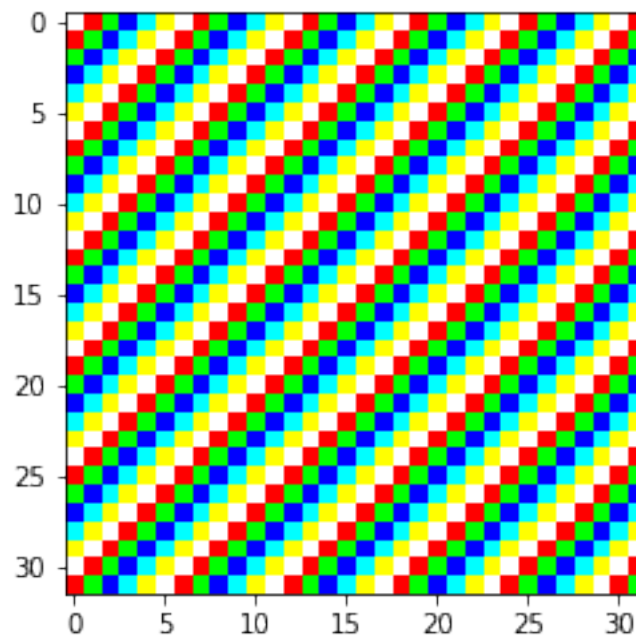


Les couleurs ne sont pas significatives, ce sont des valeurs entières dans `range(colors)`. On voudrait pouvoir choisir la vraie couleur correspondant à chaque valeur. Pour cela on peut utiliser une simple indexation par tableau :

```
[22]: # une palette de couleurs
      palette = np.array([
          [255, 255, 255], # 0 -> blanc
```

```
[255, 0, 0],      # 1 -> rouge
[0, 255, 0],      # 2 -> vert
[0, 0, 255],      # 3 -> bleu
[0, 255, 255],    # 4 -> cyan
[255, 255, 0],    # 5 -> magenta
], dtype=np.uint8)
```

```
[23]: plt.imshow(palette[image]);
```



Remarquez que la forme générale n'a pas changé, mais le résultat de l'indexation a une dimension supplémentaire de 3 couleurs :

```
[24]: image.shape
```

```
[24]: (32, 32)
```

```
[25]: palette[image].shape
```

```
[25]: (32, 32, 3)
```

Indexation multiple (par tuple)

Une fois que vous avez compris ce mécanisme d'indexation par un tableau, on peut encore généraliser pour définir une indexation par deux (ou plus) tableaux de formes identiques.

Ainsi, lorsque `index1` et `index2` ont la même forme :

- on peut écrire `A[index1, index2]`
- qui a la même forme externe que les `index`

- où on a remplacé i, j par $A[i][j]$
- qui peut donc être un tableau si A est de dimension > 2 .

```
[26]: # un tableau à indexer
ix, iy = np.indices((4, 3))
A = 10 * ix + iy
print(A)
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

```
[27]: # les deux tableaux d'indices sont carrés 2x2
index1 = [[3, 1], [0, 1]] # doivent être < 4
index2 = [[2, 0], [0, 2]] # doivent être < 3
# le résultat est donc carré 2x2
print(A[index1, index2])
```

```
[[32 10]
 [ 0 12]]
```

Et donc si :

- `index1` et `index2` sont de dimension (i, j, k)
- et A est de dimension (a, b, c)

Alors :

- le résultat est de dimension (i, j, k, c)
- il faut alors que les éléments de `index1` soient dans $[0 \dots a[$
- et les éléments de `index2` dans $[0 \dots b[$

Application à la recherche de maxima Imaginons que vous avez des mesures pour plusieurs instants :

```
[28]: times = np.linspace(1000, 5000, num=5, dtype=int)
print(times)
```

```
[1000 2000 3000 4000 5000]
```

```
[29]: # on aurait 3 mesures à chaque instant
series = np.array([
    [10, 25, 32, 23, 12],
    [12, 8, 4, 10, 7],
    [100, 80, 90, 110, 120]])
print(series)
```

```
[[ 10  25  32  23  12]
 [ 12   8   4  10   7]
 [100  80  90 110 120]]
```

Avec la fonction `np.argmax` on peut retrouver les indices des points maxima dans `series` :

```
[30]: max_indices = np.argmax(series, axis=1)
      print(max_indices)
```

```
[2 0 4]
```

Pour trouver les maxima en question, on peut faire :

```
[31]: # les trois maxima, un par serie
      maxima = series[ range(series.shape[0]), max_indices ]
      print(maxima)
```

```
[ 32  12 120]
```

```
[32]: # et ils correspondent à ces instants-ci
      times[max_indices]
```

```
[32]: array([3000, 1000, 5000])
```

Indexation par un tableau de booléens

Une forme un peu spéciale d'indexation consiste à utiliser un tableau de booléens, qui agit comme un masque :

```
[33]: suite = np.array([1, 2, 3, 4, 5, 4, 3, 2, 1])
```

Je veux filtrer ce tableau et ne garder que les valeurs < 4 :

```
[34]: # je construis un masque
      hauts = suite >= 4
      print(hauts)
```

```
[False False False  True  True  True False False False]
```

```
[35]: # je peux utiliser ce masque pour calculer les indices qui sont vrais
      suite[hauts]
```

```
[35]: array([4, 5, 4])
```

```
[36]: # et utiliser maintenant ceci par un index de tableau
      # par exemple pour annuler ces valeurs
      suite[hauts] = 0
      print(suite)
```

```
[1 2 3 0 0 0 3 2 1]
```

7.12 w7-s05-c6-divers

Divers

7.12.1 Complément - niveau avancé

```
[1]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
```

Pour finir notre introduction à `numpy`, nous allons survoler à très grande vitesse quelques traits plus annexes mais qui peuvent être utiles. Je vous laisse approfondir de votre côté les parties qui vous intéressent.

7.13 w7-s05-c6-divers

Utilisation de la mémoire

Références croisées, vues, shallow et deep copies

Pour résumer ce qu'on a vu jusqu'ici :

- un tableau `numpy` est un objet mutable ;
- une slice sur un tableau retourne une vue, on est donc dans le cas d'une référence partagée ;
- dans tous les cas que l'on a vus jusqu'ici, comme les cases des tableaux sont des objets atomiques, il n'y a pas de différence entre shallow et deep copie ;
- pour créer une copie, utilisez `np.copy()`.

Et de plus :

```
[2]: # un tableau de base
a = np.arange(3)
```

```
[3]: # une vue
v = a.view()
```

```
[4]: # une slice
s = a[:]
```

Les deux objets ne sont pas différentiables :

```
[5]: v.base is a
```

```
[5]: True
```

```
[6]: s.base is a
```

```
[6]: True
```

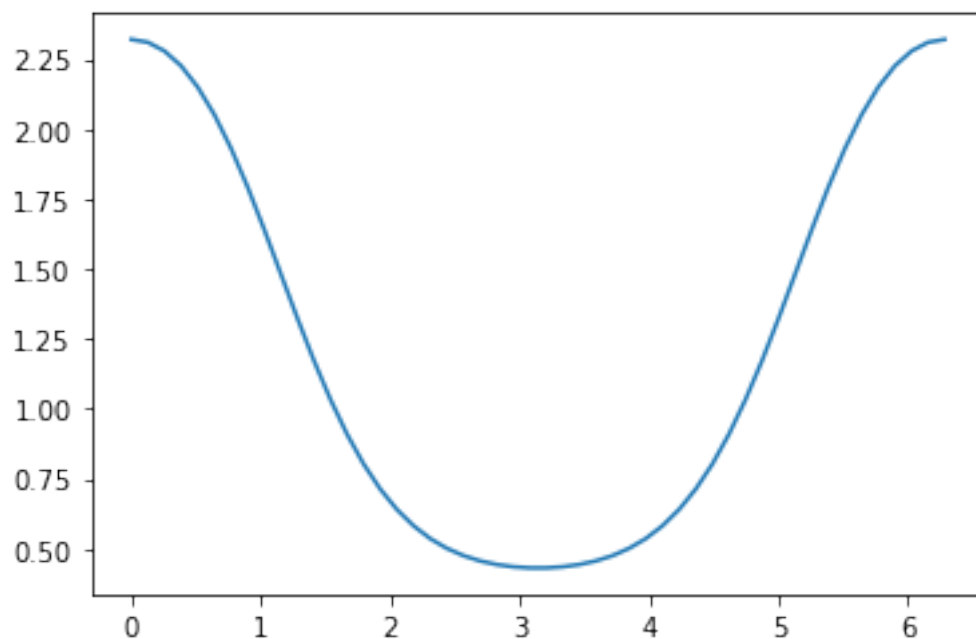
L'option `out=`

Lorsque l'on fait du calcul vectoriel, on peut avoir tendance à créer de nombreux tableaux intermédiaires qui coûtent cher en mémoire. Pour cette raison, presque tous les opérateurs **numpy** proposent un paramètre optionnel `out=` qui permet de spécifier un tableau déjà alloué, dans lequel ranger le résultat.

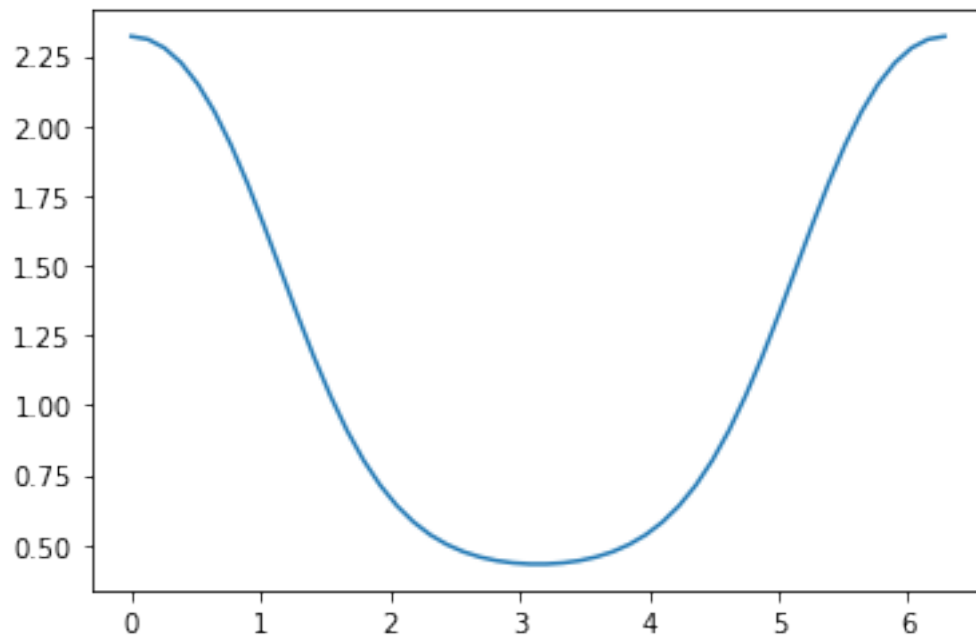
Prenons l'exemple un peu factice suivant, où on calcule $e^{\sin(\cos(x))}$ sur l'intervalle $[0, 2\pi]$:

```
[7]: # le domaine  
X = np.linspace(0, 2*np.pi)
```

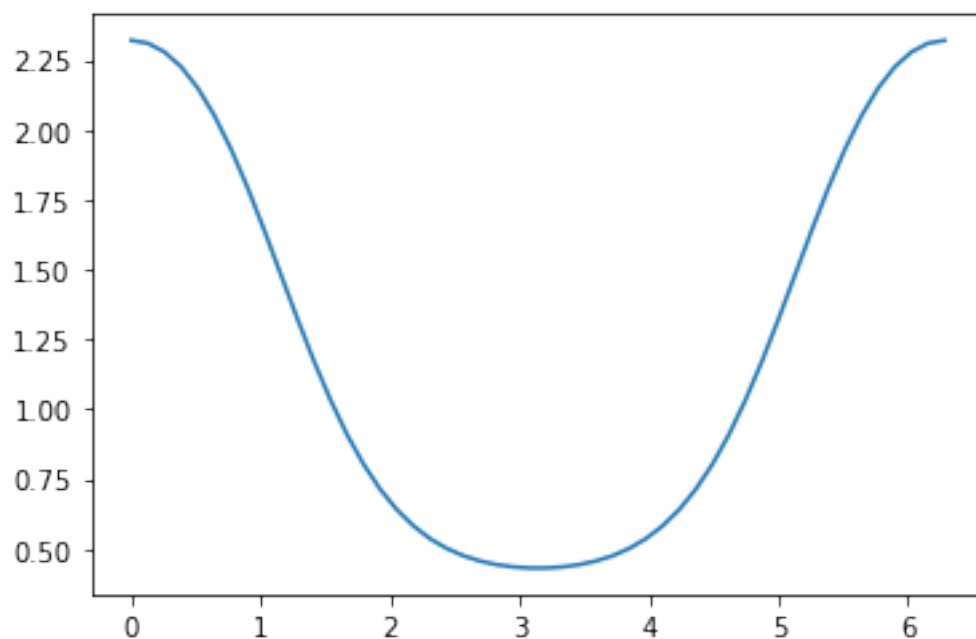
```
[8]: Y = np.exp(np.sin(np.cos(X)))  
plt.plot(X, Y);
```



```
[9]: # chaque fonction alloue un tableau pour ranger ses résultats,  
# et si je décompose, ce qui se passe en fait c'est ceci  
Y1 = np.cos(X)  
Y2 = np.sin(Y1)  
Y3 = np.exp(Y2)  
# en tout en comptant X et Y j'aurai créé 4 tableaux  
plt.plot(X, Y3);
```



```
[10]: # Mais moi je sais qu'en fait je n'ai besoin que de X et de Y  
# ce qui fait que je peux optimiser comme ceci :  
  
# je ne peux pas réécrire sur X parce que j'en aurai besoin pour le plot  
X1 = np.cos(X)  
# par contre ici je peux recycler X1 sans souci  
np.sin(X1, out=X1)  
# etc ...  
np.exp(X1, out=X1)  
plt.plot(X, X1);
```



Et avec cette approche je n'ai créé que 2 tableaux en tout.

Notez bien : je ne vous recommande pas d'utiliser ceci systématiquement, car ça défigure nettement le code. Mais il faut savoir que ça existe, et savoir y penser lorsque la création de tableaux intermédiaires a un coût important dans l'algorithme.

`np.add` et similaires

Si vous vous mettez à optimiser de cette façon, vous utiliserez par exemple `np.add` plutôt que `+`, qui ne vous permet pas de choisir la destination du résultat.

7.14 w7-s05-c6-divers

Types structurés pour les cellules

Sans transition, jusqu'ici on a vu des tableaux atomiques, où chaque cellule est en gros un seul nombre.

En fait, on peut aussi se définir des types structurés, c'est-à-dire que chaque cellule contient l'équivalent d'un struct en C.

Pour cela, on peut se définir un `dtype` élaboré, qui va nous permettre de définir la structure de chacun de ces enregistrements.

Exemple

```
[11]: # un dtype structuré
my_dtype = [
    # prenom est un string de taille 12
    ('prenom', '|S12'),
    # nom est un string de taille 15
    ('nom', '|S15'),
    # age est un entier
    ('age', np.int)
]

# un tableau qui contient des cellules de ce type
classe = np.array(
    # le contenu
    [ ('Jean', 'Dupont', 32),
      ('Daniel', 'Durand', 18),
      ('Joseph', 'Delapierre', 54),
      ('Paul', 'Girard', 20)],
    # le type
    dtype = my_dtype)
classe
```

```
[11]: array([(b'Jean', b'Dupont', 32), (b'Daniel', b'Durand', 18),
             (b'Joseph', b'Delapierre', 54), (b'Paul', b'Girard', 20)],
            dtype=[('prenom', 'S12'), ('nom', 'S15'), ('age', '<i8')])
```

Je peux avoir l'impression d'avoir créé un tableau de 4 lignes et 3 colonnes ; cependant pour `numpy` ce n'est pas comme ça que cela se présente :

```
[12]: classe.shape
```

```
[12]: (4,)
```

Rien ne m'empêcherait de créer des tableaux de ce genre en dimensions supérieures, bien entendu :

```
[13]: # ça n'a pas beaucoup d'intérêt ici, mais si on a besoin
      # on peut bien sûr avoir plusieurs dimensions
      classe.reshape((2, 2))
```

```
[13]: array([(b'Jean', b'Dupont', 32), (b'Daniel', b'Durand', 18)],
            [(b'Joseph', b'Delapierre', 54), (b'Paul', b'Girard', 20)]],
        dtype=[('prenom', 'S12'), ('nom', 'S15'), ('age', '<i8')])
```

Comment définir `dtype` ?

Il existe une grande variété de moyens pour se définir son propre `dtype`.

Je vous signale notamment la possibilité de spécifier à l'intérieur d'un `dtype` des cellules de type `object`, qui est l'équivalent d'une référence Python (approximativement, un pointeur dans un struct C) ; c'est un trait qui est utilisé par `pandas` que nous allons voir très bientôt.

Pour la définition de types structurés, [voir la documentation complète ici](#).

7.15

w7-s05-c6-divers

Assemblages et découpages

Enfin, toujours sans transition, et plus anecdotique : jusqu'ici nous avons vu des fonctions qui préservent la taille. Le stacking permet de créer un tableau plus grand en (juxta/super)posant plusieurs tableaux. Voici rapidement quelques fonctions qui permettent de faire des tableaux plus petits ou plus grands.

Assemblages : `hstack` et `vstack` (tableaux 2D)

```
[14]: a = np.arange(1, 7).reshape(2, 3)
      print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

```
[15]: b = 10 * np.arange(1, 7).reshape(2, 3)
      print(b)
```

```
[[10 20 30]
 [40 50 60]]
```

```
[16]: print(np.hstack((a, b)))
```

```
[[ 1  2  3 10 20 30]
 [ 4  5  6 40 50 60]]
```

```
[17]: print(np.vstack((a, b)))
```

```
[[ 1  2  3]
 [ 4  5  6]
 [10 20 30]
 [40 50 60]]
```

Assemblages : `np.concatenate` (3D et au delà)

```
[18]: a = np.ones((2, 3, 4))
      print(a)
```

```
[[[1.  1.  1.  1.]
   [1.  1.  1.  1.]
   [1.  1.  1.  1.]]
```

```
[[1.  1.  1.  1.]
 [1.  1.  1.  1.]
 [1.  1.  1.  1.]]]
```

```
[19]: b = np.zeros((2, 3, 2))
      print(b)
```

```
[[[0.  0.]
   [0.  0.]
   [0.  0.]]
```

```
[[0.  0.]
 [0.  0.]
 [0.  0.]]]
```

```
[20]: print(np.concatenate((a, b), axis = 2))
```

```
[[[1.  1.  1.  1.  0.  0.]
   [1.  1.  1.  1.  0.  0.]
   [1.  1.  1.  1.  0.  0.]]
```

```
[[1.  1.  1.  1.  0.  0.]
 [1.  1.  1.  1.  0.  0.]
 [1.  1.  1.  1.  0.  0.]]]
```

Pour conclure :

- `hstack` et `vstack` utiles sur des tableaux 2D ;
- au-delà, préférez `concatenate` qui a une sémantique plus claire.

Répétitions : `np.tile`

Cette fonction permet de répéter un tableau dans toutes les directions :


```
[21]: motif = np.array([[0, 1], [2, 10]])  
      print(motif)
```

```
[[ 0  1]  
 [ 2 10]]
```

```
[22]: print(np.tile(motif, (2, 3)))
```

```
[[ 0  1  0  1  0  1]  
 [ 2 10  2 10  2 10]  
 [ 0  1  0  1  0  1]  
 [ 2 10  2 10  2 10]]
```

Découpage : `np.split`

Cette opération, inverse du stacking, consiste à découper un tableau en parties plus ou moins égales :

```
[23]: complet = np.arange(24).reshape(4, 6); print(complet)
```

```
[[ 0  1  2  3  4  5]  
 [ 6  7  8  9 10 11]  
 [12 13 14 15 16 17]  
 [18 19 20 21 22 23]]
```

```
[24]: h1, h2 = np.hsplit(complet, 2)  
      print(h1)
```

```
[[ 0  1  2]  
 [ 6  7  8]  
 [12 13 14]  
 [18 19 20]]
```

```
[25]: print(h2)
```

```
[[ 3  4  5]  
 [ 9 10 11]  
 [15 16 17]  
 [21 22 23]]
```

```
[26]: complet = np.arange(24).reshape(4, 6)  
      print(complet)
```

```
[[ 0  1  2  3  4  5]  
 [ 6  7  8  9 10 11]  
 [12 13 14 15 16 17]  
 [18 19 20 21 22 23]]
```

```
[27]: v1, v2 = np.vsplit(complet, 2)  
      print(v1)
```

```
[[ 0  1  2  3  4  5]  
 [ 6  7  8  9 10 11]]
```

```
[28]: print(v2)
```

```
[[12 13 14 15 16 17]
 [18 19 20 21 22 23]]
```

7.16

w7-s05-x1-checkers

Exercice - niveau basique

```
[1]: import numpy as np
```

```
[2]: from corrections.exo_checkers import exo_checkers
```

On vous demande d'écrire une fonction `checkers` qui crée un tableau `numpy`.

La fonction prend en argument :

- un entier `size` ≥ 1
- et un booléen `corner_0_0` - qui vaut par défaut `True`

Elle construit et retourne alors un tableau carré de taille `size` x `size`, qui est rempli comme un damier avec des entiers 0 et 1 ; la valeur de la cellule d'indice 0 x 0 est correspond au paramètre `corner_0_0`.

On rappelle par ailleurs que `False == 0` et `True == 1`.

```
[3]: # voici deux exemples pour la fonction checkers
      exo_checkers.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[4]: # à vous de jouer
      def checkers(size, corner_0_0=True):
          return "votre code"

      # NOTE:
      # auto-exec-for-latex has used hidden code instead
```

```
[ ]: # pour corriger votre code
      exo_checkers.correction(checkers)

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

Visualisation

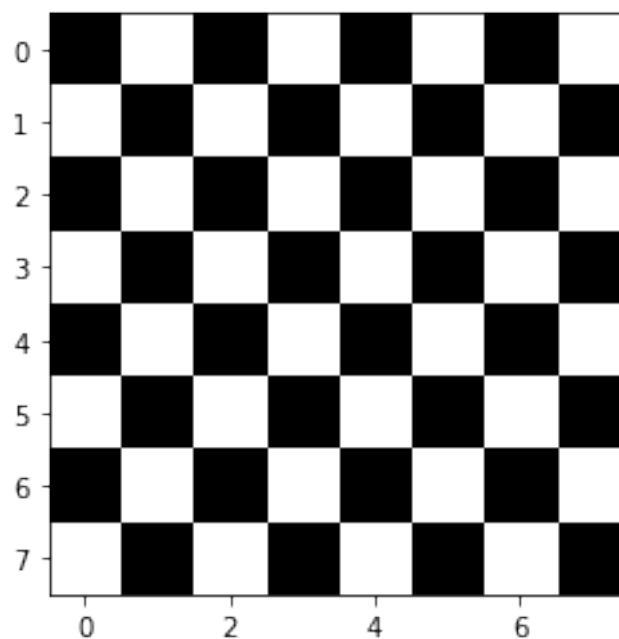
```
[5]: import matplotlib.pyplot as plt
      %matplotlib inline
      plt.ion()
```

L'exercice est terminé, mais si vous avez réussi et que vous voulez visualiser le résultat, voici comment vous pouvez aussi voir ce type de tableau :

```
[6]: checkerboard = checkers(8, False)
```

Pour le voir comme une image :

```
[7]: # convertir en flottant pour imshow
checkerboard = checkerboard.astype(np.float)
# afficher avec une colormap 'gray' pour avoir du noir et blanc
plt.imshow(checkerboard, cmap='gray');
```



7.17 w7-s05-x2-hundreds

Exercice - niveau basique

```
[1]: import numpy as np
```

```
[2]: from corrections.exo_hundreds import exo_hundreds
```

On vous demande d'écrire une fonction `hundreds` qui crée un tableau `numpy`.

La fonction prend en argument :

- deux entiers `lines`, `columns`
- un nombre entier `offset`

Le résultat doit être un tableau de taille `lines` x `columns`, composé d'entiers, et on veut qu'en une case de coordonnées `(i, j)` la valeur du tableau soit égale à

$$result[i,j] = 100 * i + 10 * j + offset$$

```
[3]: # voici deux exemples pour la fonction hundreds
      exo_hundreds.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[4]: # à vous de jouer
      def hundreds(lines, columns, offset):
          return "votre code"

      # NOTE:
      # auto-exec-for-latex has used hidden code instead
```

```
[ ]: # pour corriger votre code
      exo_hundreds.correction(hundreds)

      # NOTE
      # auto-exec-for-latex has skipped execution of this cell
```

7.17.1 Plusieurs angles possibles

- La première idée peut-être, consiste à faire comme en Fortran, avec deux boucles imbriquées ; c'est facile à écrire, ça fonctionne, mais ce n'est pas très élégant ;
- vous pouvez aussi penser à utiliser du broadcasting :
 - dans ce cas-là `np.indices()` peut vous être utile ;
 - vous pouvez aussi vous entraîner à fabriquer la souche des lignes et des colonnes à la main avec `np.arange()` en combinaison avec `np.newaxis` ;
- et sans doute d'autres auxquelles je n'ai pas pensé !)

7.18 w7-s05-x3-stairs

Exercice - niveau intermédiaire

```
[1]: import numpy as np
```

```
[2]: from corrections.exo_stairs import exo_stairs
```

On vous demande d'écrire une fonction `stairs` qui crée un tableau `numpy`.

La fonction prend en argument un entier `taille` et construit un tableau carré de taille $2 * taille + 1$.

Aux quatre coins du tableau on trouve la valeur 0. Dans la case centrale on trouve la valeur $2 * taille$.

Si vous partez de n'importe quelle case et que vous vous déplacez d'une case horizontalement ou verticalement vers une cas plus proche du centre, vous incrémentez la valeur du tableau de 1.

```
[3]: # voici deux exemples pour la fonction stairs
      exo_stairs.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[4]: # à vous de jouer
def stairs(taille):
    return "votre code"

# NOTE:
# auto-exec-for-latex has used hidden code instead
```

```
[ ]: # pour corriger votre code
exo_stairs.correction(stairs)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Visualisation

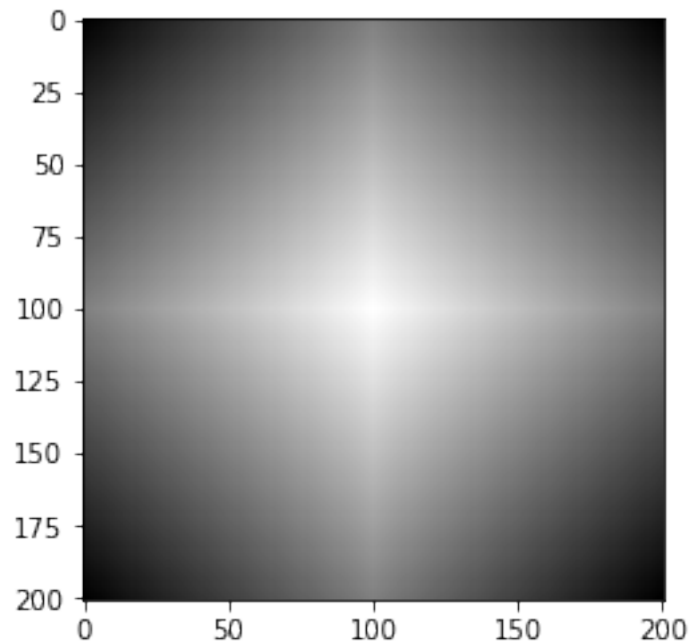
```
[5]: import matplotlib.pyplot as plt
      %matplotlib inline
      plt.ion()
```

L'exercice est terminé, voyons à nouveau notre résultat sous forme d'image :

```
[6]: squares = stairs(100)
```

Pour le voir comme une image avec un niveau de gris comme code de couleurs (noir = 0, blanc = maximum = 201 dans notre cas) :

```
[7]: # convertir en flottant pour imshow
squares = squares.astype(np.float)
# afficher avec une colormap 'gray'
plt.imshow(squares, cmap='gray');
```



7.19 w7-s05-x4-dice

Exercice - niveau avancé

```
[1]: import numpy as np
```

```
[2]: from corrections.exo_dice import exo_dice
```

On étudie les probabilités d'obtenir une certaine somme avec plusieurs dés.

Tout le monde connaît le cas classique avec deux dés à 6 faces, ou l'on construit mentalement la grille suivante :

+	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Imaginons que vous êtes étudiant, vous venez de faire un exercice de maths qui vous a mené à une formule qui permet de calculer, pour un jeu à `nb_dice` dés, chacun à `sides` faces, le nombre de tirages qui donnent une certaine somme `target`.

Vous voulez vérifier votre formule, en appliquant une méthode de force brute.

C'est l'objet de cet exercice. Vous devez écrire une fonction `dice` qui prend en paramètres :

- `target` : la somme cible à atteindre,
- `nb_dice` : le nombre de dés,
- `sides` : le nombre de faces sur chaque dé.

On convient que par défaut `nb_dice=2` et `sides=6`, qui correspond au cas habituel.

Dans ce cas-là par exemple, on voit, en comptant la longueur des diagonales sur la figure, que `dice(7)` doit valoir 6, puisque le tableau comporte 6 cases contenant 7 sur la diagonale.

```
[3]: # voici quelques exemples pour la fonction dice
     exo_dice.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

À nouveau, on demande explicitement ici un parcours de type force brute.

Pour devancer les remarques sur le forum de discussion :

- ce n'est pas parce cette semaine on étudie numpy que vous devez vous sentir obligé de le faire en numpy.
- vous pouvez même vous donner comme objectif de le faire deux fois, avec et sans numpy :)

```
[4]: # à vous de jouer
     def dice(target, nb_dice=2, sides=6):
         return "votre code"

     # NOTE:
     # auto-exec-for-latex has used hidden code instead
```

```
[ ]: # pour corriger votre code
     exo_dice.correction(dice)

     # NOTE
     # auto-exec-for-latex has skipped execution of this cell
```

```
[1]: %load_ext autoreload
     %autoreload 2
```

The autoreload extension is already loaded. To reload it, use:

```
%reload_ext autoreload
```

7.20

w7-s05-x5-matdiag

construire une matrice diagonale

```
[2]: import numpy as np
```

On vous demande d'écrire une fonction `matdiag` qui

1. accepte un paramètre qui est une liste de flottants $[x_1, x_2, \dots, x_n]$
2. retourne une matrice carrée diagonale dont les éléments valent

$$m_{ii} = x_i$$

$$m_{ij} = 0 \text{ pour } ij$$

Quelques précisions :

- il est raisonnable de retourner toujours un tableau de type `float64`
- vous n'avez pas besoin de vérifier que l'appelant passe au moins un paramètre, ou dit autrement, les jeux de tests n'essaient pas d'appeler la fonction sans argument.

```
[3]: # c'est ce qu'on voit sur cet exemple

from corrections.exo_matdiag import exo_matdiag

exo_matdiag.example()
```

GridBox(children=(HTML(value='appel', _dom_classes=('header

```
[4]: # à vous de jouer
def matdiag(liste):
    ...
```

```
[ ]: exo_matdiag.correction(matdiag)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

7.20.1 Indices

Vous trouverez dans les solutions 3 façons d'implémenter cette fonction ; elles utilisent respectivement :
une approche naïve, une approche à base de slicing, et une approche à base d'une fonction prédéfinie dans numpy.

7.21 w7-s05-x6-xixj

remplir une matrice : $m(i, j) = x_i * x_j$

```
[1]: import numpy as np
```

On vous demande d'écrire une fonction `xixj` qui

1. accepte un nombre quelconque de paramètres, x_1, x_2, \dots, x_n , tous des flottants
2. retourne une matrice carrée symétrique M dont les termes valent

$$m_{ij} = x_i \cdot x_j$$

Vous n'avez pas besoin de vérifier que l'appelant passe au moins un paramètre, ou dit autrement, les jeux de tests n'essaient pas d'appeler la fonction sans argument.


```
[2]: # c'est ce qu'on voit sur cet exemple

from corrections.exo_xixj import exo_xixj

exo_xixj.example()
```

```
GridBox(children=(HTML(value='<span style="font-size:medium;"\>appel</span>', _dom_classes=('header
```

```
[3]: # à vous de jouer
# n'oubliez pas de déclarer les paramètres de votre fonction
def xixj():
    ...
```

```
[ ]: exo_xixj.correction(xixj)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

7.21.1 Indices

Vous trouverez dans les solutions 3 façons d'implémenter cette fonction ; elles utilisent respectivement : l'opérateur `@`, la méthode `array.dot()`, le broadcasting.

Souvenez vous que la transposée d'une matrice peut être obtenue en numpy avec l'attribut `.T` :

```
[4]: ligne = np.array([1, 2, 3])
      ligne.reshape(3, 1)
```

```
[4]: array([[1],
           [2],
           [3]])
```

```
[5]: ligne.T
```

```
[5]: array([1, 2, 3])
```

7.22 w7-s06-c1-data-science

La data science en général

7.22.1 et en Python en particulier

7.22.2 Complément - niveau intermédiaire

Qu'est-ce qu'un data scientist ?

J'aimerais commencer cette séquence par quelques réflexions générales sur ce qu'on appelle data science. Ce mot valise, récemment devenu à la mode, et que tout le monde veut ajouter à son CV, est un domaine qui regroupe tous les champs de l'analyse scientifique des données. Cela demande donc, pour être fait sérieusement, de maîtriser :

1. un large champ de connaissances scientifiques, notamment des notions de statistiques appliquées ;
2. les données que vous manipulez ;
3. un langage de programmation pour automatiser les traitements.

Statistiques appliquées Pour illustrer le premier point, pour quelque chose d'aussi simple qu'une moyenne, il est déjà possible de faire des erreurs. Quel intérêt de considérer une moyenne d'une distribution bimodale ?

Par exemple, j'ai deux groupes de personnes et je veux savoir lequel a le plus de chance de gagner à une épreuve de tir à la corde. L'âge moyen de mon groupe A est de 55 ans, l'âge moyen de mon groupe B est de 30 ans. Il me semble alors pouvoir affirmer que le groupe B a plus de chances de gagner. Seulement, dans le groupe B il y a 10 enfants de 5 ans et 10 personnes de 55 ans et dans le groupe A j'ai une population homogène de 20 personnes ayant 55 ans. Finalement, ça sera sans doute le groupe A qui va gagner.

Quelle erreur ai-je faite ? J'ai utilisé un outil statistique qui n'était pas adapté à l'analyse de mes groupes de personnes. Cette erreur peut vous paraître stupide, mais ces erreurs peuvent être très subtiles voire extrêmement difficiles à identifier.

Connaissance des données C'est une des parties les plus importantes, mais largement sous estimées : analyser des données sur lesquelles on n'a pas d'expertise est une aberration. Le risque principal est d'ignorer l'existence d'un facteur caché, ou de supposer à tort l'indépendance des données (sachant que nombre d'outils statistiques ne fonctionnent que sur des données indépendantes). Sans rentrer plus dans le détail, je vous conseille de lire cet article de [David Louapre sur le paradoxe de Simpson et la vidéo associée](#), pour vous donner l'intuition que travailler sur des données qu'on ne maîtrise pas peut conduire à d'importantes erreurs d'interprétation.

Maîtrise d'un langage de programmation Comme vous l'avez sans doute compris, le succès grandissant de la data science est dû à la démocratisation d'outils informatiques comme R, ou la suite d'outils disponibles dans Python, dont nous abordons certains aspects cette semaine.

Il y a ici cependant de nouveau des difficultés. Comme nous allons le voir il est très facile de faire des erreurs qui seront totalement silencieuses, par conséquent, vous obtiendrez presque toujours un résultat, mais totalement faux. Sans une profonde compréhension des mécanismes et des implémentations, vous avez la garantie de faire n'importe quoi.

Vous le voyez, je ne suis pas très encourageant, pour faire de la data science vous devrez maîtriser la bases des outils statistiques, comprendre les données que vous manipulez et maîtriser parfaitement les outils que vous utilisez. Beaucoup de gens pensent qu'en faisant un peu de R ou de Python on peut s'affirmer data scientist, c'est faux, et si vous êtes, par exemple, journaliste ou économiste et que vos résultats ont un impact politique, vous avez une vraie responsabilité et vos erreurs peuvent avoir d'importantes conséquences.

Présentation de **pandas**

numpy est l'outil qui permet de manipuler des tableaux en Python, et **pandas** est l'outil qui permet d'ajouter des index à ces tableaux. Par conséquent, **pandas** repose entièrement sur **numpy** et toutes les données que vous manipulez en **pandas** sont des tableaux **numpy**.

pandas est un projet qui évolue régulièrement, on vous recommande donc d'utiliser au moins **pandas** dans sa version 0.21. Voici les versions que l'on utilise ici.

```
[1]: import numpy as np
      print(f"numpy version {np.__version__}")
```

```
import pandas as pd
print(f"pandas version {pd.__version__}")
```

```
numpy version 1.17.0
pandas version 0.25.0
```

Il est important de comprendre que le monde de la data science en Python suit un autre paradigme que Python. Là où Python favorise la clarté, la simplicité et l'uniformité, **numpy** and **pandas** favorisent l'efficacité. La conséquence est une augmentation de la complexité et une moins bonne uniformité. Aussi, personne ne joue le rôle de BDFL dans la communauté data science comme le fait Guido van Rossum pour Python. Nous entrons donc largement dans une autre philosophie que celle de Python.

Les structures de données en **pandas** Il y a deux structures de données principales en **pandas**, la classe **Series** et la classe **DataFrame**. Une **Series** est un tableau à une dimension où chaque élément est indexé avec essentiellement un autre array (souvent de chaînes de caractères), et une **DataFrame** est un tableau à deux dimensions où les lignes et les colonnes sont indexées. La clef ici est de comprendre que l'intérêt de **pandas** est de pouvoir manipuler les tableaux **numpy** qui sont indexés, et le travail de **pandas** est de rendre les opérations sur ces index très efficaces.

Vous pouvez bien sûr vous demander à quoi cela sert, alors regardons un petit exemple. Nous allons revenir sur les notions utilisées dans cet exemple, notre but ici est de vous montrer l'utilité de **pandas** sur un exemple.

```
[2]: # seaborn est un module pour dessiner des courbes qui améliore
# sensiblement matplotlib, mais ça n'est pas ce qui nous intéresse ici.
# seaborn vient avec quelques jeux de données sur lesquels on peut jouer.
import seaborn as sns

# chargeons un jeu de données qui représente des pourboires
tips = sns.load_dataset('tips')
```

`load_dataset` retourne une **DataFrame**.

```
[3]: type(tips)
```

```
[3]: pandas.core.frame.DataFrame
```

Regardons maintenant à quoi ressemble une **DataFrame** :

```
[4]: # voici à quoi ressemblent ces données. On a la note totale (total_bill),
# le pourboire (tip), le sexe de la personne qui a donné le pourboire,
# si la personne est fumeur ou non fumeur (smoker), le jour du repas,
# le moment du repas (time) et le nombre de personnes à table (size)
tips.head()
```

```
[4]:   total_bill  tip  sex smoker  day  time  size
0      16.99  1.01 Female    No  Sun  Dinner    2
1      10.34  1.66  Male    No  Sun  Dinner    3
2      21.01  3.50  Male    No  Sun  Dinner    3
3      23.68  3.31  Male    No  Sun  Dinner    2
4      24.59  3.61 Female    No  Sun  Dinner    4
```

On voit donc un exemple de **DataFrame** qui représente des données indexées, à la fois par des labels sur les colonnes, et par un rang entier sur les lignes. C'est l'utilisation de ces index qui va nous permettre de faire des requêtes expressives sur ces données.

```
[5]: # commençons par une rapide description statistique de ces données
tips.describe()
```

```
[5]:      total_bill      tip      size
count  244.000000  244.000000  244.000000
mean    19.785943    2.998279    2.569672
std      8.902412    1.383638    0.951100
min      3.070000    1.000000    1.000000
25%     13.347500    2.000000    2.000000
50%     17.795000    2.900000    2.000000
75%     24.127500    3.562500    3.000000
max     50.810000   10.000000    6.000000
```

```
[6]: # prenons la moyenne par sexe
tips.groupby('sex').mean()
```

```
[6]:      total_bill      tip      size
sex
Male    20.744076    3.089618    2.630573
Female  18.056897    2.833448    2.459770
```

```
[7]: # et maintenant la moyenne par jour
tips.groupby('day').mean()
```

```
[7]:      total_bill      tip      size
day
Thur    17.682742    2.771452    2.451613
Fri     17.151579    2.734737    2.105263
Sat     20.441379    2.993103    2.517241
Sun     21.410000    3.255132    2.842105
```

```
[8]: # et pour finir la moyenne par moment du repas
tips.groupby('time').mean()
```

```
[8]:      total_bill      tip      size
time
Lunch    17.168676    2.728088    2.411765
Dinner   20.797159    3.102670    2.630682
```

Vous voyez qu'en quelques requêtes simples et intuitives (nous reviendrons bien sûr sur ces notions) on peut grâce à la notion d'index, obtenir des informations précieuses sur nos données. Vous voyez qu'en l'occurrence, travailler directement sur le tableau **numpy** aurait été beaucoup moins aisé.

Conclusion

Nous avons vu que la data science est une discipline complexe qui demande de nombreuses compétences. Une de ces compétences est la maîtrise d'un langage de programmation, et à cet égard la suite data science de Python qui se base sur **numpy** et **pandas** offre une solution très performante.

Il nous reste une dernière question à aborder : R ou la suite data science de Python ?

Notre préférence va bien évidemment à la suite data science de Python parce qu'elle bénéficie de toute la puissance de Python. R est un langage dédié à la statistique qui n'offre pas la puissance d'un langage générique comme Python. Mais dans le contexte de la data science, R et la suite data science de Python sont deux excellentes solutions. À très grosse maille, la syntaxe de R est plus complexe que celle de Python, par contre, R est très utilisé par les statisticiens, il peut donc avoir une implémentation d'un nouvel algorithme de l'état de l'art plus rapidement que la suite data science de Python.

7.23

w7-s06-c2-Series

Series de pandas

7.23.1 Complément - niveau intermédiaire

Création d'une **Series**

Un objet de type **Series** est un tableau **numpy** à une dimension avec un index, par conséquent, une **Series** a une certaine similarité avec un dictionnaire, et peut d'ailleurs être directement construite à partir de ce dictionnaire. Notons que, comme pour un dictionnaire, l'accès ou la modification est en $O(1)$, c'est-à-dire à temps constant indépendamment du nombre d'éléments dans la **Series**.

```
[1]: # Regardons la construction d'une Series
import numpy as np
import pandas as pd

# à partir d'un itérable
s = pd.Series([x**2 for x in range(10)])
print(s)
```

```
0    0
1    1
2    4
3    9
4   16
5   25
6   36
7   49
8   64
9   81
dtype: int64
```

```
[2]: # en contrôlant maintenant le type
s = pd.Series([x**2 for x in range(10)], dtype='int8')
print(s)
```

```
0    0
1    1
2    4
3    9
4   16
5   25
6   36
7   49
8   64
9   81
```

```
dtype: int8
```

```
[3]: # en définissant un index, par défaut l'index est un rang démarrant à 0
s = pd.Series([x**2 for x in range(10)],
              index=[x for x in 'abcdefghij'],
              dtype='int8',
              )
print(s)
```

```
a    0
b    1
c    4
d    9
e   16
f   25
g   36
h   49
i   64
j   81
dtype: int8
```

```
[4]: # et directement à partir d'un dictionnaire,
# les clefs forment l'index
d = {k:v**2 for k, v in zip('abcdefghij', range(10))}
print(d)
```

```
{'a': 0, 'b': 1, 'c': 4, 'd': 9, 'e': 16, 'f': 25, 'g': 36, 'h': 49, 'i': 64, 'j': 81}
```

```
[5]: s = pd.Series(d, dtype='int8')
print(s)
```

```
a    0
b    1
c    4
d    9
e   16
f   25
g   36
h   49
i   64
j   81
dtype: int8
```

Évidemment, l'intérêt d'un index est de pouvoir accéder à un élément par son index, comme nous aurons l'occasion de le revoir :

```
[6]: print(s['f'])
```

```
25
```

Index

L'index d'une `Series` est un objet implémenté sous la forme d'un `ndarray` de `numpy`, mais qui ne peut contenir que des objets hashables (pour garantir la performance de l'accès).

```
[7]: # pour accéder à l'index d'un objet Series
      # attention, index est un attribut, pas une fonction
      print(s.index)
```

```
Index(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'], dtype='object')
```

L'index va également supporter un certain nombre de méthodes qui vont faciliter son utilisation. Pour plus de détails, voyez [la documentation de l'objet Index](#) et de ses sous-classes.

L'autre moitié de l'objet **Series** est accessible via l'attribut **values**. ATTENTION à nouveau ici, c'est un attribut de l'objet et non pas une méthode, ce qui est très troublant par rapport à l'interface d'un dictionnaire.

```
[8]: # regardons les valeurs de ma Series
      # ATTENTION !! values est un attribut, pas une fonction
      print(s.values)
```

```
[ 0  1  4  9 16 25 36 49 64 81]
```

Mais une **Series** a également une interface de dictionnaire à laquelle on accède de la manière suivante :

```
[9]: # les clefs correspondent à l'index
      k = s.keys() # attention ici c'est un appel de fonction !
      print(f"Les clefs: {k}")
```

```
Les clefs: Index(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'], dtype=
              'object')
```

```
[10]: # et les couples (clefs, valeurs) sous forme d'un objet zip
        for k,v in s.items(): # attention ici aussi c'est un appel de fonction !
            print(k, v)
```

```
a 0
b 1
c 4
d 9
e 16
f 25
g 36
h 49
i 64
j 81
```

```
[11]: # pour finir remarquons que le test d'appartenance est possible sur les index
        print(f"Est-ce que a est dans s ? {'a' in s}")
        print(f"Est-ce que z est dans s ? {'z' in s}")
```

```
Est-ce que a est dans s ? True
Est-ce que z est dans s ? False
```

Vous remarquerez ici qu'alors que **values** et **index** sont des attributs de la **Series**, **keys()** et **items()** sont des méthodes. Voici un exemple des nombreuses petites incohérences de **pandas** avec lesquelles il faut vivre.

Pièges à éviter

Avant d'aller plus loin, il faut faire attention à la gestion du type des objets contenus dans notre **Series** (on aura le même problème avec les **DataFrame**). Alors qu'un **ndarray** de **numpy** a un type qui ne change pas, une **Series** peut implicitement changer le type de ses valeurs lors d'opérations d'affectations.

```
[12]: # créons une Series et regardons le type de ses valeurs
s = pd.Series({k:v**2 for k, v in zip('abcdefghij', range(10))})
print(s.values.dtype)
```

int64

```
[13]: # On a déjà vu que l'on ne pouvait pas modifier lors d'une affectation le
# type d'un ndarray numpy

try:
    s.values[2] = 'spam'
except ValueError as e:
    print(f"On ne peut pas affecter une str à un ndarray de int64:\n{e}")
```

On ne peut pas affecter une str à un ndarray de int64:
invalid literal for int() with base 10: 'spam'

```
[14]: # Par contre, on peut le faire sur une Series
s['c'] = 'spam'

# et maintenant le type des valeurs de la Series a changé
print(s.values.dtype)
```

object

C'est un point extrêmement important puisque toutes les opérations vectorisées vont avoir leur performance impactée et le résultat obtenu peut même être faux. Regardons cela :

```
[15]: s = pd.Series(range(10_000))
print(s.values.dtype)
```

int64

```
[16]: # combien de temps prend le calcul du carré des valeurs
%timeit s**2
```

104 μ s \pm 2.34 μ s per loop (mean \pm std. dev. of 7 runs, 10000 loops each)

```
[17]: # ajoutons 'spam' à la fin de la Series
s[10_000] = 'spam'

# oups, je me suis trompé, enlevons cet élément
del s[10_000]

# calculons de nouveau le temps de calcul pour obtenir le carré des valeurs
%timeit s**2
```

2.58 ms \pm 22.2 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)


```
[18]: # que se passe-t-il, pourquoi le calcul est maintenant plus long
      s.values.dtype
```

```
[18]: dtype('O')
```

Maintenant, les opérations vectorisées le sont sur des objets Python et non plus sur des int64, il y a donc un impact sur la performance.

Et on peut même obtenir un résultat carrément faux. Regardons cela :

```
[19]: # créons une series de trois entiers
      s = pd.Series([1, 2, 3])
      print(s)
```

```
0    1
1    2
2    3
dtype: int64
```

```
[20]: # puis ajoutons un nouvel élément, mais ici je me trompe, c'est une str
      # au lieu d'un entier
      s[3] = '4'

      # à part le type qui pourrait attirer mon attention, rien dans l'affichage
      # ne distingue les entiers de la str, à part le dtype
      print(s)
```

```
0    1
1    2
2    3
3    4
dtype: object
```

```
[21]: # seulement si j'additionne, les entiers sont additionnés,
      # mais les chaînes de caractères concaténées.
      print(s+s)
```

```
0    2
1    4
2    6
3   44
dtype: object
```

Alignement d'index

Un intérêt majeur de `pandas` est de faire de l'alignement d'index sur les objets que l'on manipule. Regardons un exemple :

```
[22]: argent_poche_janvier = pd.Series([30, 35, 20],
                                       index=['alice', 'bob', 'julie'])
      argent_poche_février = pd.Series([30, 35, 20],
                                       index=['alice', 'julie', 'sonia'])
      argent_poche_janvier + argent_poche_février
```

```
[22]: alice    60.0
      bob      NaN
      julie    55.0
      sonia     NaN
      dtype: float64
```

On voit que les deux **Series** ont bien été alignées, mais on a un problème. Lorsqu'une valeur n'est pas définie, elle vaut NaN et si on ajoute NaN à une autre valeur, le résultat est NaN. On peut corriger ce problème avec un appel explicite de la fonction `add` qui accepte un argument `fill_value` qui sera la valeur par défaut en cas d'absence d'une valeur lors de l'opération.

```
[23]: argent_poche_janvier.add(argent_poche_février, fill_value=0)
```

```
[23]: alice    60.0
      bob     35.0
      julie    55.0
      sonia    20.0
      dtype: float64
```

Accès aux éléments d'une **Series**

Comme les **Series** sont basées sur des `ndarray` de `numpy`, elles supportent les opérations d'accès aux éléments des `ndarray`, notamment la notion de masque et les broadcasts, tout ça en conservant évidemment les index.

```
[24]: s = pd.Series([30, 35, 20], index=['alice', 'bob', 'julie'])

      # qui a plus de 25 ans
      print(s[s>25])
```

```
alice    30
bob      35
dtype: int64
```

```
[25]: # regardons uniquement 'alice' et 'julie'
      print(s[['alice', 'julie']])
```

```
alice    30
julie    20
dtype: int64
```

```
[26]: # et affectons sur un masque
      s[s<=25] = np.NaN
      print(s)
```

```
alice    30.0
bob      35.0
julie     NaN
dtype: float64
```

```
[27]: # notons également, que naturellement les opérations de broadcast
      # sont supportées
      s = s + 10
```

```
print(s)
```

```
alice    40.0
bob      45.0
julie    NaN
dtype: float64
```

Slicing sur les **Series**

L'opération de slicing sur les **Series** est une source fréquente d'erreur qui peut passer inaperçue pour les raisons suivantes :

- on peut slicer sur les labels des index, mais aussi sur la position (l'indice) d'un élément dans la **Series**;
- les opérations de slices sur les positions et les labels se comportent différemment, [un slice sur les positions exclut la borne de droite \(comme tous les slices en Python\)](#), [mais un slice sur un label inclut la borne de droite](#);
- il peut y avoir ambiguïté entre un label et la position d'un élément lorsque le label est un entier.

Nous allons détailler chacun de ces cas, mais sachez qu'il existe une solution qui évite toute ambiguïté, c'est d'utiliser les interfaces `loc` et `iloc` que nous verrons un peu plus loin.

Regardons maintenant ces différents problèmes :

```
[28]: s = pd.Series([30, 35, 20, 28], index=['alice', 'bob', 'julie', 'sonia'])
      print(s)
```

```
alice    30
bob      35
julie    20
sonia    28
dtype: int64
```

```
[29]: # on peut accéder directement à la valeur correspondant à alice
      print(s['alice'])

      # mais aussi par la position d'alice dans l'index
      print(s[0])
```

```
30
30
```

```
[30]: # On peut faire un slice sur les labels, dans ce cas la borne
      # de droite est incluse
      s['alice':'julie']
```

```
[30]: alice    30
      bob      35
      julie    20
      dtype: int64
```

```
[31]: # et on peut faire un slice sur les positions, mais dans ce cas
      # la borne de droite est exclue, comme un slice normal en Python
      s[0:2]
```

```
[31]: alice    30
      bob      35
      dtype: int64
```

Ce comportement mérite quelques explications. On voit bien qu'exclure la borne de droite peut se comprendre sur une position (si on exclut `i` on prend `i-1`), par contre, c'est mal défini pour un label.

En effet, l'ordre d'un index est défini au moment de sa création et le label venant juste avant un autre label `L` ne peut pas être trouvé uniquement avec la connaissance de `L`.

C'est pour cette raison que les concepteurs de `pandas` ont préféré inclure la borne de droite.

Regardons maintenant plus en détail cette notion d'ordre sur les index.

```
[32]: # Regardons le slice sur un index avec un ordre particulier
s = pd.Series([30, 35, 20, 28], index=['alice', 'bob', 'julie', 'sonia'])
print(s['alice':'julie'])
```

```
alice    30
bob      35
julie    20
dtype: int64
```

```
[33]: # Si on change l'ordre de l'index, ça change la signification du slice
s = pd.Series([30, 35, 20, 28], index=['alice', 'bob', 'sonia', 'julie'])
print(s['alice':'julie'])
```

```
alice    30
bob      35
sonia    20
julie    28
dtype: int64
```

Vous devez peut-être vous demander si un slice sur l'index est toujours défini. La réponse est non ! Pour qu'un slice soit défini sur un index, il faut que l'index ait une croissance monotone ou qu'il n'y ait pas de label dans l'index qui soit dupliqué.

Donc la croissance monotone n'est pas nécessaire tant qu'il n'y a pas de duplication de labels. Regardons cela.

```
[34]: # mon index a des labels dupliqués, mais a une croissance monotone
s = pd.Series([30, 35, 20, 12], index=['a', 'a', 'b', 'c'])
# le slice est défini
s['a': 'b']
```

```
[34]: a    30
      a    35
      b    20
      dtype: int64
```

```
[35]: # mon index a des labels dupliqués et n'a pas de croissance monotone
s = pd.Series([30, 35, 20, 12], index=['a', 'b', 'c', 'a'])
# le slice n'est plus défini
try:
    s['a': 'b']
except KeyError as e:
```

```
print(f"Je n'arrive pas à extraire un slice :\n{e}")
```

Je n'arrive pas à extraire un slice :

```
"Cannot get left slice bound for non-unique label: 'a'"
```

Pour finir sur les problèmes que l'on peut rencontrer avec les slices, que se passe-t-il si on a un index qui a pour label des entiers ?

Lorsque l'on va faire un slice, il va y avoir ambiguïté entre la position du label et le label lui-même. Dans ce cas, `pandas` donne la priorité à la position, mais ce qui est troublant, c'est que lorsqu'on accède à un seul élément en dehors d'un slice, `pandas` donne la priorité à l'index.

Encore une petite incohérence :

```
[36]: s = pd.Series(['a', 'b', 'c'], index=[2, 0, 1])
      print(f"Si on accède directement à un élément, priorité au label : {s[0]}")
      print(f"Si on calcule un slice, priorité à la position : {s[0:1]}")
```

Si on accède directement à un élément, priorité au label : b

Si on calcule un slice, priorité à la position : 2 a

```
dtype: object
```

`loc` et `iloc`

La solution à tous ces problèmes est de dire explicitement ce que l'on veut faire. On peut en effet dire explicitement si l'on veut utiliser les labels ou les positions, c'est ce qu'on vous recommande de faire pour éviter les comportements implicites.

Pour utiliser les labels il faut utiliser `s.loc[]` et pour utiliser les positions il faut utiliser `s.iloc[]` (le `i` est pour localisation implicite, c'est-à-dire la position). Regardons cela :

```
[37]: print(s)
```

```
2    a
0    b
1    c
dtype: object
```

```
[38]: # accès au label
      print(s.loc[0])
```

b

```
[39]: # accès à la position
      print(s.iloc[0])
```

a

```
[40]: # slice sur les labels, ATTENTION, il inclut la borne de droite
      print(s.loc[2:0])
```

```
2    a
0    b
dtype: object
```

```
[41]: # slice sur les positions, ATTENTION, il exclut la borne de droite
      print(s.iloc[0:2])
```

```
2    a
0    b
dtype: object
```

Pour aller plus loin, vous pouvez lire la documentation officielle :

<http://pandas.pydata.org/pandas-docs/stable/indexing.html>

Conclusion

Nous avons vu que les **Series** forment une extension des `ndarray` de dimension 1, en leur ajoutant un index qui permet une plus grande expressivité pour accéder aux éléments. Seulement cette expressivité vient au prix de quelques subtilités (conversions implicites de type, accès aux labels ou aux positions) qu'il faut maîtriser.

Nous verrons dans le prochain complément la notion de **DataFrame** qui est sans doute la plus utile et la plus puissante structure de données de **pandas**. Tous les pièges que nous avons vus pour les **Series** sont valables pour les **DataFrames**.

7.24 w7-s07-c1-DataFrame

DataFrame de pandas

7.24.1 Complément - niveau intermédiaire

Création d'une **DataFrame**

Une **DataFrame** est un tableau **numpy** à deux dimensions avec un index pour les lignes et un index pour les colonnes. Il y a de nombreuses manières de construire une **DataFrame**.

```
[1]: # Regardons la construction d'une DataFrame
      import numpy as np
      import pandas as pd

      # Créons une Series pour définir des âges
      age = pd.Series([30, 20, 50], index=['alice', 'bob', 'julie'])

      # et une Series pour définir des tailles
      height = pd.Series([150, 170, 168], index=['alice', 'marc', 'julie'])

      # On peut maintenant combiner ces deux Series en DataFrame,
      # chaque Series définissant une colonne, une manière de le faire est
      # de définir un dictionnaire qui contient pour clef le nom de la colonne
      # et pour valeur la Series correspondante
      stat = pd.DataFrame({'age': age, 'height': height})
      print(stat)
```

```
      age  height
alice  30.0    150.0
bob    20.0      NaN
```

```
julie  50.0   168.0
marc    NaN   170.0
```

On remarque que **pandas** fait automatiquement l'alignement des index, lorsqu'une valeur n'est pas présente, elle est automatiquement remplacée par **NaN**. **Panda** va également broadcaster une valeur unique définissant une colonne sur toutes les lignes. Regardons cela :

```
[2]: stat = pd.DataFrame({'age': age, 'height': height, 'city': 'Nice'})
      print(stat)
```

```
      age  height  city
alice  30.0   150.0  Nice
bob    20.0    NaN  Nice
julie  50.0   168.0  Nice
marc   NaN   170.0  Nice
```

```
[3]: # On peut maintenant accéder aux index des lignes et des colonnes

      # l'index des lignes
      print(stat.index)
```

```
Index(['alice', 'bob', 'julie', 'marc'], dtype='object')
```

```
[4]: # l'index des colonnes
      print(stat.columns)
```

```
Index(['age', 'height', 'city'], dtype='object')
```

Il y a de nombreuses manières d'accéder aux éléments de la **DataFrame**, certaines sont bonnes et d'autres à proscrire, commençons par prendre de bonnes habitudes. Comme il s'agit d'une structure à deux dimensions, il faut donner un indice de ligne et de colonne :

```
[5]: # Quel est l'âge de alice
      a = stat.loc['alice', 'age']
```

```
[6]: # a est un flottant
      type(a), a
```

```
[6]: (numpy.float64, 30.0)
```

```
[7]: # Quel est la moyenne de tous les âges
      c = stat.loc[:, 'age']
      m = c.mean()
      print(f"L'âge moyen est de {m:.1f} ans.")
```

```
L'âge moyen est de 33.3 ans.
```

```
[8]: # c est une Series
      type(c)
```

```
[8]: pandas.core.series.Series
```

```
[9]: # et m est un flottant
      type(m)
```

```
[9]: numpy.float64
```

On peut déjà noter plusieurs choses intéressantes :

- On peut utiliser `.loc[]` et `.iloc` comme pour les **Series**. Pour les **DataFrame** c'est encore plus important parce qu'il y a plus de risques d'ambiguïtés (notamment entre les lignes et les colonnes, on y reviendra) ;
- la méthode `mean` calcule la moyenne, ça n'est pas surprenant, mais ignore les **NaN**. C'est en général ce que l'on veut. Si vous vous demandez comment savoir si la méthode que vous utilisez ignore ou pas les **NaN**, le mieux est de regarder l'aide de cette méthode. Il existe pour un certain nombre de méthodes deux versions : une qui ignore les **NaN** et une autre qui les prend en compte ; on en reparlera.

Une autre manière de construire une **DataFrame** est de partir d'un **array** de **numpy**, et de spécifier les index pour les lignes et les colonnes avec les arguments `index` et `columns` :

```
[10]: a = np.random.randint(1, 20, 9).reshape(3, 3)
      p = pd.DataFrame(a, index=['a', 'b', 'c'], columns=['x', 'y', 'z'])
      print(p)
```

```
      x  y  z
a      3  5 11
b     13  9  6
c      7  9  4
```

Importation et exportation de données

En pratique, il est très fréquent que les données qu'on manipule soient stockées dans un fichier ou une base de données. Il existe en **pandas** de nombreux utilitaires pour importer et exporter des données et les convertir automatiquement en **DataFrame**. Vous pouvez importer ou exporter du CSV, JSON, HTML, Excel, HDF5, SQL, Python pickle, etc.

À titre d'illustration écrivons la **DataFrame** `p` dans différents formats.

```
[11]: # écrivons notre DataFrame dans un fichier CSV
      p.to_csv('my_data.csv')
      !cat my_data.csv
```

```
,x,y,z
a,3,5,11
b,13,9,6
c,7,9,4
```

```
[12]: # et dans un fichier JSON
      p.to_json('my_data.json')
      !cat my_data.json
```



```
[13]: # on peut maintenant recharger notre fichier
# la conversion en DataFrame est automatique
new_p = pd.read_json('my_data.json')
print(new_p)
```

```
   x  y  z
a   3  5 11
b  13  9  6
c   7  9  4
```

Pour la gestion des autres formats, comme il s'agit de quelque chose de très spécifique et sans difficulté particulière, je vous renvoie simplement à la documentation :

<http://pandas.pydata.org/pandas-docs/stable/io.html>

Manipulation d'une **DataFrame**

```
[14]: # construisons maintenant une DataFrame jouet

# voici une liste de prénoms
names = ['alice', 'bob', 'marc', 'bill', 'sonia']

# créons trois Series qui formeront les trois colonnes
age = pd.Series([12, 13, 16, 11, 16], index=names)
height = pd.Series([130, 140, 176, 120, 165], index=names)
sex = pd.Series(list('fmmmf'), index=names)

# créons maintenant la DataFrame
p = pd.DataFrame({'age': age, 'height': height, 'sex': sex})
print(p)
```

```
   age  height sex
alice  12    130  f
bob    13    140  m
marc   16    176  m
bill   11    120  m
sonia  16    165  f
```

```
[15]: # et chargeons le jeu de données sur les pourboires de seaborn
import seaborn as sns
tips = sns.load_dataset('tips')
```

pandas offre de nombreuses possibilités d'explorer les données. Attention, dans mes exemples je vais alterner entre le **DataFrame** `p` et le **DataFrame** `tips` suivant les besoins de l'explication.

```
[16]: # afficher les premières lignes
tips.head()
```

```
[16]:   total_bill  tip  sex smoker  day  time  size
0      16.99  1.01 Female    No  Sun  Dinner    2
1      10.34  1.66  Male    No  Sun  Dinner    3
2      21.01  3.50  Male    No  Sun  Dinner    3
3      23.68  3.31  Male    No  Sun  Dinner    2
4      24.59  3.61 Female    No  Sun  Dinner    4
```

```
[17]: # et les dernière lignes
tips.tail()
```

```
[17]:      total_bill  tip    sex smoker  day    time  size
239      29.03  5.92  Male     No   Sat  Dinner    3
240      27.18  2.00 Female    Yes   Sat  Dinner    2
241      22.67  2.00  Male     Yes   Sat  Dinner    2
242      17.82  1.75  Male     No   Sat  Dinner    2
243      18.78  3.00 Female    No  Thur  Dinner    2
```

```
[18]: # l'index des lignes
p.index
```

```
[18]: Index(['alice', 'bob', 'marc', 'bill', 'sonia'], dtype='object')
```

```
[19]: # et l'index des colonnes
p.columns
```

```
[19]: Index(['age', 'height', 'sex'], dtype='object')
```

```
[20]: # et afficher uniquement les valeurs
p.values
```

```
[20]: array([[12, 130, 'f'],
        [13, 140, 'm'],
        [16, 176, 'm'],
        [11, 120, 'm'],
        [16, 165, 'f']], dtype=object)
```

```
[21]: # échanger lignes et colonnes
# cf. la transposition de matrices
p.T
```

```
[21]:      alice  bob marc bill sonia
age      12   13   16   11   16
height   130  140  176  120  165
sex       f    m    m    m    f
```

Pour finir, il y a la méthode `describe` qui permet d'obtenir des premières statistiques sur un `DataFrame`. `describe` permet de calculer des statistiques sur des type numériques, mais aussi sur des types chaînes de caractères.

```
[22]: # par défaut describe ne prend en compte que les colonnes numériques
p.describe()
```

```
[22]:      age      height
count  5.000000  5.000000
mean   13.600000 146.200000
std     2.302173  23.605084
min    11.000000 120.000000
25%    12.000000 130.000000
50%    13.000000 140.000000
75%    16.000000 165.000000
```

```
max      16.000000  176.000000
```

```
[23]: # mais on peut le forcer à prendre en compte toutes les colonnes
p.describe(include='all')
```

```
[23]:
```

	age	height	sex
count	5.000000	5.000000	5
unique	NaN	NaN	2
top	NaN	NaN	m
freq	NaN	NaN	3
mean	13.600000	146.200000	NaN
std	2.302173	23.605084	NaN
min	11.000000	120.000000	NaN
25%	12.000000	130.000000	NaN
50%	13.000000	140.000000	NaN
75%	16.000000	165.000000	NaN
max	16.000000	176.000000	NaN

Requêtes sur une **DataFrame**

On peut maintenant commencer à faire des requêtes sur les **DataFrames**. Les **DataFrame** supportent la notion de masque que l'on a vue pour les **ndarray** de **numpy** et pour les **Series**.

```
[24]: # p.loc prend soit un label de ligne
print(p.loc['sonia'])
```

```
age      16
height   165
sex       f
Name: sonia, dtype: object
```

```
[25]: # ou alors un label de ligne ET de colonne
print(p.loc['sonia', 'age'])
```

```
16
```

On peut mettre à la place d'une label :

- une liste de labels;
- un slice sur les labels;
- un masque (c'est-à-dire un tableau de booléens);
- un callable qui retourne une des trois premières possibilités.

Noter que l'on peut également utiliser la notation `.iloc[]` avec les mêmes règles, mais elle est moins utile.

Je recommande de toujours utiliser la notation `.loc[lignes, colonnes]` pour éviter toute ambiguïté. Nous verrons que les notations `.loc[lignes]` ou pire seulement `[label]` sont sources d'erreurs.

Regardons maintenant d'autres exemples plus sophistiqués :

```
[26]: # un masque sur les femmes
p.loc[:, 'sex'] == 'f'
```

```
[26]: alice      True
      bob       False
      marc      False
      bill      False
      sonia     True
      Name: sex, dtype: bool
```

```
[27]: # si bien que pour construire un tableau
      # avec uniquement les femmes
      p.loc[p.loc[:, 'sex'] == 'f', :]
```

```
[27]:      age  height sex
      alice   12    130  f
      sonia   16    165  f
```

```
[28]: # si on veut ne garder uniquement
      # que les femmes de plus de 14 ans
      p.loc[(p.loc[:, 'sex'] == 'f') & (p.loc[:, 'age'] > 14), :]
```

```
[28]:      age  height sex
      sonia   16    165  f
```

```
[29]: # quelle est la moyenne de 'total_bill' pour les femmes
      addition_f = tips.loc[tips.loc[:, 'sex'] == 'Female', 'total_bill'].mean()
      print(f"addition moyenne des femmes : {addition_f:.2f}")
```

addition moyenne des femmes : 18.06

```
[30]: # quelle est la note moyenne des hommes
      addition_h = tips.loc[tips.loc[:, 'sex'] == 'Male', 'total_bill'].mean()
      print(f"addition moyenne des hommes : {addition_h:.2f}")
```

addition moyenne des hommes : 20.74

```
[31]: # qui laisse le plus grand pourcentage de pourboire :
      # les hommes ou les femmes ?

      pourboire_f = tips.loc[tips.loc[:, 'sex'] == 'Female', 'tip'].mean()
      pourboire_h = tips.loc[tips.loc[:, 'sex'] == 'Male', 'tip'].mean()

      print(f"Les femmes laissent {pourboire_f/addition_f:.2%} de pourboire")
      print(f"Les hommes laissent {pourboire_h/addition_h:.2%} de pourboire")
```

Les femmes laissent 15.69% de pourboire
Les hommes laissent 14.89% de pourboire

Erreurs fréquentes et ambiguïtés sur les requêtes

Nous avons vu une manière simple et non ambiguë de faire des requêtes sur les `DataFrame`. Nous allons voir qu'il existe d'autres manières qui ont pour seul avantage d'être plus concises, mais sources de nombreuses erreurs.

Souvenez-vous, utilisez toujours la notation `.loc[lignes, colonnes]` sinon, soyez sûr de savoir ce qui est réellement calculé.

```
[32]: # commençons par la notation la plus classique
p['sex'] # prend forcément un label de colonne
```

```
[32]: alice    f
      bob      m
      marc     m
      bill     m
      sonia    f
      Name: sex, dtype: object
```

```
[33]: # mais par contre, si on passe un slice, c'est forcément des lignes,
      # assez perturbant et source de confusion.
p['alice': 'marc']
```

```
[33]:      age  height sex
      alice   12    130  f
      bob     13    140  m
      marc    16    176  m
```

```
[34]: # on peut même directement accéder à une colonne par son nom
p.age
```

```
[34]: alice    12
      bob      13
      marc     16
      bill     11
      sonia    16
      Name: age, dtype: int64
```

Mais c'est fortement déconseillé parce que si un attribut de même nom existe sur une `DataFrame`, alors la priorité est donnée à l'attribut, et non à la colonne :

```
[35]: # ajoutons une colonne qui a pour nom une méthode qui existe sur
      # les DataFrame
p['mean'] = 1
print(p)
```

```
      age  height sex  mean
      alice   12    130  f     1
      bob     13    140  m     1
      marc    16    176  m     1
      bill    11    120  m     1
      sonia    16    165  f     1
```

```
[36]: # je peux bien accéder
      # à la colonne sex
p.sex
```

```
[36]: alice    f
      bob      m
      marc     m
```

```
bill      m
sonia     f
Name: sex, dtype: object
```

```
[37]: # mais pas à la colonne mean
p.mean
```

```
[37]: <bound method DataFrame.mean of          age  height sex  mean
alice    12     130   f      1
bob      13     140   m      1
marc     16     176   m      1
bill     11     120   m      1
sonia    16     165   f      1>
```

```
[38]: # à nouveau, la seule méthode non ambiguë est d'utiliser .loc
p.loc[:, 'mean']
```

```
[38]: alice    1
bob      1
marc     1
bill     1
sonia    1
Name: mean, dtype: int64
```

```
[39]: # supprimons maintenant la colonne mean *en place* (par défaut,
# drop retourne une nouvelle DataFrame)
p.drop(columns='mean', inplace=True)
print(p)
```

```
      age  height sex
alice   12     130   f
bob     13     140   m
marc    16     176   m
bill    11     120   m
sonia   16     165   f
```

Pour aller plus loin, vous pouvez lire la documentation officielle :

<http://pandas.pydata.org/pandas-docs/stable/indexing.html>

Universal functions et **pandas**

Ça n'est pas une surprise, les **Series** et **DataFrame** de **pandas** supportent les **ufunc** de **numpy**. Mais il y a une subtilité. Il est parfaitement légitime et correct d'appliquer une **ufunc** de **numpy** sur les éléments d'une **DataFrame** :

```
[40]: d = pd.DataFrame(np.random.randint(
      1, 10, 9).reshape(3, 3), columns=list('abc'))
print(d)
```

```
   a  b  c
0  9  4  1
1  3  7  4
2  3  9  3
```

```
[41]: np.log(d)
```

```
[41]:
```

	a	b	c
0	2.197225	1.386294	0.000000
1	1.098612	1.945910	1.386294
2	1.098612	2.197225	1.098612

Nous remarquons que comme on s'y attend, la `ufunc` a été appliquée à chaque élément de la `DataFrame` et que les labels des lignes et colonnes ont été préservés.

Par contre, si l'on a besoin d'alignement de labels, c'est le cas avec toutes les opérations qui s'appliquent sur deux objets comme une addition, alors les `ufunc` de `numpy` ne vont pas faire ce à quoi on s'attend. Elles vont faire les opérations sur les tableaux `numpy` sans prendre en compte les labels.

Pour avoir un alignement des labels, il faut utiliser les `ufunc` de `pandas`.

```
[42]: # prenons deux Series
s1 = pd.Series([10, 20, 30],
               index=list('abc'))
print(s1)
```

```
a    10
b    20
c    30
dtype: int64
```

```
[43]: #
s2 = pd.Series([12, 22, 32],
               index=list('acd'))
print(s2)
```

```
a    12
c    22
d    32
dtype: int64
```

```
[44]: # la ufunc numpy fait la somme
# des arrays sans prendre en compte
# les labels, donc sans alignement
np.add(s1, s2)
```

```
[44]: a    22.0
      b     NaN
      c    52.0
      d     NaN
      dtype: float64
```

```
[45]: # la ufunc pandas va faire
# un alignement des labels
# cet appel est équivalent à s1 + s2
s1.add(s2)
```

```
[45]: a    22.0
      b     NaN
```

```
c    52.0
d    NaN
dtype: float64
```

```
[46]: # comme on l'a vu sur le complément précédent, les valeurs absentes sont
# remplacées par NaN, mais on peut changer ce comportement lors de
# l'appel de .add
s1.add(s2, fill_value=0)
```

```
[46]: a    22.0
b    20.0
c    52.0
d    32.0
dtype: float64
```

```
[47]: # regardons un autre exemple sur des DataFrame
# on affiche tout ça dans les cellules suivantes
names = ['alice', 'bob', 'charle']

bananas = pd.Series([10, 3, 9], index=names)
oranges = pd.Series([3, 11, 6], index=names)
fruits_jan = pd.DataFrame({'bananas': bananas, 'orange': oranges})

bananas = pd.Series([6, 1], index=names[:-1])
apples = pd.Series([8, 5], index=names[1:])
fruits_feb = pd.DataFrame({'bananas': bananas, 'apples': apples})
```

```
[48]: # ce qui donne
fruits_jan
```

```
[48]:      bananas  orange
alice         10        3
bob           3       11
charle         9        6
```

```
[49]: # et
fruits_feb
```

```
[49]:      bananas  apples
alice         6.0     NaN
bob           1.0     8.0
charle        NaN     5.0
```

```
[50]: # regardons maintenant la somme des fruits mangés
eaten_fruits = fruits_jan + fruits_feb
print(eaten_fruits)
```

```
      apples  bananas  orange
alice     NaN     16.0     NaN
bob       NaN      4.0     NaN
charle    NaN      NaN     NaN
```



```
[51]: # On a bien un alignement des labels, mais il y a beaucoup de valeurs
      # manquantes. Corrigons cela on remplaçant les valeurs manquantes par 0
      eaten_fruits = fruits_jan.add(fruits_feb, fill_value=0)
      print(eaten_fruits)
```

	apples	bananas	orange
alice	NaN	16.0	3.0
bob	8.0	4.0	11.0
charle	5.0	9.0	6.0

Notons que lorsqu'une valeur est absente dans toutes les `DataFrame`, `NaN` est conservé.

Un dernière subtilité à connaître lors de l'alignement des labels intervient lorsque vous faites une opération sur une `DataFrame` et une `Series`. `pandas` va considérer la `Series` comme une ligne et va la broadcaster sur les autres lignes. Par conséquent, l'index de la `Series` va être considéré comme des colonnes et aligné avec les colonnes de la `DataFrame`.

```
[52]: dataframe = pd.DataFrame(
      np.random.randint(1, 10, size=(3, 3)),
      columns=list('abc'), index=list('xyz'))
      dataframe
```

```
[52]:   a  b  c
      x  3  6  8
      y  5  6  2
      z  8  2  9
```

```
[53]: series_row = pd.Series(
      [100, 200, 300],
      index=list('abc'))
      series_row
```

```
[53]: a    100
      b    200
      c    300
      dtype: int64
```

```
[54]: series_col = pd.Series(
      [400, 500, 600],
      index=list('xyz'))
      series_col
```

```
[54]: x    400
      y    500
      z    600
      dtype: int64
```

```
[55]: # la Series est considérée comme une ligne et son index
      # s'aligne sur les colonnes de la DataFrame
      # la Series va être broadcastée
      # sur les autres lignes de la DataFrame

      dataframe + series_row
```

```
[55]:      a      b      c
x  103   206   308
y  105   206   302
z  108   202   309
```

```
[56]: # du coup si les labels ne correspondent pas,
      # le résultat sera le suivant

      dataframe + series_col
```

```
[56]:      a      b      c      x      y      z
x  NaN  NaN  NaN  NaN  NaN  NaN
y  NaN  NaN  NaN  NaN  NaN  NaN
z  NaN  NaN  NaN  NaN  NaN  NaN
```

```
[57]: # on peut dans ce cas, changer le comportement par défaut en forçant
      # l'alignement de la Series suivant un autre axe avec l'argument axis

      dataframe.add(series_col, axis=0)
```

```
[57]:      a      b      c
x  403   406   408
y  505   506   502
z  608   602   609
```

Ici, `axis=0` signifie que la **Series** est considérée comme une colonne et qu'elle va être broadcastée sur les autres colonnes (le long de l'axe de ligne).

Opérations sur les chaînes de caractères

Nous allons maintenant parler de la vectorisation des opérations sur les chaînes de caractères. Il y a plusieurs choses importantes à savoir :

- les méthodes sur les chaînes de caractères ne sont disponibles que pour les **Series** et les **Index**, mais pas pour les **DataFrame**;
- ces méthodes ignorent les **NaN** et remplacent les valeurs qui ne sont pas des chaînes de caractères par **NaN**;
- ces méthodes retournent une copie de l'objet (**Series** ou **Index**), il n'y a pas de modification en place;
- la plupart des méthodes Python sur le type **str** existe sous forme vectorisée;
- on accède à ces méthodes avec la syntaxe :
 - `Series.str.<vectorized method name>`
 - `Index.str.<vectorized method name>`

Regardons quelques exemples :

```
[58]: # Créons une Series avec des noms ayant une capitalisation inconsistante
      # et une mauvaise gestion des espaces
names = ['alice ', ' bob', 'Marc', 'bill', 3, ' JULIE ', np.NaN]
age = pd.Series(names)
```

```
[59]: # nettoions maintenant ces données
```

```
# on met en minuscule
a = age.str.lower()

# on enlève les espaces
a = a.str.strip()
a
```

```
[59]: 0    alice
      1     bob
      2    marc
      3    bill
      4     NaN
      5   julie
      6     NaN
      dtype: object
```

```
[60]: # comme les méthodes vectorisées retournent un objet de même type, on
      # peut les chaîner comme ceci

      [x for x in age.str.lower().str.strip()]
```

```
[60]: ['alice', 'bob', 'marc', 'bill', nan, 'julie', nan]
```

On peut également utiliser l'indexation des `str` de manière vectorisée :

```
[61]: print(a)
```

```
0    alice
1     bob
2    marc
3    bill
4     NaN
5   julie
6     NaN
      dtype: object
```

```
[62]: print(a.str[-1])
```

```
0     e
1     b
2     c
3     l
4    NaN
5     e
6    NaN
      dtype: object
```

Pour aller plus loin vous pouvez lire la documentation officielle :

<http://pandas.pydata.org/pandas-docs/stable/text.html>

Gestion des valeurs manquantes

Nous avons vu que des opérations sur les **DataFrame** pouvaient générer des valeurs **NaN** lors de l'alignement. Il est également possible d'avoir de telles valeurs manquantes dans votre jeu de données original. **pandas** offre plusieurs possibilités pour gérer correctement ces valeurs manquantes.

Avant de voir ces différentes possibilités, définissons cette notion de valeur manquante.

Une valeur manquante peut-être représentée avec **pandas** soit par **np.NaN** soit par l'objet Python **None**.

- **np.NaN** est un objet de type **float**, par conséquent il ne peut apparaître que dans un array de **float** ou un array d'**object**. Notons que **np.NaN** apparaît avec **pandas** comme simplement **NaN** et que dans la suite on utilise de manière indifférente les deux notations, par contre, dans du code, il faut obligatoirement utiliser **np.NaN** ;
 - si on ajoute un **NaN** dans un array d'entier, ils seront convertis en **float64** ;
 - si on ajoute un **NaN** dans un array de booléens, ils seront convertis en **object** ;
- **NaN** est contaminant, toute opération avec un **NaN** a pour résultat **NaN** ;
- lorsque l'on utilise **None**, il est automatiquement converti en **NaN** lorsque le type de l'array est numérique.

Illustrons ces propriétés :

```
[63]: # une Series d'entiers
s = pd.Series([1, 2])
s
```

```
[63]: 0    1
      1    2
dtype: int64
```

```
[64]: # on insère un NaN, la Series est alors convertie en float64
s[0] = np.NaN
s
```

```
[64]: 0    NaN
      1    2.0
dtype: float64
```

```
[65]: # on réinitialise
s = pd.Series([1, 2])
s
```

```
[65]: 0    1
      1    2
dtype: int64
```

```
[66]: # et on insère None
s[0] = None

# Le résultat est le même
# None est converti en NaN
s
```

```
[66]: 0    NaN
      1    2.0
      dtype: float64
```

Regardons maintenant, les méthodes de **pandas** pour gérer les valeurs manquantes (donc NaN ou None) :

- `isna()` retourne un masque mettant à **True** les valeurs manquantes (il y a un alias `isnull()`);
- `notna()` retourne un masque mettant à **False** les valeurs manquantes (il y a un alias `notnull()`);
- `dropna()` retourne un nouvel objet sans les valeurs manquantes;
- `fillna()` retourne un nouvel objet avec les valeurs manquantes remplacées.

On remarque que l'ajout d'alias pour les méthodes est de nouveau une source de confusion avec laquelle il faut vivre.

On remarque également qu'alors que `isnull()` et `notnull()` sont des méthodes simples, `dropna()` et `fillna()` impliquent l'utilisation de stratégies. Regardons cela :

```
[67]: # créons une DataFrame avec quelques valeurs manquantes
names = ['alice', 'bob', 'charles']
bananas = pd.Series([6, 1], index=names[:-1])
apples = pd.Series([8, 5], index=names[1:])
fruits_feb = pd.DataFrame({'bananas': bananas, 'apples': apples})
print(fruits_feb)
```

	bananas	apples
alice	6.0	NaN
bob	1.0	8.0
charles	NaN	5.0

```
[68]: fruits_feb.isna()
```

```
[68]:      bananas  apples
alice      False    True
bob        False    False
charles     True    False
```

```
[69]: fruits_feb.notna()
```

```
[69]:      bananas  apples
alice         True    False
bob           True     True
charles       False     True
```

Par défaut, `dropna()` va enlever toutes les lignes qui contiennent au moins une valeur manquante. Mais on peut changer ce comportement avec des arguments :

```
[70]: p = pd.DataFrame([[1, 2, np.NaN], [3, np.NaN, np.NaN], [7, 5, np.NaN]])
print(p)
```

	0	1	2
0	1	2.0	NaN
1	3	NaN	NaN
2	7	5.0	NaN

```
[71]: # comportement par défaut, j'enlève toutes les lignes avec au moins
      # une valeur manquante; il ne reste rien !
      p.dropna()
```

```
[71]: Empty DataFrame
      Columns: [0, 1, 2]
      Index: []
```

```
[72]: # maintenant, je fais l'opération par colonne
      p.dropna(axis=1)
```

```
[72]:    0
      0  1
      1  3
      2  7
```

```
[73]: # je fais l'opération par colonne si toute la colonne est manquante
      p.dropna(axis=1, how='all')
```

```
[73]:    0    1
      0  1  2.0
      1  3  NaN
      2  7  5.0
```

```
[74]: # je fais l'opération par ligne si au moins 2 valeurs sont manquantes
      p.dropna(thresh=2)
```

```
[74]:    0    1    2
      0  1  2.0 NaN
      2  7  5.0 NaN
```

Par défaut, `fillna()` remplace les valeurs manquantes avec un argument pas défaut. Mais on peut ici aussi changer ce comportement. Regardons cela :

```
[75]: print(p)
```

```
    0    1    2
0  1  2.0 NaN
1  3  NaN NaN
2  7  5.0 NaN
```

```
[76]: # je remplace les valeurs manquantes par -1
      p.fillna(-1)
```

```
[76]:    0    1    2
      0  1  2.0 -1.0
      1  3 -1.0 -1.0
      2  7  5.0 -1.0
```

```
[77]: # je remplace les valeurs manquantes avec la valeur suivante sur la colonne
      # bfill est pour back fill, c'est-à-dire remplace en arrière à partir des
      # valeurs existantes
      p.fillna(method='bfill')
```

```
[77]:      0      1      2
      0      1  2.0 NaN
      1      3  5.0 NaN
      2      7  5.0 NaN
```

```
[78]: # je remplace les valeurs manquantes avec la valeur précédente sur la ligne
      # ffill est pour forward fill, remplace en avant à partir des valeurs
      # existantes
      p.fillna(method='ffill', axis=1)
```

```
[78]:      0      1      2
      0  1.0  2.0  2.0
      1  3.0  3.0  3.0
      2  7.0  5.0  5.0
```

Regardez l'aide de ces méthodes pour aller plus loin.

```
[79]: p.dropna?
```

```
[80]: p.fillna?
```

Analyse statistique des données

Nous n'avons pas le temps de couvrir les possibilités d'analyse statistique de la suite data science de Python. **pandas** offre quelques possibilités basiques avec des calculs de moyennes, d'écart types ou de covariances que l'on peut éventuellement appliquer par fenêtres à un jeu de données. Pour avoir plus de détails dessus vous pouvez consulter cette documentation :

<http://pandas.pydata.org/pandas-docs/stable/computation.html>

Dans la suite data science de Python, il a aussi des modules spécialisés dans l'analyse statistique comme :

- [StatsModels](#)
- [ScikitLearn](#)

ou des outils de calculs scientifiques plus génériques comme [SciPy](#).

De nouveau, il s'agit d'outils appliqués à des domaines spécifiques et ils se basent tous sur le couple **numpy/pandas**.

7.24.2 Complément - niveau avancé

Les MultiIndex

pandas avait historiquement d'autres structures de données en plus des **Series** et des **DataFrame** permettant d'exprimer des dimensionnalités supérieures à 2, comme par exemple les **Panel**. Mais pour des raisons de maintenance du code et d'optimisation, les développeurs ont décidé de ne garder que les **Series** et les **DataFrame**. Alors, comment exprimer des données avec plus de deux dimensions ?

On utilise pour cela des **MultiIndex**. Un **MultiIndex** est un index qui peut être utilisé partout où l'on utilise un index (dans une **Series**, ou comme ligne ou colonne d'une **DataFrame**) et qui a pour caractéristique d'avoir plusieurs niveaux.

Comme tous types d'index, et parce qu'un `MultiIndex` est une sous classe d'`Index`, `pandas` va correctement aligner les `Series` et les `DataFrame` avec des `MultiIndex`.

Regardons tout de suite un exemple :

```
[81]: # construisons une DataFrame jouet

# voici une liste de prénoms
names = ['alice', 'bob', 'sonia']

# créons trois Series qui formeront trois colonnes
age = pd.Series([12, 13, 16], index=names)
height = pd.Series([130, 140, 165], index=names)
sex = pd.Series(list('fmf'), index=names)

# créons maintenant la DataFrame
p = pd.DataFrame({'age': age, 'height': height, 'sex': sex})
print(p)
```

	age	height	sex
alice	12	130	f
bob	13	140	m
sonia	16	165	f

```
[82]: # unstack, en première approximation, permet de passer d'une DataFrame à
# une Series avec un MultiIndex
s = p.unstack()
print(s)
```

age	alice	12
	bob	13
	sonia	16
height	alice	130
	bob	140
	sonia	165
sex	alice	f
	bob	m
	sonia	f

dtype: object

```
[83]: # et voici donc l'index de cette Series
s.index
```

```
[83]: MultiIndex([( 'age', 'alice'),
                  ( 'age', 'bob'),
                  ( 'age', 'sonia'),
                  ('height', 'alice'),
                  ('height', 'bob'),
                  ('height', 'sonia'),
                  ( 'sex', 'alice'),
                  ( 'sex', 'bob'),
                  ( 'sex', 'sonia')],
                  )
```

Il existe évidemment des moyens de créer directement un `MultiIndex` et ensuite de le définir comme index d'une `Series` ou comme index de ligne ou colonne d'une `DataFrame` :


```
[84]: # on peut créer un MultiIndex à partir d'une liste de liste
names = ['alice', 'alice', 'alice', 'bob', 'bob', 'bob']
age = [2014, 2015, 2016, 2014, 2015, 2016]
s_list = pd.Series([40, 42, 45, 38, 40, 40], index=[names, age])
print(s_list)
```

```
alice  2014    40
        2015    42
        2016    45
bob    2014    38
        2015    40
        2016    40
dtype: int64
```

```
[85]: # ou à partir d'un dictionnaire de tuples
s_tuple = pd.Series({'alice', 2014): 40,
                    ('alice', 2015): 42,
                    ('alice', 2016): 45,
                    ('bob', 2014): 38,
                    ('bob', 2015): 40,
                    ('bob', 2016): 40})

print(s_tuple)
```

```
alice  2014    40
        2015    42
        2016    45
bob    2014    38
        2015    40
        2016    40
dtype: int64
```

```
[86]: # ou avec la méthode from_product()
name = ['alice', 'bob']
year = [2014, 2015, 2016]
i = pd.MultiIndex.from_product([name, year])
s = pd.Series([40, 42, 45, 38, 40, 40], index=i)
print(s)
```

```
alice  2014    40
        2015    42
        2016    45
bob    2014    38
        2015    40
        2016    40
dtype: int64
```

On peut même nommer les niveaux d'un MultiIndex.

```
[87]: name = ['alice', 'bob']
year = [2014, 2015, 2016]
i = pd.MultiIndex.from_product([name, year], names=['name', 'year'])
s = pd.Series([40, 42, 45, 38, 40, 40], index=i)
print(s)
```

```
name    year
```

```

alice  2014    40
        2015    42
        2016    45
bob    2014    38
        2015    40
        2016    40
dtype: int64

```

```

[88]: # on peut changer le nom des niveaux du MultiIndex
s.index.names = ['NAMES', 'YEARS']
print(s)

```

```

NAMES  YEARS
alice  2014    40
        2015    42
        2016    45
bob    2014    38
        2015    40
        2016    40
dtype: int64

```

Créons maintenant une DataFrame jouet avec des MultiIndex pour étudier comment accéder aux éléments de la DataFrame.

```

[89]: index = pd.MultiIndex.from_product([[2013, 2014],
                                         [1, 2, 3]],
                                         names=['year',
                                                'visit'])

columns = pd.MultiIndex.from_product([['Bob', 'Sue'],
                                      ['avant', 'arrière']],
                                      names=['client',
                                             'pression'])

# on crée des pressions de pneus factices
data = 2 + np.random.rand(6, 4)

# on crée la DataFrame
mecanics_data = pd.DataFrame(data, index=index, columns=columns)
print(mecanics_data)

```

client		Bob		Sue	
pression		avant	arrière	avant	arrière
year	visit				
2013	1	2.327609	2.921158	2.940770	2.699021
	2	2.787798	2.772582	2.278799	2.156303
	3	2.478950	2.184081	2.809702	2.577808
2014	1	2.104409	2.118292	2.673146	2.839571
	2	2.804779	2.309748	2.461093	2.764424
	3	2.072695	2.206907	2.768370	2.480717

Il y a plusieurs manières d'accéder aux éléments, mais une seule que l'on recommande :

utilisez la notation `.loc[ligne, colonne]`, `.iloc[ligne, colonne]`.

```

[90]: # pression en 2013 pour Bob
mecanics_data.loc[2013, 'Bob']

```

```
[90]: pression    avant    arrière
      visit
      1          2.327609  2.921158
      2          2.787798  2.772582
      3          2.478950  2.184081
```

```
[91]: # pour accéder aux sous niveaux du MultiIndex, on utilise des tuples
      mechanics_data.loc[(2013, 2), ('Bob', 'avant')]
```

```
[91]: 2.7877978544233937
```

Le slice sur le MultiIndex est un peu délicat. On peut utiliser la notation : si on veut slicer sur tous les éléments d'un MultiIndex, sans prendre en compte un niveau. Si on spécifie les niveaux, il faut utiliser un objet `slice` ou `pd.IndexSlice` :

```
[92]: # slice(None) signifie tous les éléments du niveau
      print(mechanics_data.loc[slice((2013, 2), (2014, 1)), ('Sue', slice(None))])
```

```
client      Sue
pression    avant  arrière
year visit
2013 2      2.278799  2.156303
      3      2.809702  2.577808
2014 1      2.673146  2.839571
```

```
[93]: # on peut utiliser la notation : si on ne distingue par les niveaux
      print(mechanics_data.loc[(slice(None), slice(1, 2)), :])
```

```
client      Bob      Sue
pression    avant  arrière  avant  arrière
year visit
2013 1      2.327609  2.921158  2.940770  2.699021
      2      2.787798  2.772582  2.278799  2.156303
2014 1      2.104409  2.118292  2.673146  2.839571
      2      2.804779  2.309748  2.461093  2.764424
```

```
[94]: # on peut aussi utiliser pd.IndexSlice pour slicer avec une notation
      # un peu plus concise
      idx = pd.IndexSlice
      print(mechanics_data.loc[idx[:, 1:2], idx['Sue', :]])
```

```
client      Sue
pression    avant  arrière
year visit
2013 1      2.940770  2.699021
      2      2.278799  2.156303
2014 1      2.673146  2.839571
      2      2.461093  2.764424
```

Pour aller plus loin, regardez la documentation des MultiIndex :

<http://pandas.pydata.org/pandas-docs/stable/advanced.html>

7.24.3 Conclusion

La **DataFrame** est la structure de données la plus souple et la plus puissante de **pandas**. Nous avons vu comment créer des **DataFrame** et comment accéder aux éléments. Nous verrons dans le prochain complément les techniques permettant de faire des opérations complexes (et proches dans l'esprit de ce que l'on peut faire avec une base de données) comme les opérations de **merge** ou de **groupby**.

7.25

w7-s08-c1-operations-avancees-pandas

Opération avancées en **pandas**

7.25.1 Complément - niveau intermédiaire

Introduction

pandas supporte des opérations de manipulation des **Series** et **DataFrame** qui sont similaires dans l'esprit à ce que l'on peut faire avec une base de données et le langage SQL, mais de manière plus intuitive et expressive et beaucoup plus efficacement puisque les opérations se déroulent toutes en mémoire.

Vous pouvez concaténer (**concat**) des **DataFrame**, faire des jointures (**merge**), faire des regroupements (**groupby**) ou réorganiser les index (**pivot**).

Nous allons dans la suite développer ces différentes techniques.

```
[1]: import numpy as np
import pandas as pd
```

Concaténations avec **concat**

concat est utilisé pour concaténer des **Series** ou des **DataFrame**. Regardons un exemple.

```
[2]: s1 = pd.Series([30, 35], index=['alice', 'bob'])
s2 = pd.Series([32, 22, 29], index=['bill', 'alice', 'jo'])
```

```
[3]: s1
```

```
[3]: alice    30
bob        35
dtype: int64
```

```
[4]: s2
```

```
[4]: bill      32
alice      22
jo         29
dtype: int64
```

```
[5]: pd.concat([s1, s2])
```

```
[5]: alice    30
      bob      35
      bill     32
      alice    22
      jo       29
      dtype: int64
```

On remarque, cependant, que par défaut il n'y a pas de contrôle sur les labels d'index dupliqués. On peut corriger cela avec l'argument `verify_integrity`, qui va produire une exception s'il y a des labels d'index communs. Évidemment, cela a un coût de calcul supplémentaire, ça n'est donc à utiliser que si c'est nécessaire.

```
[6]: try:
      pd.concat([s1, s2], verify_integrity=True)
    except ValueError as e:
      print(f"erreur de concaténation:\n{e}")
```

```
erreur de concaténation:
Indexes have overlapping values: Index(['alice'], dtype='object')
```

```
[7]: # créons deux Series avec les index sans recouvrement
s1 = pd.Series(range(1000), index=[chr(x) for x in range(1000)])
s2 = pd.Series(range(1000), index=[chr(x+2000) for x in range(1000)])
```

```
[8]: # temps de concaténation avec vérification des recouvrements
%timeit pd.concat([s1, s2], verify_integrity=True)
```

351 μ s \pm 13.2 μ s per loop (mean \pm std. dev. of 7 runs, 1000 loops each)

```
[9]: # temps de concaténation sans vérification des recouvrements
%timeit pd.concat([s1, s2])
```

184 μ s \pm 11.1 μ s per loop (mean \pm std. dev. of 7 runs, 1000 loops each)

Par défaut, `concat` concatène les lignes, c'est-à-dire que `s2` sera sous `s1`, mais on peut changer ce comportement en utilisant l'argument `axis` :

```
[10]: p1 = pd.DataFrame(np.random.randint(1, 10, size=(2,2)),
                        columns=list('ab'), index=list('xy'))
p2 = pd.DataFrame(np.random.randint(1, 10, size=(2,2)),
                  columns=list('ab'), index=list('zt'))
```

```
[11]: p1
```

```
[11]:   a  b
x   8  5
y   2  1
```

```
[12]: p2
```

```
[12]:   a  b
z   8  1
t   8  4
```

```
[13]: # équivalent à pd.concat([p1, p2], axis=0)
      # concaténation des lignes
      pd.concat([p1, p2])
```

```
[13]:   a  b
      x  8  5
      y  2  1
      z  8  1
      t  8  4
```

```
[14]: p1 = pd.DataFrame(np.random.randint(1, 10, size=(2,2)),
                        columns=list('ab'), index=list('xy'))
      p2 = pd.DataFrame(np.random.randint(1, 10, size=(2,2)),
                        columns=list('cd'), index=list('xy'))
```

```
[15]: p1
```

```
[15]:   a  b
      x  3  5
      y  6  8
```

```
[16]: p2
```

```
[16]:   c  d
      x  3  9
      y  9  8
```

```
[17]: # concaténation des colonnes
      pd.concat([p1, p2], axis=1)
```

```
[17]:   a  b  c  d
      x  3  5  3  9
      y  6  8  9  8
```

Regardons maintenant ce cas :

```
[18]: pd.concat([p1, p2])
```

```
/Users/tparment/miniconda3/envs/flotpython-course/lib/python3.7/site-packag
es/ipykernel_launcher.py:1: FutureWarning: Sorting because non-concatena
tion axis is not aligned. A future version
of pandas will change to not sort by default.
```

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=True'.

```
"""Entry point for launching an IPython kernel.
```

```
[18]:   a  b  c  d
      x  3.0  5.0  NaN  NaN
      y  6.0  8.0  NaN  NaN
      x  NaN  NaN  3.0  9.0
```

```
y NaN NaN 9.0 8.0
```

Vous remarquez que lors de la concaténation, on prend l'union des tous les labels des index de `p1` et `p2`, il y a donc des valeurs absentes qui sont mises à `NaN`. On peut contrôler ce comportement de plusieurs manières comme nous allons le voir ci-dessous.

Par défaut (ce que l'on a fait ci-dessus), `join` utilise la stratégie dite **outer**, c'est-à-dire qu'on prend l'union des labels.

```
[19]: # on concatène les lignes, l'argument join décide quels labels on garde
      # sur l'autre axe (ici sur les colonnes).

      # si on spécifie 'inner' on prend l'intersection des labels
      # du coup il ne reste rien ..
      pd.concat([p1, p2], join='inner')
```

```
[19]: Empty DataFrame
      Columns: []
      Index: [x, y, x, y]
```

Avec `reindex`, on peut spécifier les labels qu'on veut garder dans l'index des lignes (`axis=0`, c'est la valeur par défaut) ou des colonnes (`axis=1`) :

```
[20]: # on peut passer à reindex une liste de labels...
      pd.concat([p1, p2], axis=1).reindex(['x'])
```

```
[20]:   a  b  c  d
      x  3  5  3  9
```

```
[21]: # ou un objet Index
      # Pour les colonnes je spécifie un reindex avec axis=1
      pd.concat([p1, p2], axis=1).reindex(p2.columns, axis=1)
```

```
[21]:   c  d
      x  3  9
      y  9  8
```

```
[22]: pd.concat([p1, p2], axis=1).reindex(['a', 'b'], axis=1)
```

```
[22]:   a  b
      x  3  5
      y  6  8
```

Notons que les `Series` et `DataFrame` ont une méthode `append` qui est un raccourci vers `concat`, mais avec moins d'options.

Pour aller plus loin, voici la documentation officielle :

<http://pandas.pydata.org/pandas-docs/stable/merging.html#concatenating-objects>

Jointures avec `merge`

`merge` est dans l'esprit similaire au `JOIN` en `SQL`. L'idée est de combiner deux `DataFrame` en fonction d'un critère d'égalité sur des colonnes. Regardons un exemple :

```
[23]: df1 = pd.DataFrame({'employee': ['Bob', 'Lisa', 'Sue'],
                        'group': ['Accounting', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Sue'],
                    'hire_date': [2004, 2008, 2014]})
```

```
[24]: df1
```

```
[24]:   employee      group
0      Bob  Accounting
1     Lisa  Engineering
2      Sue           HR
```

```
[25]: df2
```

```
[25]:   employee  hire_date
0     Lisa      2004
1      Bob      2008
2      Sue      2014
```

On souhaite ici combiner `df1` et `df2` de manière à ce que les lignes contenant le même employee soient alignées. Notre critère de merge est donc l'égalité des labels sur la colonne `employee`.

```
[26]: pd.merge(df1, df2)
```

```
[26]:   employee      group  hire_date
0      Bob  Accounting      2008
1     Lisa  Engineering      2004
2      Sue           HR      2014
```

Par défaut, `merge` fait un inner join (ou jointure interne) en utilisant comme critère de jointure les colonnes de même nom (ici `employee`). inner join veut dire que pour joindre deux lignes il faut que le même `employee` apparaisse dans les deux `DataFrame`.

Il existe trois type de merges :

- one-to-one, c'est celui que l'on vient de voir. C'est le merge lorsqu'il n'y a pas de labels dupliqués dans les colonnes utilisées comme critère de merge ;
- many-to-one, c'est le merge lorsque l'une des deux colonnes contient des labels dupliqués, dans ce cas, on applique la stratégie one-to-one pour chaque label dupliqué, donc les entrées dupliquées sont préservées ;
- many-to-many, c'est la stratégie lorsqu'il y a des entrées dupliquées dans les deux colonnes. Dans ce cas, on fait un produit cartésien des lignes.

D'une manière générale, gardez en tête que `pandas` fait essentiellement ce à quoi on s'attend. Regardons cela sur des exemples :

```
[27]: df1 = pd.DataFrame({'patient': ['Bob', 'Lisa', 'Sue'],
                        'repas': ['SS', 'SS', 'SSR']})
df2 = pd.DataFrame({'repas': ['SS', 'SSR'],
                    'explication': ['sans sel', 'sans sucre']})
```

```
[28]: df1
```



```
[28]: patient repas
0      Bob      SS
1      Lisa      SS
2      Sue      SSR
```

```
[29]: df2
```

```
[29]: repas explication
0      SS      sans sel
1      SSR     sans sucre
```

```
[30]: # la colonne commune pour le merge est 'repas' et dans une des colonnes
# (sur df1), il y a des labels dupliqués, on applique la stratégie many-to-one
pd.merge(df1, df2)
```

```
[30]: patient repas explication
0      Bob      SS      sans sel
1      Lisa      SS      sans sel
2      Sue      SSR     sans sucre
```

```
[31]: df1 = pd.DataFrame({'patient': ['Bob', 'Lisa', 'Sue'],
                          'repas': ['SS', 'SS', 'SSR']})
df2 = pd.DataFrame({'repas': ['SS', 'SS', 'SSR'],
                    'explication': ['sans sel', 'légumes', 'sans sucre']})
```

```
[32]: df1
```

```
[32]: patient repas
0      Bob      SS
1      Lisa      SS
2      Sue      SSR
```

```
[33]: df2
```

```
[33]: repas explication
0      SS      sans sel
1      SS      légumes
2      SSR     sans sucre
```

```
[34]: # la colonne commune pour le merge est 'repas' et dans les deux colonnes
# il y a des labels dupliqués, on applique la stratégie many-to-many
pd.merge(df1, df2)
```

```
[34]: patient repas explication
0      Bob      SS      sans sel
1      Bob      SS      légumes
2      Lisa      SS      sans sel
3      Lisa      SS      légumes
4      Sue      SSR     sans sucre
```

Dans un merge, on peut contrôler les colonnes à utiliser comme critère de merge. Regardons ces différents cas sur des exemples :

```
[35]: df1 = pd.DataFrame({'employee': ['Bob', 'Lisa', 'Sue'],
                        'group': ['Accounting', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Sue'],
                    'hire_date': [2004, 2008, 2014]})
```

```
[36]: df1
```

```
[36]:   employee      group
0      Bob  Accounting
1     Lisa  Engineering
2      Sue           HR
```

```
[37]: df2
```

```
[37]:   employee  hire_date
0     Lisa      2004
1      Bob      2008
2      Sue      2014
```

```
[38]: # on décide d'utiliser la colonne 'employee' comme critère de merge
pd.merge(df1, df2, on='employee')
```

```
[38]:   employee      group  hire_date
0      Bob  Accounting      2008
1     Lisa  Engineering      2004
2      Sue           HR      2014
```

```
[39]: df1 = pd.DataFrame({'employee': ['Bob', 'Lisa', 'Sue'],
                        'group': ['Accounting', 'Engineering', 'HR']})
df2 = pd.DataFrame({'name': ['Lisa', 'Bob', 'Sue'],
                    'hire_date': [2004, 2008, 2014]})
```

```
[40]: df1
```

```
[40]:   employee      group
0      Bob  Accounting
1     Lisa  Engineering
2      Sue           HR
```

```
[41]: df2
```

```
[41]:   name  hire_date
0  Lisa      2004
1   Bob      2008
2   Sue      2014
```

```
[42]: # mais on peut également définir un nom de colonne différent
# à gauche et à droite
m = pd.merge(df1, df2, left_on='employee', right_on='name')
m
```

```
[42]: employee      group  name  hire_date
      0      Bob    Accounting  Bob      2008
      1      Lisa  Engineering  Lisa      2004
      2      Sue          HR      Sue      2014
```

```
[43]: # dans ce cas, comme on garde les colonnes utilisées comme critère dans
      # le résultat du merge, on peut effacer la colonne inutile ainsi
      m.drop('name', axis=1)
```

```
[43]: employee      group  hire_date
      0      Bob    Accounting      2008
      1      Lisa  Engineering      2004
      2      Sue          HR          2014
```

`merge` permet également de contrôler la stratégie à appliquer lorsqu'il y a des valeurs dans une colonne utilisée comme critère de merge qui sont absentes dans l'autre colonne. C'est ce que l'on appelle jointure à gauche, jointure à droite, jointure interne (comportement par défaut) et jointure externe. Pour ceux qui ne sont pas familiers avec ces notions, regardons des exemples :

```
[44]: df1 = pd.DataFrame({'name': ['Bob', 'Lisa', 'Sue'],
                        'pulse': [70, 63, 81]})
      df2 = pd.DataFrame({'name': ['Eric', 'Bob', 'Marc'],
                        'weight': [60, 100, 70]})
```

```
[45]: df1
```

```
[45]:   name  pulse
      0   Bob     70
      1  Lisa     63
      2   Sue     81
```

```
[46]: df2
```

```
[46]:   name  weight
      0  Eric      60
      1   Bob     100
      2  Marc      70
```

```
[47]: # la colonne 'name' est le critère de merge dans les deux DataFrame.
      # Seul Bob existe dans les deux colonnes. Dans un inner join
      # (le cas par défaut) on ne garde que les lignes pour lesquelles il y a une
      # même valeur présente à gauche et à droite
      pd.merge(df1, df2) # équivalent à pd.merge(df1, df2, how='inner')
```

```
[47]:   name  pulse  weight
      0   Bob     70     100
```

```
[48]: # le outer join va au contraire faire une union des lignes et compléter ce
      # qui manque avec NaN
      pd.merge(df1, df2, how='outer')
```

```
[48]:   name  pulse  weight
      0   Bob   70.0   100.0
```

```
1 Lisa 63.0 NaN
2 Sue 81.0 NaN
3 Eric NaN 60.0
4 Marc NaN 70.0
```

```
[49]: # le left join ne garde que les valeurs de la colonne de gauche
pd.merge(df1, df2, how='left')
```

```
[49]:   name  pulse  weight
0   Bob     70   100.0
1  Lisa     63    NaN
2   Sue     81    NaN
```

```
[50]: # et le right join ne garde que les valeurs de la colonne de droite
pd.merge(df1, df2, how='right')
```

```
[50]:   name  pulse  weight
0   Bob   70.0    100
1  Eric   NaN     60
2  Marc   NaN     70
```

Pour aller plus loin, vous pouvez lire la documentation. Vous verrez notamment que vous pouvez merger sur les index (au lieu des colonnes) ou le cas où vous avez des colonnes de même nom qui ne font pas partie du critère de merge :

<http://pandas.pydata.org/pandas-docs/stable/merging.html#database-style-dataframe-joining-merging>

Regroupement avec **groupby**

Regardons maintenant cette notion de groupement. Il s'agit d'une notion très puissante avec de nombreuses options que nous ne couvrirons que partiellement. La logique derrière **groupby** est de créer des groupes dans une **DataFrame** en fonction des valeurs d'une (ou plusieurs) colonne(s), toutes les lignes contenant la même valeur sont dans le même groupe. On peut ensuite appliquer à chaque groupe des opérations qui sont :

- soit des calculs sur chaque groupe ;
- soit un filtre sur chaque groupe qui peut garder ou supprimer un groupe ;
- soit une transformation qui va modifier tout le groupe (par exemple, pour centrer les valeurs sur la moyenne du groupe).

Regardons quelques exemples :

```
[51]: d = pd.DataFrame({'key': list('ABCABC'), 'val': range(6)})
d
```

```
[51]:   key  val
0    A    0
1    B    1
2    C    2
3    A    3
4    B    4
5    C    5
```

```
[52]: # utilisons comme colonne de groupement 'key'
g = d.groupby('key')
g
```

```
[52]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x11487def0>
```

`groupby` produit un nouvel objet, mais ne fait aucun calcul. Les calculs seront effectués lors de l'appel d'une fonction sur ce nouvel objet. Par exemple, calculons la somme pour chaque groupe.

```
[53]: g.sum()
```

```
[53]:      val
key
A      3
B      5
C      7
```

`groupby` peut utiliser comme critère de groupement une colonne, une liste de colonnes, ou un index (c'est notamment utile pour les `Series`).

Une particularité de `groupby` est que le critère de groupement devient un index dans le nouvel objet généré. L'avantage est que l'on a maintenant un accès optimisé sur ce critère, mais l'inconvénient est que sur certaines opérations qui détruisent l'index on peut perdre ce critère. On peut contrôler ce comportement avec `as_index`.

```
[54]: g = d.groupby('key', as_index=False)
g.sum()
```

```
[54]:  key  val
0    A    3
1    B    5
2    C    7
```

L'objet produit par `groupby` permet de manipuler les groupes, regardons cela :

```
[55]: d = pd.DataFrame({'key': list('ABCABC'),
                       'val1': range(6),
                       'val2': range(100, 106)})
d
```

```
[55]:  key  val1  val2
0    A     0   100
1    B     1   101
2    C     2   102
3    A     3   103
4    B     4   104
5    C     5   105
```

```
[56]: g = d.groupby('key')

# g.groups donne accès au dictionnaire des groupes,
# les clefs sont le nom du groupe
# et les valeurs les index des lignes
# appartenant au groupe
```

```
g.groups
```

```
[56]: {'A': Int64Index([0, 3], dtype='int64'),
      'B': Int64Index([1, 4], dtype='int64'),
      'C': Int64Index([2, 5], dtype='int64')}
```

```
[57]: # pour accéder directement au groupe, on peut utiliser get_group
      g.get_group('A')
```

```
[57]:   key  val1  val2
      0   A     0   100
      3   A     3  103
```

```
[58]: # on peut également filtrer un groupe par colonne
      # lors d'une opération
      g.sum()['val2']
```

```
[58]: key
      A    203
      B    205
      C    207
      Name: val2, dtype: int64
```

```
[59]: # ou directement sur l'objet produit par groupby
      g['val2'].sum()
```

```
[59]: key
      A    203
      B    205
      C    207
      Name: val2, dtype: int64
```

On peut également itérer sur les groupes avec un boucle `for` classique :

```
[60]: import seaborn as sns
      # on charge le fichier de données des pourboires
      tips = sns.load_dataset('tips')

      # pour rappel
      tips.head()
```

```
[60]:   total_bill  tip  sex smoker  day  time  size
      0     16.99  1.01 Female    No  Sun  Dinner     2
      1     10.34  1.66   Male    No  Sun  Dinner     3
      2     21.01  3.50   Male    No  Sun  Dinner     3
      3     23.68  3.31   Male    No  Sun  Dinner     2
      4     24.59  3.61 Female    No  Sun  Dinner     4
```

```
[61]: # on groupe le DataFrame par jours
      g = tips.groupby('day')

      # on calcule la moyenne du pourboire par jour
      for (group, index) in g:
```

```
print(f"On {group} the mean tip is {index['tip'].mean():.3}")
```

```
On Thur the mean tip is 2.77
On Fri the mean tip is 2.73
On Sat the mean tip is 2.99
On Sun the mean tip is 3.26
```

L'objet produit par `groupby` supporte ce que l'on appelle le dispatch de méthodes. Si une méthode n'est pas directement définie sur l'objet produit par `groupby`, elle est appelée sur chaque groupe (il faut donc qu'elle soit définie sur les `DataFrame` ou les `Series`). Regardons cela :

```
[62]: # on groupe par jour et on extrait uniquement la colonne 'total_bill'
# pour chaque groupe
g = tips.groupby('day')['total_bill']

# on demande à pandas d'afficher les float avec seulement deux chiffres
# après la virgule
pd.set_option('display.float_format', '{:.2f}'.format)

# on appelle describe() sur g, mais elle n'est pas définie sur cet objet,
# elle va donc être appelée (dispatch) sur chaque groupe
g.describe()
```

```
[62]:      count  mean  std  min   25%   50%   75%  max
day
Thur   62.00  17.68  7.89  7.51  12.44  16.20  20.16  43.11
Fri    19.00  17.15  8.30  5.75  12.09  15.38  21.75  40.17
Sat    87.00  20.44  9.48  3.07  13.91  18.24  24.74  50.81
Sun    76.00  21.41  8.83  7.25  14.99  19.63  25.60  48.17
```

```
[63]: # Mais, il y a tout de même un grand nombre de méthodes
# définies directement sur l'objet produit par le groupby

methods = [x for x in dir(g) if not x.startswith('_')]
f"Le type {type(g).__name__} expose {len(methods)} méthodes."
```

```
[63]: 'Le type SeriesGroupBy expose 68 méthodes.'
```

```
[64]: # profitons de la mise en page des dataframes
# pour afficher ces méthodes sur plusieurs colonnes
# on fait un peu de gymnastique
# il y a d'ailleurs sûrement plus simple..
columns = 7
nb_methods = len(methods)
nb_pad = (columns - nb_methods % columns) % columns

array = np.array(methods + nb_pad * ['']).reshape((columns, -1))
```

```
[65]: pd.DataFrame(data=array.transpose())
```

```
[65]:      0      1      2      3      4  \
0    agg  cumcount  fillna  is_monotonic_decreasing  ngroups
1  aggregate  cummax  filter  is_monotonic_increasing  nlargest
2     all  cummin  first  last  nsmallest
3     any  cumprod  get_group  mad  nth
```

4	apply	cumsum	groups	max	nunique
5	backfill	describe	head	mean	ohlc
6	bfill	diff	hist	median	pad
7	corr	dtype	idxmax	min	pct_change
8	count	expanding	idxmin	ndim	pipe
9	cov	ffill	indices	ngroup	plot

	5	6
0	prod	sum
1	quantile	tail
2	rank	take
3	resample	transform
4	rolling	tshift
5	sem	unique
6	shift	value_counts
7	size	var
8	skew	
9	std	

Nous allons regarder la méthode `aggregate` (dont l'alias est `agg`). Cette méthode permet d'appliquer une fonction (ou liste de fonctions) à chaque groupe avec la possibilité d'appliquer une fonction à une colonne spécifique du groupe.

Une subtilité de `aggregate` est que l'on peut passer soit un objet fonction, soit un nom de fonction sous forme d'une `str`. Pour que l'utilisation du nom de la fonction marche, il faut que la fonction soit définie sur l'objet produit par le `groupby` ou qu'elle soit définie sur les groupes (donc avec `dispatching`).

```
[66]: # calculons la moyenne et la variance pour chaque groupe
      # et chaque colonne numérique
      tips.groupby('day').agg(['mean', 'std'])
```

```
[66]:      total_bill      tip      size
      mean std mean std mean std
day
Thur      17.68 7.89 2.77 1.24 2.45 1.07
Fri       17.15 8.30 2.73 1.02 2.11 0.57
Sat       20.44 9.48 2.99 1.63 2.52 0.82
Sun       21.41 8.83 3.26 1.23 2.84 1.01
```

```
[67]: # de manière équivalente avec les objets fonctions
      tips.groupby('day').agg([np.mean, np.std])
```

```
[67]:      total_bill      tip      size
      mean std mean std mean std
day
Thur      17.68 7.89 2.77 1.24 2.45 1.07
Fri       17.15 8.30 2.73 1.02 2.11 0.57
Sat       20.44 9.48 2.99 1.63 2.52 0.82
Sun       21.41 8.83 3.26 1.23 2.84 1.01
```

```
[68]: # en appliquant une fonction différente pour chaque colonne,
      # on passe alors un dictionnaire qui a pour clef le nom de la
      # colonne et pour valeur la fonction à appliquer à cette colonne
      tips.groupby('day').agg({'tip': np.mean, 'total_bill': np.std})
```



```
[68]:      tip  total_bill
      day
Thur  2.77          7.89
Fri   2.73          8.30
Sat   2.99          9.48
Sun   3.26          8.83
```

La méthode `filter` a pour but de filtrer les groupes en fonction d'un critère. Mais attention, `filter` retourne un sous ensemble des données originales dans lesquelles les éléments appartenant aux groupes filtrés ont été enlevés.

```
[69]: d = pd.DataFrame({'key': list('ABCABC'),
                        'val1': range(6),
                        'val2' : range(100, 106)})
d
```

```
[69]:   key  val1  val2
0    A     0    100
1    B     1    101
2    C     2    102
3    A     3    103
4    B     4    104
5    C     5    105
```

```
[70]: # regardons la somme par groupe
d.groupby('key').sum()
```

```
[70]:      val1  val2
key
A         3    203
B         5    205
C         7    207
```

```
[71]: # maintenant gardons dans les données originales toutes les lignes
# pour lesquelles la somme de leur groupe est supérieure à 3
# (ici les groupes B et C)
d.groupby('key').filter(lambda x: x['val1'].sum() > 3)
```

```
[71]:   key  val1  val2
1    B     1    101
2    C     2    102
4    B     4    104
5    C     5    105
```

La méthode `transform` a pour but de retourner un sous ensemble des données originales dans lesquelles une fonction a été appliquée par groupe. Un usage classique est de centrer des valeurs par groupe, ou de remplacer les NaN d'un groupe par la valeur moyenne du groupe.

Attention, `transform` ne doit pas faire de modifications en place, sinon le résultat peut être faux. Faites donc bien attention de ne pas appliquer des fonctions qui font des modifications en place.

```
[72]: r = np.random.normal(0.5, 2, 4)
d = pd.DataFrame({'key': list('ab'*2), 'data': r, 'data2': r*2})
d
```

```
[72]:   key  data  data2
      0   a  0.15   0.31
      1   b  0.54   1.08
      2   a  1.80   3.60
      3   b  4.46   8.92
```

```
[73]: # je groupe sur la colonne 'key'
      g = d.groupby('key')
```

```
[74]: # maintenant je centre chaque groupe par rapport à sa moyenne
      g.transform(lambda x: x - x.mean())
```

```
[74]:   data  data2
      0 -0.82 -1.65
      1 -1.96 -3.92
      2  0.82  1.65
      3  1.96  3.92
```

Notez que la colonne **key** a disparu, ce comportement est expliqué ici :

<http://pandas.pydata.org/pandas-docs/stable/groupby.html#automatic-exclusion-of-nuisance-columns>

Pour aller plus loin sur **groupby** vous pouvez lire la documentation :

<http://pandas.pydata.org/pandas-docs/stable/groupby.html>

Réorganisation des indexes avec **pivot**

Une manière de voir la notion de pivot est de considérer qu'il s'agit d'une extension de **groupby** à deux dimensions. Pour illustrer cela, prenons un exemple en utilisant le jeu de données seaborn sur les passagers du Titanic.

```
[75]: titanic = sns.load_dataset('titanic')
```

```
[76]: # regardons le format de ce jeu de données
      titanic.head()
```

```
[76]:   survived  pclass    sex  age  sibsp  parch  fare embarked  class  wh
      o \
      0      0      3  male 22.00     1     0  7.25          S  Third  ma
      n
      1      1      1 female 38.00     1     0 71.28          C  First  woma
      n
      2      1      3 female 26.00     0     0  7.92          S  Third  woma
      n
      3      1      1 female 35.00     1     0 53.10          S  First  woma
      n
      4      0      3  male 35.00     0     0  8.05          S  Third  ma
      n

      adult_male deck  embark_town alive  alone
      0      True  NaN  Southampton    no  False
      1     False   C   Cherbourg   yes  False
      2     False  NaN  Southampton   yes   True
```

```
3      False    C Southampton  yes  False
4      True   NaN Southampton  no   True
```

```
[77]: # regardons maintenant le taux de survie par classe et par sex
titanic.pivot_table('survived', index='class', columns='sex')
```

```
[77]: sex      female  male
class
First      0.97  0.37
Second     0.92  0.16
Third      0.50  0.14
```

Je ne vais pas entrer plus dans le détail, mais vous voyez qu'il s'agit d'un outil très puissant.

Pour aller plus loin, vous pouvez regarder la documentation officielle :

<http://pandas.pydata.org/pandas-docs/stable/reshaping.html>

mais vous aurez des exemples beaucoup plus parlants en regardant ici :

<https://github.com/jakevdp/PythonDataScienceHandbook/blob/master/notebooks/03.09-Pivot-Tables.ipynb>

7.26 w7-s09-c1-TimeSeries

Séries temporelles en **pandas**

7.26.1 Complément - niveau intermédiaire

Parsing des dates et gestion des erreurs

Lorsqu'il y a des erreurs de parsing des dates, pandas offre la possibilité de lancer une exception, ou de produire un objet NaT pour Not a Time qui se manipule ensuite comme un NaN.

```
[1]: import pandas as pd
date = '100/06/2018' # cette date ne peut pas être parsée

try:
    pd.to_datetime(date) # comportement pas défaut qui lance une exception
except ValueError as e:
    print(e)
```

```
('Unknown string format:', '100/06/2018')
```

```
[2]: # retourne l'input en cas d'erreur
pd.to_datetime(date, errors='ignore')
```

```
[2]: '100/06/2018'
```

```
[3]: # retourne NaT en cas d'erreur
pd.to_datetime(date, errors='coerce')
```

[3]: NaT

```
[4]: # la dernière date n'est pas valide
d = pd.to_datetime(['jun 2018', '10/12/1980',
                   '25 january 2000', '100 june 1900'],
                   errors='coerce')
print(d)
```

```
DatetimeIndex(['2018-06-01', '1980-10-12', '2000-01-25', 'NaT'], dtype='datetime64[ns]', freq=None)
```

```
[5]: # on peut utiliser les méthodes pour les NaN directement sur un NaT
d.fillna(pd.to_datetime('10 june 1980'))
```

```
[5]: DatetimeIndex(['2018-06-01', '1980-10-12', '2000-01-25', '1980-06-10'], dtype='datetime64[ns]', freq=None)
```

Pour aller plus loin

Vous trouverez de nombreux exemples [dans la documentation officielle de pandas](#)

7.26.2 Conclusion

Ce notebook clôt notre survol de **numpy** et **pandas**. C'est un sujet vaste que nous avons déjà largement dégrossi. Pour aller plus loin vous avez évidemment la documentation officielle de **numpy** et **pandas** :

- [reference numpy](#)
- [reference pandas](#)

Mais vous avez aussi l'excellent livre de Jake VanderPlas “Python Data Science Handbook” qui est entièrement disponible sous forme de notebooks en ligne :

<https://github.com/jakevdp/PythonDataScienceHandbook>

Il s'agit d'un très beau travail (c'est rare) utilisant les dernières versions de Python, **pandas** and **numpy** (c'est encore plus rare), fait par un physicien qui fait de la data science et qui a contribué au développement de nombreux modules de data science en Python.

Je vous conseille par ailleurs, pour ceux qui sont à l'aise en anglais, [une série de 10 vidéos sur YouTube](#) publiées par le même Jake VanderPlas, où il étudie un jeu de données du début (chargement des données) à la fin (classification).

Pour finir, si vous voulez faire de la data science, il y a un livre incontournable : “An Introduction of Statistical Learning” de G. James, D. Witten, T. Hastie, R. Tibshirani. Ce livre utilise R, mais vous pouvez facilement l'appliquer en utilisant **pandas**.

Les auteurs mettent à disposition gratuitement le PDF du livre ici :

<http://www-bcf.usc.edu/~gareth/ISL/>

N'oubliez pas, si ces ressources vous sont utiles, d'acheter ces livres pour supporter ces auteurs. Les ressources de grande qualité sont rares, elles demandent un travail énorme à produire, elles doivent être encouragées et récompensées.

7.27 w7-s10-c1-matplotlib-2d

matplotlib - 2D

7.27.1 Complément - niveau basique

Plutôt que de récrire (encore) un tutorial sur `matplotlib`, je préfère utiliser les ressources disponibles en ligne en anglais :

- pour la dimension 2 : https://matplotlib.org/2.0.2/users/pyplot_tutorial.html ;
- pour la dimension 3 : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html.

Je vais essentiellement utiliser des extraits tels quels. N'hésitez pas à consulter ces documents originaux pour davantage de précisions.

```
[1]: # les imports habituels
import numpy as np
import matplotlib.pyplot as plt
```

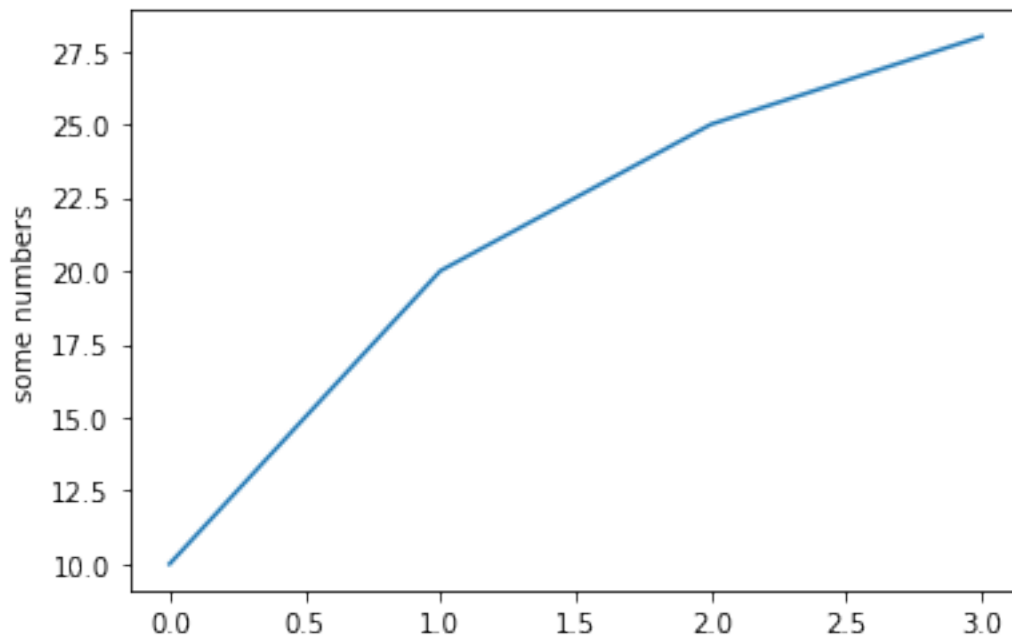
Intentionnellement dans ce notebook, on ne va pas utiliser le mode automatique de `matplotlib` dans les notebooks (pour rappel, `plt.ion()`), car on veut justement apprendre à utiliser `matplotlib` dans un contexte normal.

`plt.plot`

Nous avons déjà vu plusieurs fois comment tracer une courbe avec `matplotlib`, avec la fonction `plot`. Si on donne seulement une liste de valeurs, elles sont considérées comme les Y, les X étant les entiers en nombre suffisant et en commençant à 0.

```
[2]: # si je ne donne qu'une seule liste à plot
# alors ce sont les Y
plt.plot([10, 20, 25, 28])
# on peut aussi facilement ajouter une légende
# ici sur l'axe des y
plt.ylabel('some numbers')

plt.show()
```



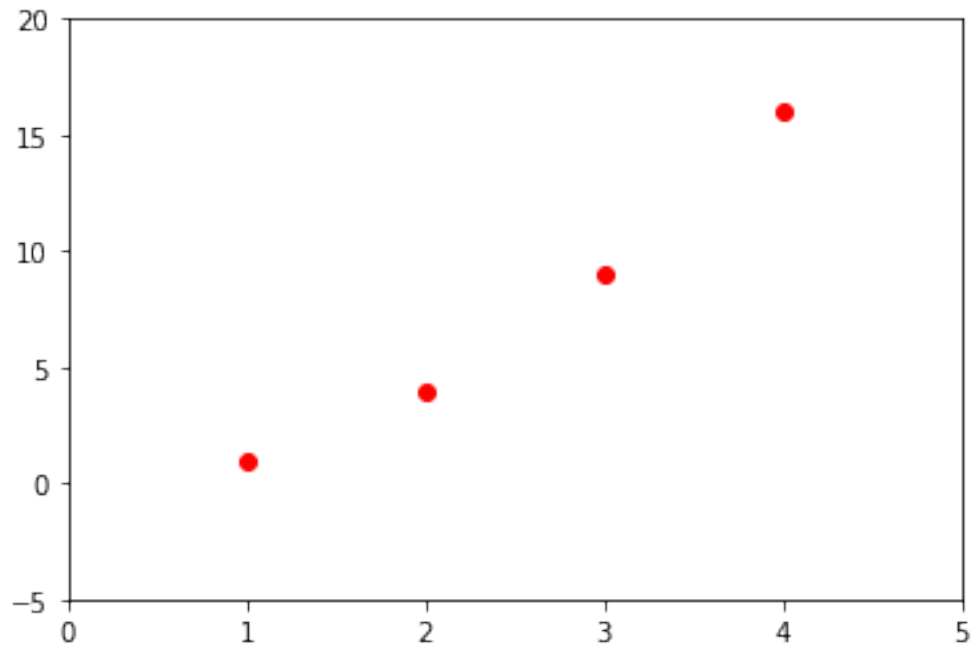
On peut changer le style utilisé par `plot` pour tracer ; ce style est spécifié sous la forme d'une chaîne de caractères, par défaut `'b-'`, qui signifie une ligne bleue (`b` pour bleu, et `-` pour ligne). Ici on va préciser à la place `ro`, `r` qui signifie rouge et `o` qui signifie cercle.

Voyez [la documentation de référence de plot](#) pour une liste complète.

```
[3]: # mais le plus souvent on passe à plot
# une liste de X ET une liste de Y
plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 16, 25], 'ro')

# ici on veut dire d'utiliser
# pour l'axe des X : entre 0 et 5
# pour l'axe des Y : entre -5 et 20
plt.axis([0, 5, -5, 20])

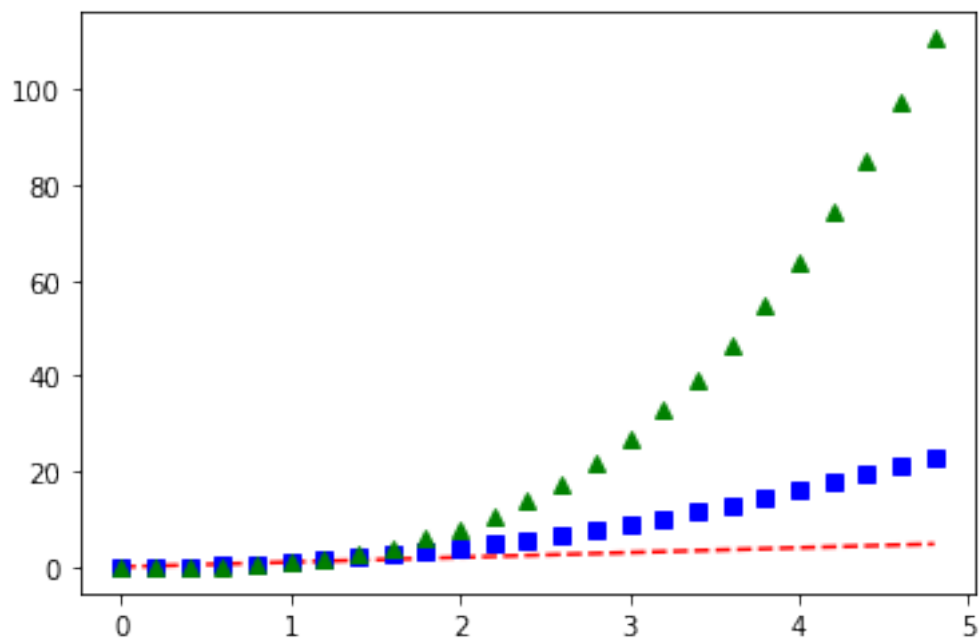
plt.show()
```



On peut très simplement dessiner plusieurs fonctions dans la même zone :

```
[4]: # échantillon de points entre 0 et 5 espacés de 0.2
t = np.arange(0., 5., 0.2)

# plusieurs styles de ligne
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
# on pourrait ajouter d'autres plot bien sûr aussi
plt.show()
```



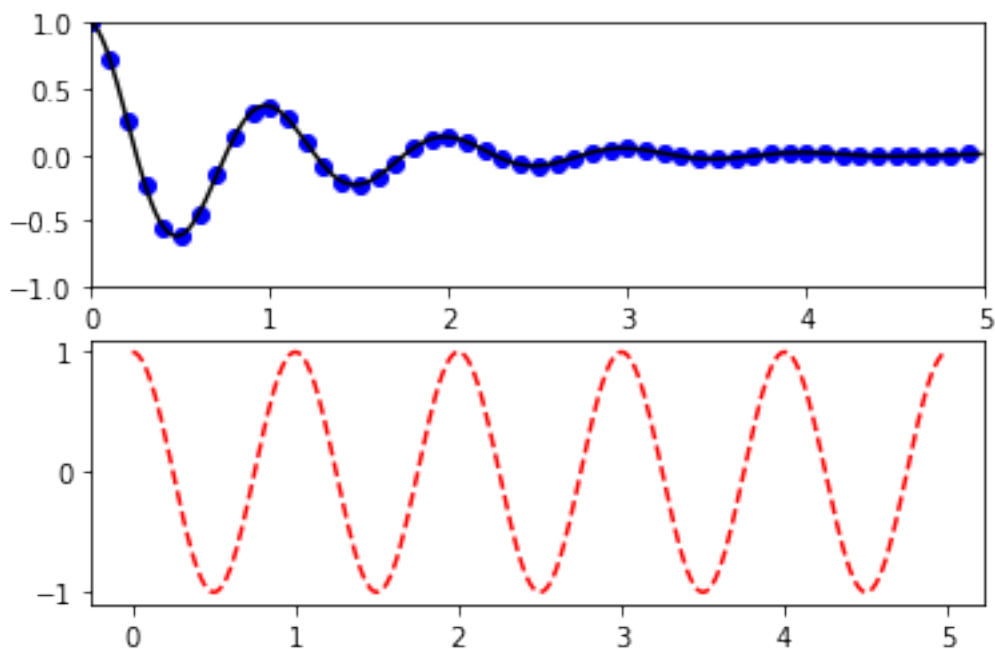
Plusieurs subplots

```
[5]: def f(t):
      return np.exp(-t) * np.cos(2*np.pi*t)

      ## deux domaines presque identiques
      # celui-ci pour les points bleus
      t1 = np.arange(0.0, 5.0, 0.1)
      # celui-ci pour la ligne bleue
      t2 = np.arange(0.0, 5.0, 0.02)

      # cet appel n'est pas nécessaire
      # vous pouvez vérifier qu'on pourrait l'enlever
      plt.figure(1)
      # on crée un 'subplot'
      plt.subplot(211)
      # le fonctionnement de matplotlib est dit 'stateful'
      # par défaut on dessine dans le dernier objet créé
      plt.axis([0, 5, -1, 1])
      plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

      # une deuxième subplot
      plt.subplot(212)
      # on écrit dedans
      plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
      plt.show()
```



C'est pour pouvoir construire de tels assemblages qu'il y a une fonction `plt.show()`, qui indique que la figure est terminée.

Il faut revenir un peu sur les arguments passés à `subplot`. Lorsqu'on écrit :

```
plt.subplot(211)
```

ce qui est par ailleurs juste un raccourci pour :

```
plt.subplot(2, 1, 1)
```

on veut dire qu'on veut créer un quadrillage de 2 lignes de 1 colonne, et que le subplot va occuper le 1er emplacement.

Plusieurs figures

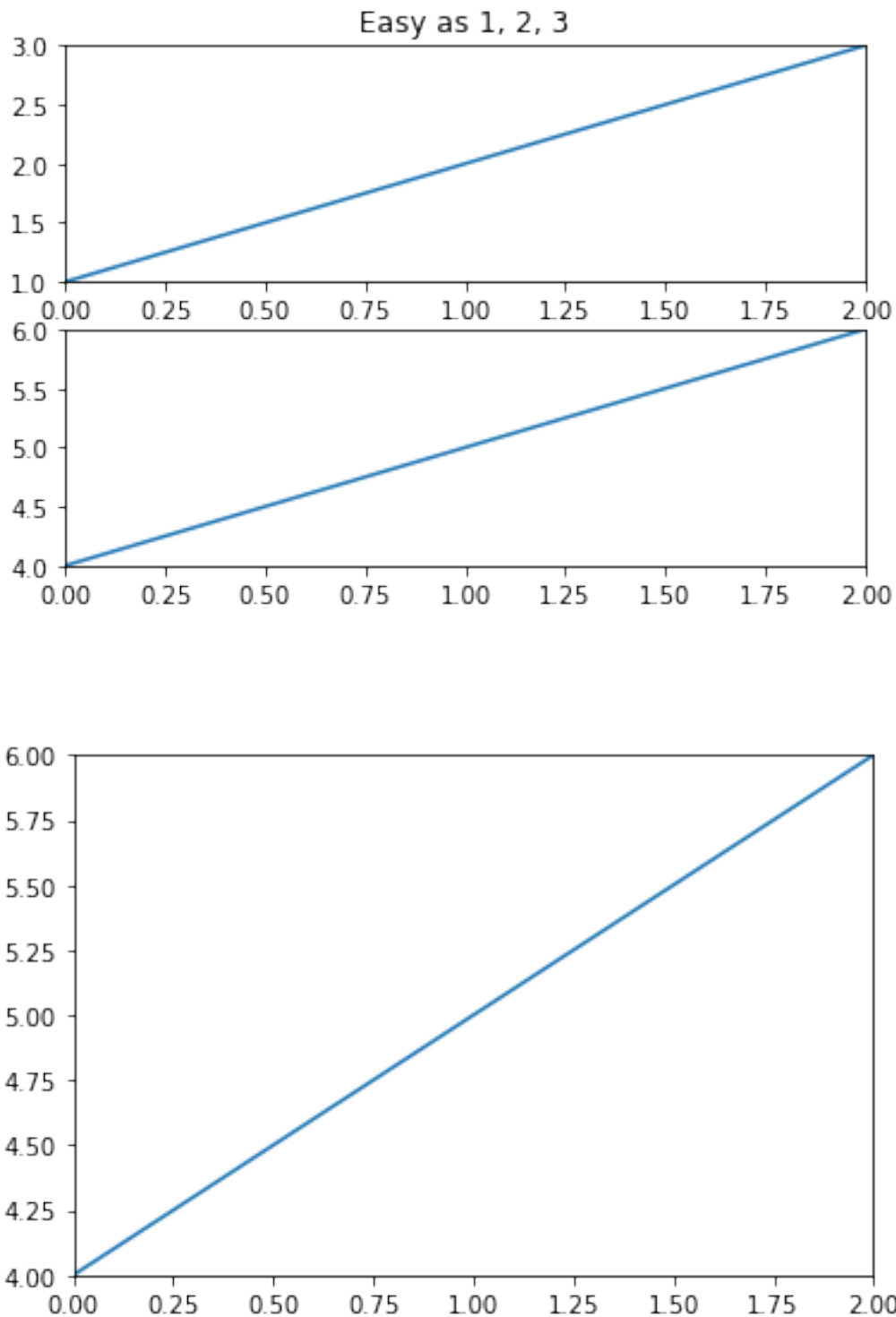
En fait, on peut créer plusieurs figures, et plusieurs subplots dans chaque figure. Dans l'exemple qui suit on illustre encore mieux cette notion de statefulness. Je commence par vous donner l'exemple du tutorial tel quel :

```
[6]: plt.figure(1)           # the first figure
      plt.subplot(211)       # the first subplot in the first figure
      plt.axis([0, 2, 1, 3])
      plt.plot([1, 2, 3])
      plt.subplot(212)       # the second subplot in the first figure
      plt.axis([0, 2, 4, 6])
      plt.plot([4, 5, 6])

      plt.figure(2)          # a second figure
      plt.axis([0, 2, 4, 6])
      plt.plot([4, 5, 6])    # creates a subplot(111) by default

      plt.figure(1)          # figure 1 current;
                              # subplot(212) still current
      plt.subplot(211)       # make subplot(211) in figure1 current
      plt.title('Easy as 1, 2, 3') # subplot 211 title
      plt.show()
```

```
/Users/tparment/miniconda3/envs/flotpython-course/lib/python3.7/site-packag
es/ipykernel_launcher.py:16: MatplotlibDeprecationWarning: Adding an axe
s using the same arguments as a previous axes currently reuses the earli
er instance. In a future version, a new instance will always be created
and returned. Meanwhile, this warning can be suppressed, and the futur
e behavior ensured, by passing a unique label to each axes instance.
app.launch_new_instance()
```



Cette façon de faire est améliorable. D'abord c'est source d'erreurs, il faut se souvenir de ce qui précède, et du coup, si on change un tout petit peu la logique, ça risque de casser tout le reste. En outre selon les environnements, on peut obtenir un vilain avertissement.

C'est pourquoi je vous conseille plutôt, pour faire la même chose que ci-dessus, d'utiliser `plt.subplots` qui vous retourne la figure avec ses subplots, que vous pouvez ranger dans des variables Python :

```
[7]: # c'est assez déroutant au départ, mais
# traditionnellement les subplots sont appelés 'axes'
# c'est pourquoi ici on utilise ax1, ax2 et ax3 pour désigner
# des subplots

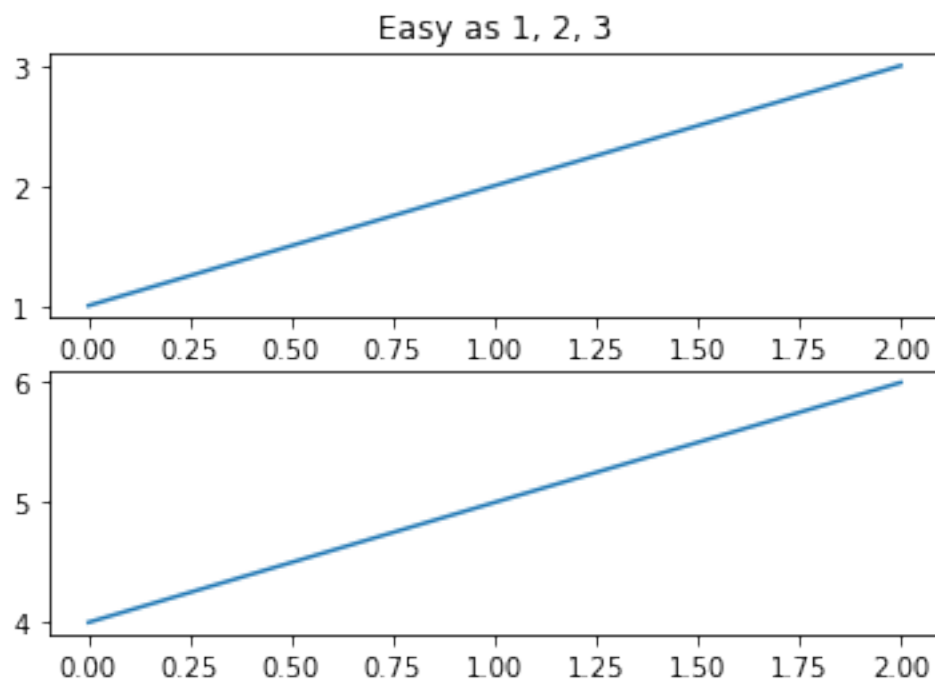
# ici je crée une figure et deux subplots,
# sur une grille de 2 lignes * 1 colonne
fig1, (ax1, ax2) = plt.subplots(2, 1)

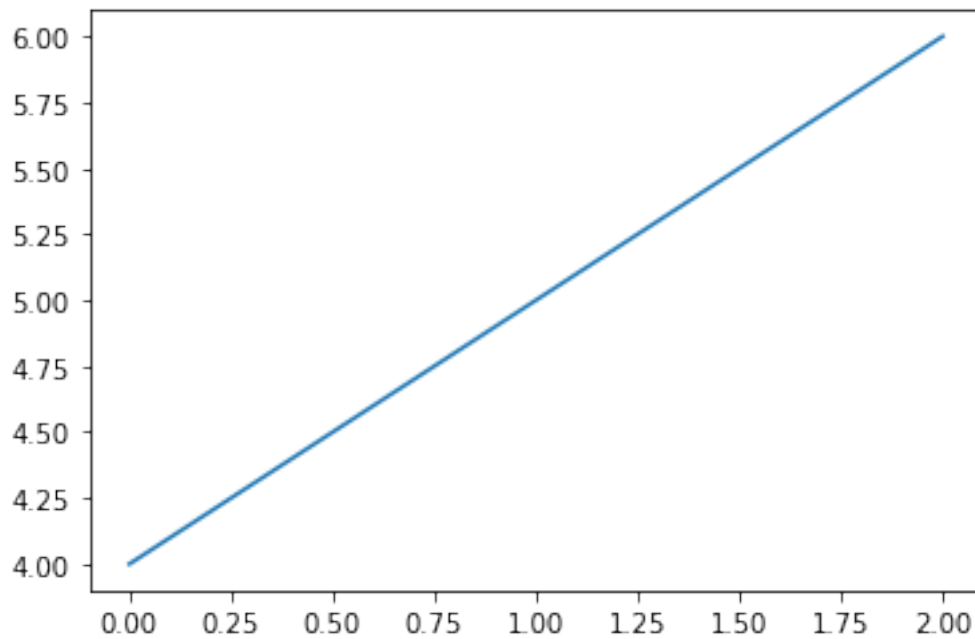
# au lieu de faire plt.plot, vous pouvez envoyer
# la méthode plot à un subplot
ax1.plot([1, 2, 3])
ax2.plot([4, 5, 6])

fig2, ax3 = plt.subplots(1, 1)
ax3.plot([4, 5, 6])

# pour revenir au premier subplot
# il suffit d'utiliser la variable ax1
# attention on avait fait avec 'plt.title'
# ici c'est la méthode 'set_title'
ax1.set_title('Easy as 1, 2, 3')

plt.show()
```





`plt.hist`

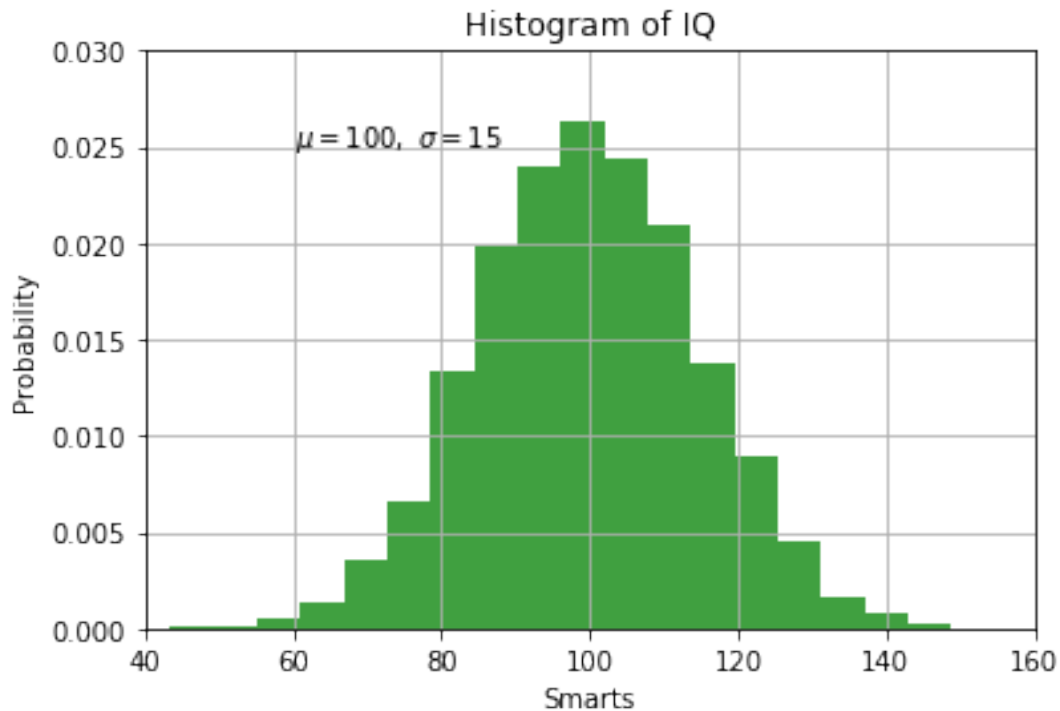
S'agissant de la dimension 2, voici le dernier exemple que nous tirons du tutoriel `matplotlib`, surtout pour illustrer `plt.hist`, mais qui montre aussi comment ajouter du texte :

```
[8]: # pour être reproductible, on fixe la graine
# du générateur aléatoire
np.random.seed(19680801)

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# dessiner un histogramme
# on range les valeurs en 20 boites (bins)
n, bins, patches = plt.hist(x, 20, density=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```



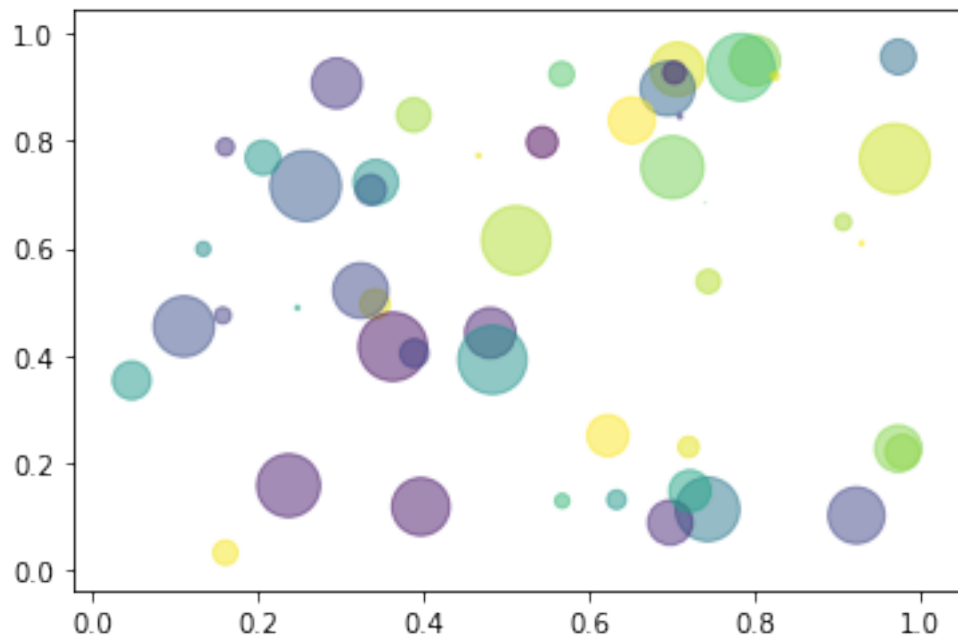
`plt.scatter`

Je vous recommande aussi de regarder également la fonction `plt.scatter` qui permet de faire par exemple des choses comme ceci :

```
[9]: # pour être reproductible, on fixe la graine
# du générateur aléatoire
np.random.seed(19680801)

N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2 # 0 to 15 point radii

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```



`plt.boxplot`

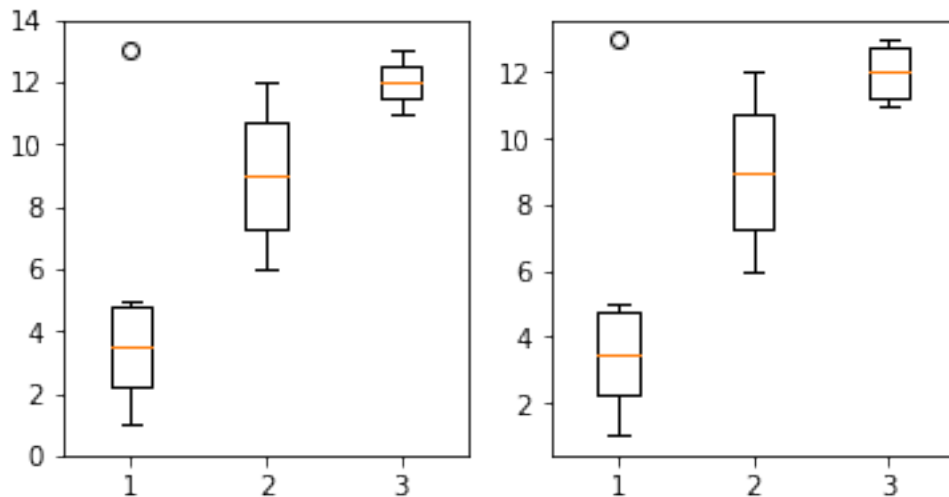
Avec `boxplot` vous obtenez des boîtes à moustache :

```
[10]: plt.figure(figsize=(6, 3))

plt.subplot(121)
# on peut passer à boxplot une liste de suites de nombres
# chaque suite donne lieu à une boîte à moustache
# ici 3 suites
plt.boxplot([[1, 2, 3, 4, 5, 13], [6, 7, 8, 10, 11, 12], [11, 12, 13]])
plt.ylim(0, 14)

plt.subplot(122)
# on peut aussi comme toujours lui passer un ndarray numpy
# attention c'est lu dans l'autre sens, ici aussi on a 3 suites de nombres
plt.boxplot(np.array([[1, 6, 11],
                      [2, 7, 12],
                      [3, 8, 13],
                      [4, 10, 11],
                      [5, 11, 12],
                      [13, 12, 13]]))

plt.show()
```



7.28 w7-s10-c2-matplotlib-3d

matplotlib 3D

Nous poursuivons notre introduction à `matplotlib` avec les visualisations en 3 dimensions. Comme pour la première partie sur les fonctions en 2 dimensions, nous allons seulement paraphraser [le tutoriel en ligne](#), avec l'avantage toutefois que nous procurent les notebooks.

```
[1]: # la ration habituelle d'imports
import matplotlib.pyplot as plt
# et aussi numpy, même si ça n'est pas strictement nécessaire
import numpy as np
```

Pour pouvoir faire des visualisations en 3D, il vous faut importer ceci :

```
[2]: # même si l'on n'utilise pas explicitement
# d'attributs du module Axes3D
# cet import est nécessaire pour faire
# des visualisations en 3D
from mpl_toolkits.mplot3d import Axes3D
```

Dans ce notebook nous allons utiliser un mode de visualisation un peu plus élaboré, mieux intégré à l'environnement des notebooks :

```
[3]: # ce mode d'interaction va nous permettre de nous déplacer
# dans l'espace pour voir les courbes en 3D
# depuis plusieurs points de vue
%matplotlib notebook
```

Comme on va le voir très vite, avec ces réglages vous aurez la possibilité d'explorer interactivement les visualisations en 3D.

Un premier exemple : une courbe

Commençons par le premier exemple du tutorial, qui nous montre comment dessiner une ligne suivant une courbe définie de manière paramétrique (ici, x et y sont fonctions de z). Les points importants sont :

- la composition d'un plot (plusieurs figures, chacune composée de plusieurs subplots), reste bien entendu valide ; j'ai enrichi l'exemple initial pour mélanger un subplot en 3D avec un subplot en 2D ;
- l'utilisation du paramètre `projection='3d'` lorsqu'on crée un subplot qui va se prêter à une visualisation en 3D ;
- l'objet subplot ainsi créé est une instance de la classe `Axes3DSubplot` ;
- on peut envoyer à cet objet :
 - la méthode `plot` qu'on avait déjà vue pour la dimension 2 (c'est ce que l'on fait dans ce premier exemple) ;
 - des méthodes spécifiques à la 3D, que l'on voit dans les exemples suivants.

```
[4]: # je choisis une taille raisonnable compte tenu de l'espace
# disponible dans fun-mooc
fig = plt.figure(figsize=(6, 3))

# voici la façon de créer un *subplot*
# qui se prête à une visualisation en 3D
ax = fig.add_subplot(121, projection='3d')

# à présent, copié de
# https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#line-plots
# on crée une courbe paramétrique
# où x et y sont fonctions de z
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
# on fait maintenant un appel à plot normal
# mais avec un troisième paramètre
ax.plot(x, y, z, label='parametric curve')
ax.legend()

# on peut tout à fait ajouter un plot usuel
# dans un subplot, comme on l'a vu pour la 2D
ax2 = fig.add_subplot(122)
x = np.linspace(0, 10)
y = x**2
ax2.plot(x, y)
plt.show()
```

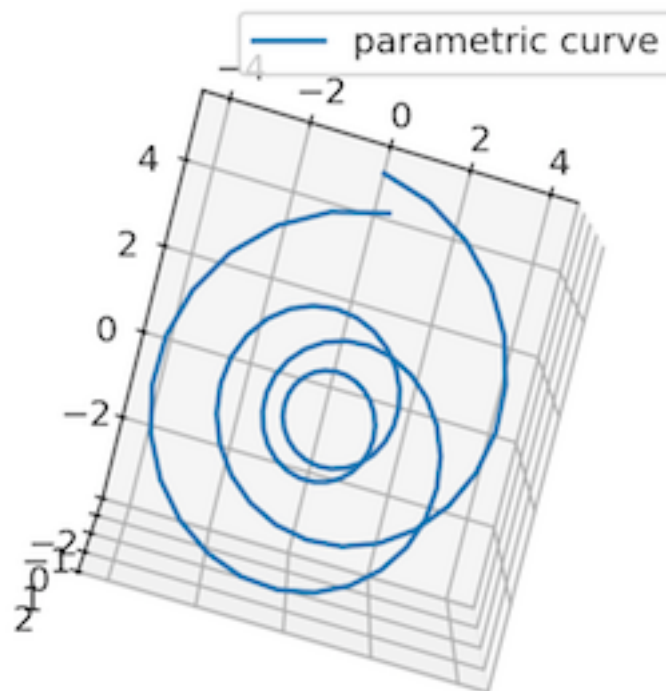
<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Un autre point à remarquer est qu'avec le mode d'interaction que nous avons choisi :

```
%matplotlib notebook
```

vous bénéficiez d'un mode d'interaction plus riche avec la figure. Par exemple, vous pouvez cliquer dans la figure en 3D, et vous déplacer pour changer de point de vue ; par exemple si vous sélectionnez l'outil **Pan/Zoom** (l'outil avec 4 flèches), vous pouvez arriver à voir ceci :



Les différents boutons d'outil sont [décrits plus en détail ici](#). Je dois avouer ne pas arriver à tout utiliser lorsque la visualisation est faite dans un notebook, mais la possibilité de modifier le point de vue peut s'avérer intéressante pour explorer les données.

En explorant les autres exemples du tutorial, vous pouvez commencer à découvrir l'éventail des possibilités offertes par `matplotlib`.

`Axes3DSubplot.scatter`

Comme en dimension 2, `scatter` permet de montrer un nuage de points.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#scatter-plots

`scatter3d_demo.py`

```
[5]: '''
=====
3D scatterplot
=====

Demonstration of a basic scatterplot in 3D.
'''

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
```

```
import numpy as np
```

```
[6]: fig = plt.figure(figsize=(4, 4))

def randrange(n, vmin, vmax):
    """
    Helper function to make an array of random numbers having shape (n, )
    with each number distributed Uniform(vmin, vmax).
    """
    return (vmax - vmin)*np.random.rand(n) + vmin

ax = fig.add_subplot(111, projection='3d')

n = 100

# For each set of style and range settings, plot n random points in the box
# defined by x in [23, 32], y in [0, 100], z in [zlow, zhigh].
for c, m, zlow, zhigh in [('r', 'o', -50, -25), ('b', '^', -30, -5)]:
    xs = randrange(n, 23, 32)
    ys = randrange(n, 0, 100)
    zs = randrange(n, zlow, zhigh)
    ax.scatter(xs, ys, zs, c=c, marker=m)

ax.set_xlabel('X Label')
plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Axes3DSubplot.plot_wireframe

Utilisez cette méthode pour dessiner en mode “fil de fer”.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#wireframe-plots.

wire3d_demo.py

```
[7]: from mpl_toolkits.mplot3d import axes3d
```

```
[8]: fig = plt.figure(figsize=(4, 4))

ax = fig.add_subplot(111, projection='3d')

# Grab some test data.
X, Y, Z = axes3d.get_test_data(0.05)

# Plot a basic wireframe.
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Axes3DSubplot.plot_surface

Comme on s'en doute, `plot_surface` sert à dessiner des surfaces dans l'espace ; ces exemples montrent surtout comment utiliser des couleurs ou des patterns.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#surface-plots.

surface3d_demo.py

```
[9]: '''
=====
3D surface (color map)
=====

Demonstrates plotting a 3D surface colored with the coolwarm color map.
The surface is made opaque by using antialiased=False.

Also demonstrates using the LinearLocator and custom formatting for the
z axis tick labels.
'''

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np
```

```
[10]: fig = plt.figure(figsize=(4, 4))

ax = fig.gca(projection='3d')

# Make data.
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                      linewidth=0, antialiased=False)

# Customize the z axis.
ax.set_zlim(-1.01, 1.01)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)
```

```
plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

surface3d_demo2.py

```
[11]: '''
=====
3D surface (solid color)
=====

Demonstrates a very basic plot of a 3D surface using a solid color.
'''

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
```

```
[12]: fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')

# Make data
u = np.linspace(0, 2 * np.pi, 30)
v = np.linspace(0, np.pi, 30)
x = 10 * np.outer(np.cos(u), np.sin(v))
y = 10 * np.outer(np.sin(u), np.sin(v))
z = 10 * np.outer(np.ones(np.size(u)), np.cos(v))

# Plot the surface
ax.plot_surface(x, y, z, color='b')

plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

surface3d_demo3.py

```
[13]: '''
=====
3D surface (checkerboard)
=====

Demonstrates plotting a 3D surface colored in a checkerboard pattern.
'''

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator
```

```
import numpy as np
```

```
[14]: fig = plt.figure(figsize=(4, 4))
      ax = fig.gca(projection='3d')

      # Make data.
      X = np.arange(-5, 5, 0.25)
      xlen = len(X)
      Y = np.arange(-5, 5, 0.25)
      ylen = len(Y)
      X, Y = np.meshgrid(X, Y)
      R = np.sqrt(X**2 + Y**2)
      Z = np.sin(R)

      # Create an empty array of strings with the same shape as the meshgrid, and
      # populate it with two colors in a checkerboard pattern.
      colortuple = ('y', 'b')
      colors = np.empty(X.shape, dtype=str)
      for y in range(ylen):
          for x in range(xlen):
              colors[x, y] = colortuple[(x + y) % len(colortuple)]

      # Plot the surface with face colors taken from the array we made.
      surf = ax.plot_surface(X, Y, Z, facecolors=colors, linewidth=0)

      # Customize the z axis.
      ax.set_zlim(-1, 1)
      ax.w_zaxis.set_major_locator(LinearLocator(6))

      plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Axes3DSubplot.plot_trisurf

plot_trisurf se prête aussi au rendu de surfaces, mais sur la base de maillages en triangles.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#tri-surface-plots.

trisurf3d_demo.py

```
[15]: '''
      =====
      Triangular 3D surfaces
      =====

      Plot a 3D surface with a triangular mesh.
      '''

      from mpl_toolkits.mplot3d import Axes3D
      import matplotlib.pyplot as plt
```

```
import numpy as np
```

```
[16]: fig = plt.figure(figsize=(4, 4))
      ax = fig.gca(projection='3d')

      n_radii = 8
      n_angles = 36

      # Make radii and angles spaces (radius r=0 omitted to eliminate duplication).
      radii = np.linspace(0.125, 1.0, n_radii)
      angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)

      # Repeat all angles for each radius.
      angles = np.repeat(angles[...], np.newaxis, n_radii, axis=1)

      # Convert polar (radii, angles) coords to cartesian (x, y) coords.
      # (0, 0) is manually added at this stage, so there will be no duplicate
      # points in the (x, y) plane.
      x = np.append(0, (radii*np.cos(angles)).flatten())
      y = np.append(0, (radii*np.sin(angles)).flatten())

      # Compute z to make the pringle surface.
      z = np.sin(-x*y)

      ax = fig.gca(projection='3d')

      ax.plot_trisurf(x, y, z, linewidth=0.2, antialiased=True)

      plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

trisurf3d_demo2.py

```
[17]: '''
      =====
      More triangular 3D surfaces
      =====

      Two additional examples of plotting surfaces with triangular mesh.

      The first demonstrates use of plot_trisurf's triangles argument, and the
      second sets a Triangulation object's mask and passes the object directly
      to plot_trisurf.
      '''

      import numpy as np
      import matplotlib.pyplot as plt
      from mpl_toolkits.mplot3d import Axes3D
      import matplotlib.tri as mtri
```

```
[18]: fig = plt.figure(figsize=(6, 3))

#=====
# First plot
#=====

# Make a mesh in the space of parameterisation variables u and v
u = np.linspace(0, 2.0 * np.pi, endpoint=True, num=50)
v = np.linspace(-0.5, 0.5, endpoint=True, num=10)
u, v = np.meshgrid(u, v)
u, v = u.flatten(), v.flatten()

# This is the Mobius mapping, taking a u, v pair and returning an x, y, z
# triple
x = (1 + 0.5 * v * np.cos(u / 2.0)) * np.cos(u)
y = (1 + 0.5 * v * np.cos(u / 2.0)) * np.sin(u)
z = 0.5 * v * np.sin(u / 2.0)

# Triangulate parameter space to determine the triangles
tri = mtri.Triangulation(u, v)

# Plot the surface. The triangles in parameter space determine which x, y, z
# points are connected by an edge.
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_trisurf(x, y, z, triangles=tri.triangles, cmap=plt.cm.Spectral)
ax.set_zlim(-1, 1)

#=====
# Second plot
#=====

# Make parameter spaces radii and angles.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi/n_angles

# Map radius, angle pairs to x, y, z points.
x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = mtri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid**2 + ymid**2 < min_radius**2, 1, 0)
triang.set_mask(mask)

# Plot the surface.
```

```
ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.plot_trisurf(triang, z, cmap=plt.cm.CMRmap)

plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Axes3DSubplot.contour

Pour dessiner des contours.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#contour-plots.

contour3d_demo.py

```
[19]: from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

```
[20]: fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

contour3d_demo2.py

```
[21]: from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

```
[22]: fig = plt.figure(figsize=(4, 4))
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, extend3d=True, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

contour3d_demo3.py

```
[23]: from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

```
[24]: fig = plt.figure(figsize=(4, 4))
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contour(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)

ax.set_xlabel('X')
ax.set_xlim(-40, 40)
ax.set_ylabel('Y')
ax.set_ylim(-40, 40)
ax.set_zlabel('Z')
ax.set_zlim(-100, 100)

plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Axes3DSubplot.contourf

Comme `Axes3DSubplot.contour`, mais avec un rendu plein plutôt que sous forme de lignes (le `f` provient de l'anglais *filled*).

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#filled-contour-plots.

contourf3d_demo.py

```
[25]: from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

```
[26]: fig = plt.figure(figsize=(4, 4))
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contourf(X, Y, Z, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

contourf3d_demo2.py

```
[27]: """
.. versionadded:: 1.1.0
   This demo depends on new features added to contourf3d.
"""

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

```
[28]: fig = plt.figure(figsize=(4, 4))
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contourf(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contourf(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
cset = ax.contourf(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)

ax.set_xlabel('X')
ax.set_xlim(-40, 40)
ax.set_ylabel('Y')
ax.set_ylim(-40, 40)
ax.set_zlabel('Z')
ax.set_zlim(-100, 100)

plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Axes3DSubplot.add_collection3d

Pour afficher des polygones.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#polygon-plots.

```
[29]: """
=====
Generate polygons to fill under 3D line graph
=====

Demonstrate how to create polygons which fill the space under a line
graph. In this example polygons are semi-transparent, creating a sort
of 'jagged stained glass' effect.
"""

from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection
```

```
import matplotlib.pyplot as plt
from matplotlib import colors as mcolors
import numpy as np
```

```
[30]: fig = plt.figure(figsize=(4, 4))
      ax = fig.gca(projection='3d')

      def cc(arg):
          return mcolors.to_rgba(arg, alpha=0.6)

      xs = np.arange(0, 10, 0.4)
      verts = []
      zs = [0.0, 1.0, 2.0, 3.0]
      for z in zs:
          ys = np.random.rand(len(xs))
          ys[0], ys[-1] = 0, 0
          verts.append(list(zip(xs, ys)))

      poly = PolyCollection(verts, facecolors=[cc('r'), cc('g'), cc('b'),
                                              cc('y')])
      poly.set_alpha(0.7)
      ax.add_collection3d(poly, zs=zs, zdir='y')

      ax.set_xlabel('X')
      ax.set_xlim3d(0, 10)
      ax.set_ylabel('Y')
      ax.set_ylim3d(-1, 4)
      ax.set_zlabel('Z')
      ax.set_zlim3d(0, 1)

      plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Axes3DSubplot.bar

Pour construire des diagrammes à barres.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#bar-plots.

bars3d_demo.py

```
[31]: """
      =====
      Create 2D bar graphs in different planes
      =====

      Demonstrates making a 3D plot which has 2D bar graphs projected onto
      planes y=0, y=1, etc.
      """

      from mpl_toolkits.mplot3d import Axes3D
```

```
import matplotlib.pyplot as plt
import numpy as np
```

```
[32]: fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
for c, z in zip(['r', 'g', 'b', 'y'], [30, 20, 10, 0]):
    xs = np.arange(20)
    ys = np.random.rand(20)

    # You can provide either a single color or an array. To demonstrate this,
    # the first bar of each set will be colored cyan.
    cs = [c] * len(xs)
    cs[0] = 'c'
    ax.bar(xs, ys, zs=z, zdir='y', color=cs, alpha=0.8)

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Axes3DSubplot.quiver

Pour afficher des champs de vecteurs sous forme de traits.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#quiver.

quiver3d_demo.py

```
[33]: '''
=====
3D quiver plot
=====

Demonstrates plotting directional arrows at points on a 3d meshgrid.
'''

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
```

```
[34]: fig = plt.figure(figsize=(4, 4))
ax = fig.gca(projection='3d')

# Make the grid
x, y, z = np.meshgrid(np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.8))
```

```
# Make the direction data for the arrows
u = np.sin(np.pi * x) * np.cos(np.pi * y) * np.cos(np.pi * z)
v = -np.cos(np.pi * x) * np.sin(np.pi * y) * np.cos(np.pi * z)
w = (np.sqrt(2.0 / 3.0) * np.cos(np.pi * x) * np.cos(np.pi * y) *
      np.sin(np.pi * z))

ax.quiver(x, y, z, u, v, w, length=0.1, normalize=True)

plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

7.29 w7-s10-c3-notebooks-interactifs

Notebooks interactifs

7.29.1 Complément - niveau basique

Pour conclure cette série sur les outils de visualisation, nous allons voir quelques fonctionnalités disponibles uniquement dans l'environnement des notebooks, et qui offrent des possibilités supplémentaires par rapport aux visualisations que l'on a vues jusqu'à maintenant.

installation

Pour exécuter ou créer un notebook depuis votre ordinateur, il vous faut installer Jupyter, ce que se fait bien sûr depuis le terminal :

```
pip install jupyter
```

En 2020 il existe deux versions de l'interface Jupyter dites classic et lab, la seconde étant plus puissante en termes d'UI ; pour installer le tout, faire plutôt

```
pip install jupyterlab
```

Pour lancer un serveur jupyter, faire selon le mode choisi

```
jupyter notebook
# ou
jupyter lab
```

Contenus

Pour le contenu des notebooks :

- une cellule est marquée comme étant soit du code, soit du texte(markdown) ;

- pour les cellules de markdown, on peut très simplement :
 - insérer des formules mathématiques, en insérant un fragment de \LaTeX entre deux simples \$, comme $\forall x \in \mathbb{R}$, ou encore sur une ligne séparée en entourant entre deux doubles dollars \$\$, comme

$$\forall \epsilon > 0, \exists \alpha > 0, \forall x, |x - x_0| < \epsilon \implies |f(x) - f(x_0)| < \epsilon$$

- et bien sûr [toute la panoplie des effets markdown](#), quoi qu'il faut se méfier car tout cela n'est pas très bien standardisé actuellement.
- un notebook choisit son kernel (en clair son langage) ; le mot Jupyter vient de Julia + Python + R, et aujourd'hui il y a moyen de faire tourner presque tous les langages, même `bash` et `C++` (mais en mode interprété bien sûr)

Courbes

```
[1]: import numpy as np
import matplotlib.pyplot as plt
```

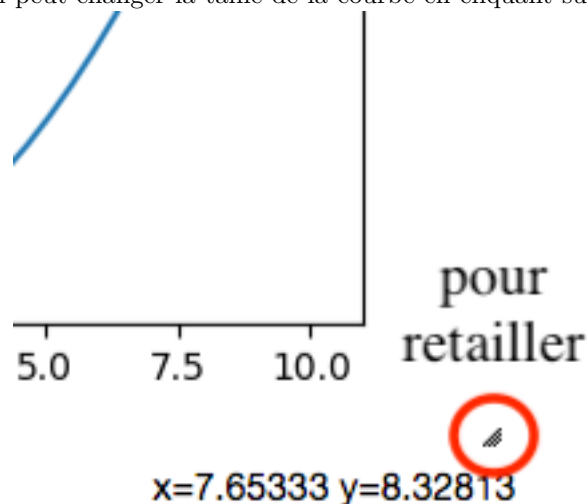
Comme on l'a déjà vu plein de fois, la bonne façon de créer un graphique matplotlib c'est avec la formule magique suivante :

```
[2]: # ça c'est pour choisir la sortie 'notebook'
%matplotlib notebook

# et ça c'est pour dire 'interactive on'
# pour éviter de devoir plt.show() tout le temps
plt.ion()
```

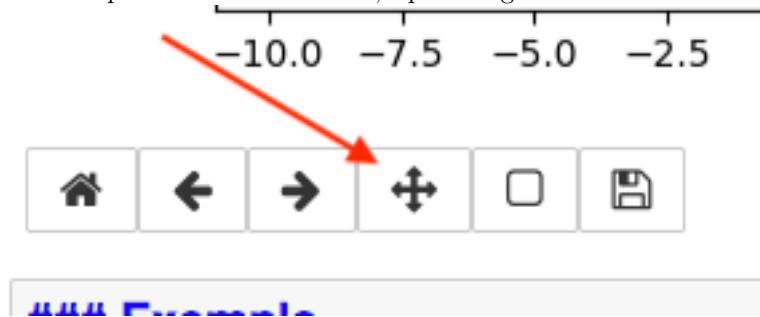
Avec ces réglages - enfin surtout le premier - il y a pas mal de possibilités qui sont très pratiques :

- pour commencer on peut changer la taille de la courbe en cliquant sur le petit coin visible en bas



à droite de la figure

— les courbes apparaissent avec une barre d'outils en dessous; entraînez-vous à utiliser par exemple l'outil de zoom, pour agrandir et vous déplacer dans la courbe



À titre d'exercice, sur cette courbe le nombre d'or correspond à une des racines du polynôme, à vous de trouver sa valeur avec une précision de

```
[3]: X = np.linspace(-2, 2)
ZERO = X * 0
def golden(x):
    return x**2 - x - 1
plt.plot(X, golden(X));
plt.plot(X, ZERO);
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Voici à quoi je suis arrivé de mon côté (je ne dis pas que c'est forcément la méthode la plus rapide pour trouver le nombre d'or;-) :

Mais tous les outils de visualisation décents vous proposer des mécanismes analogues, soyez-y attentifs car ça fait parfois gagner beaucoup de temps.

Exemple de notebook interactif

Je vous signale enfin un [exemple de notebook publié par la célèbre revue Nature](#), qui pourra vous donner une idée de ce qu'il est possible de faire avec un notebook interactif. Interactif dans le sens où on peut faire varier les paramètres d'une expérience et voir l'impact du changement se refléter immédiatement sur la visualisation.

Comme il n'est malheureusement plus actif en ligne semble-t-il, je vous invite à le faire marcher localement à partir [de la version sur github ici](#).

7.29.2 Complément - niveau intermédiaire

Une visualisation interactive simple : **interact**

Pour refaire de notre côté quelque chose d'analogue, nous allons commencer par animer la fonction sinus, avec un bouton pour régler la fréquence. Pour cela nous allons utiliser la fonction **interact**; à nouveau c'est un utilitaire qui fait partie de l'écosystème des notebooks, et plus précisément du module **ipywidgets** :

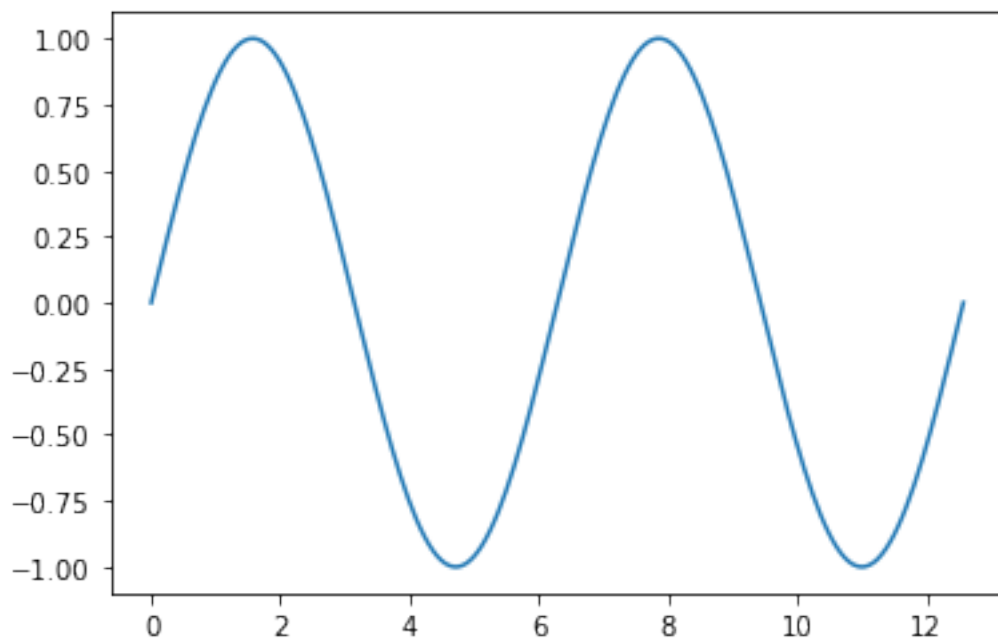
```
[4]: # dans cette partie on a besoin de  
# revenir dans un mode plus usuel  
%matplotlib inline
```

```
[5]: from ipywidgets import interact
```

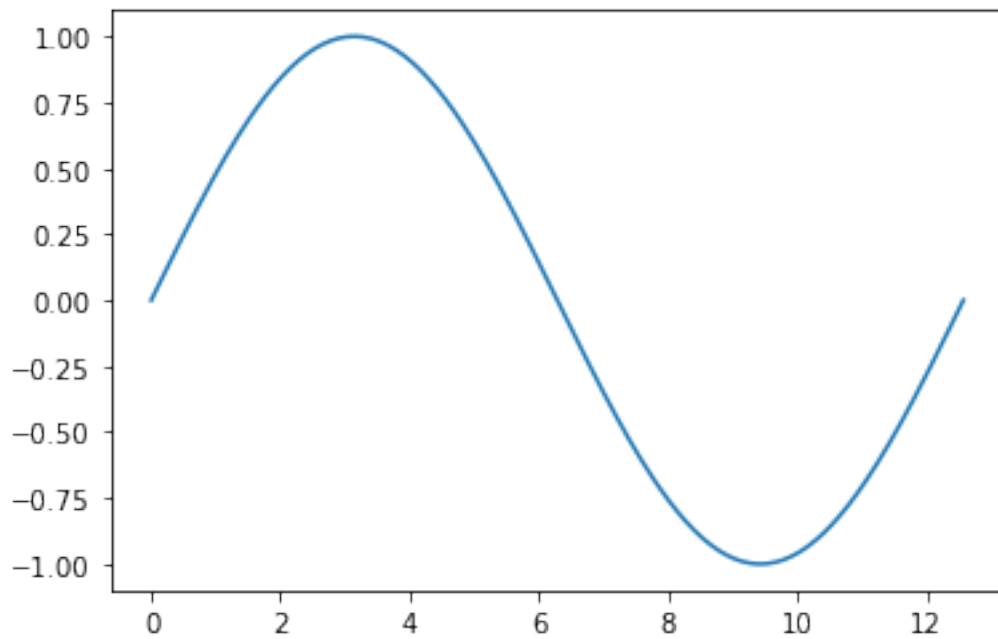
Dans un premier temps, j'écris une fonction qui prend en paramètre la fréquence, et qui dessine la fonction sinus sur un intervalle fixe de 0. à 4π :

```
[6]: def sinus(freq):  
    X = np.linspace(0., 4*np.pi, 200)  
    Y = np.sin(freq*X)  
    plt.plot(X, Y)
```

```
[7]: sinus(1)
```



```
[8]: sinus(0.5)
```

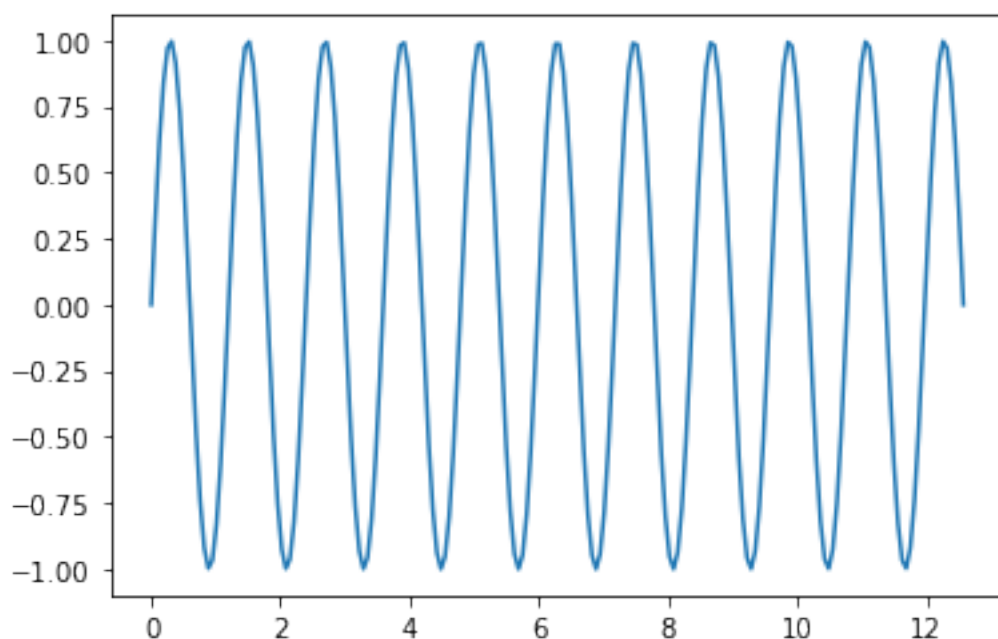



Maintenant, plutôt que de tracer individuellement les courbes une à une, j'utilise `interact` qui va m'afficher une réglette pour changer le paramètre `freq`. Ça se présente comme ceci :

```
[9]: # je change maintenant la taille des visualisations  
plt.figure(figsize=(12, 4));
```

<Figure size 864x288 with 0 Axes>

```
[10]: interact(sinus, freq=(0.5, 10., 0.25));
```



Mécanisme d'`interact`

La fonction `interact` s'attend à recevoir :

- en premier argument : une fonction `f` ;
- et ensuite autant d'arguments nommés supplémentaires que de paramètres attendus par `f`.

Comme dans mon cas la fonction `sinus` attend un paramètre nommé `freq`, le deuxième argument de `interact` lui est passé aussi avec le nom `freq`.

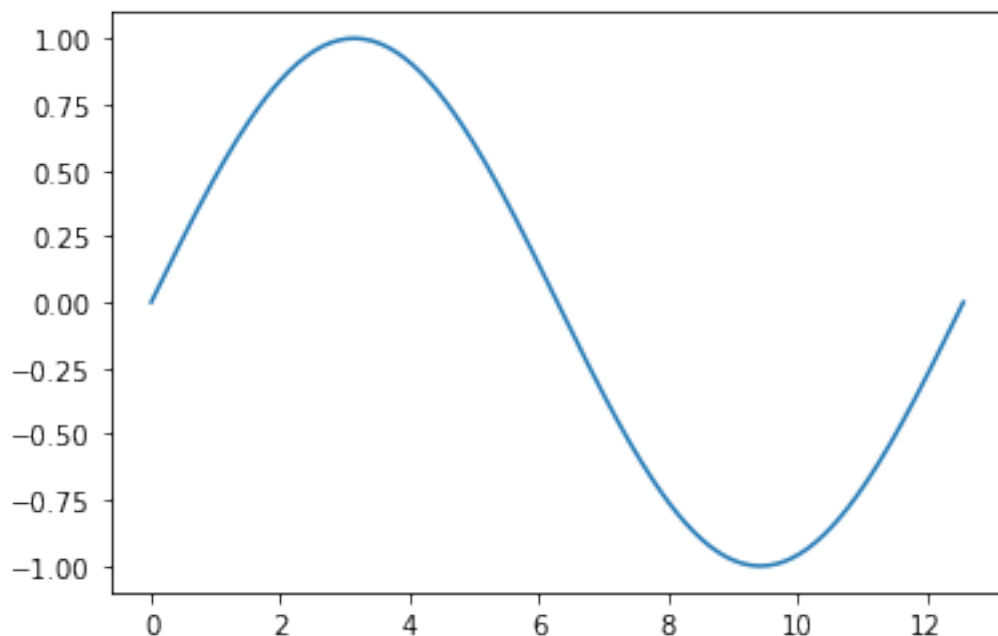
Les objets `Slider`

Chacun des arguments à `interact` (en plus de la fonction) correspond à un objet de type `Slider` (dans la ménagerie de `ipywidget`). Ici en passant juste le tuple `(0.5, 10., 0.25)` j'utilise un raccourci pour dire que je veux pouvoir régler le paramètre `freq` sur une plage allant de 0.5 à 10 avec un pas de 0.25.

Mon premier exemple avec `interact` est en réalité équivalent à ceci :

```
[11]: from ipywidgets import FloatSlider

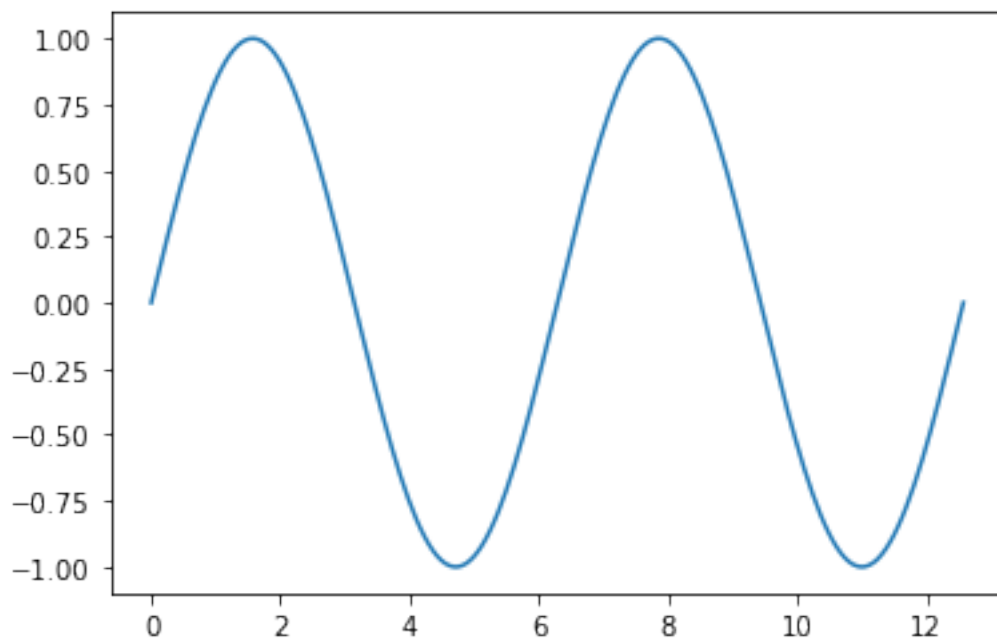
[12]: # exactement équivalent à la version ci-dessus
interact(sinus, freq=FloatSlider(min=0.5, max=10., step=0.25));
```



Mais en utilisant la forme bavarde, je peux choisir davantage d'options, comme notamment :

- mettre `continuous_update = False`; l'effet de ce réglage, c'est que l'on met à jour la figure seulement lorsque je lâche la réglette; c'est utile lorsque les calculs sont un peu lents, comme ici avec l'infrastructure notebook qui est à distance;
- mettre `value=1.` pour choisir la valeur initiale :

```
[13]: # exactement équivalent à la version ci-dessus
# sauf qu'on ne redessine que lorsque la réglette
# est relâchée
interact(sinus, freq=FloatSlider(min=0.5, max=10.,
                                step=0.25, value=1.,
                                continuous_update=False));
```



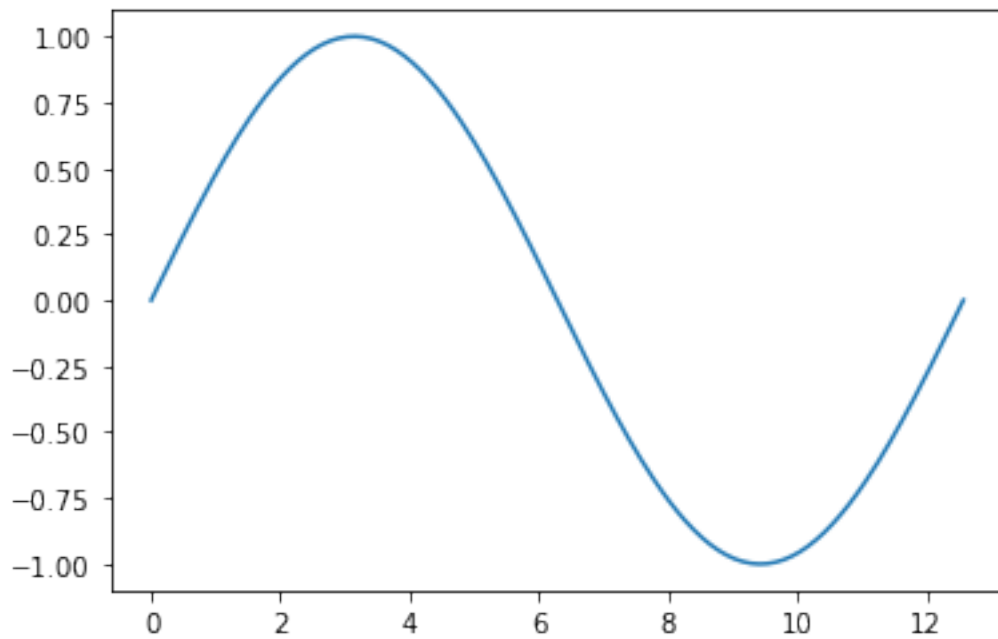
Plusieurs paramètres

Voyons tout de suite un exemple avec deux paramètres, je vais écrire maintenant une fonction qui me permet de changer aussi la phase :

```
[14]: def sinus2(freq, phase):
X = np.linspace(0., 4*np.pi, 200)
Y = np.sin(freq*(X+phase))
plt.plot(X, Y)
```

Et donc maintenant je passe à `interact` un troisième paramètre :

```
[15]: interact(sinus2,
               freq=FloatSlider(min=0.5, max=10., step=0.5,
                               continuous_update=False),
               phase=FloatSlider(min=0., max=2*np.pi, step=np.pi/6,
                               continuous_update=False),
               );
```

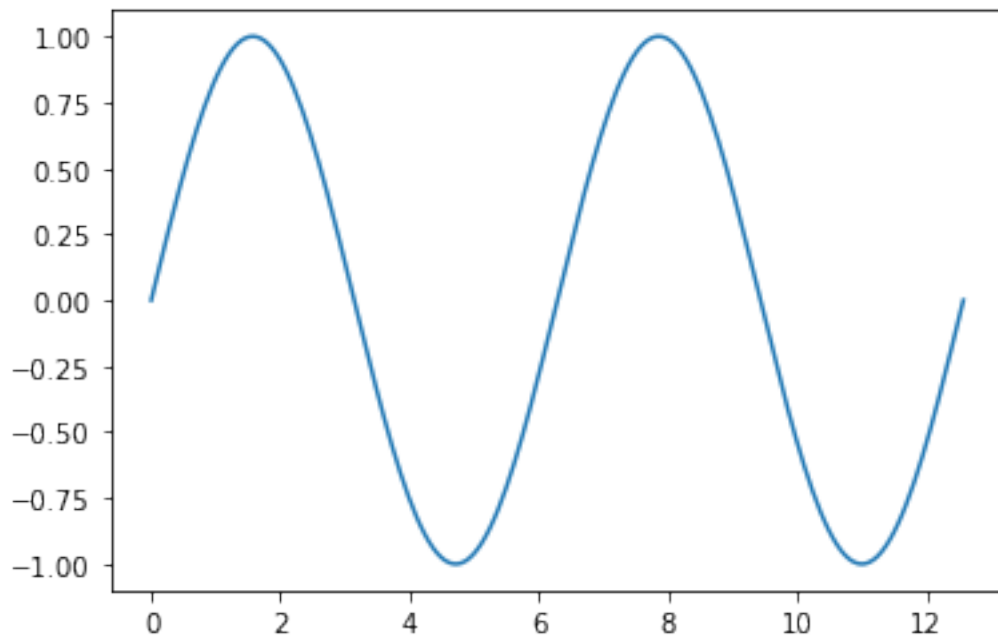


Bouche-trou : **fixed**

Si j'ai une fonction qui prend plus de paramètres que je ne veux montrer de réglettes, je peux fixer un des paramètres par exemple comme ceci :

```
[16]: from ipywidgets import fixed
```

```
[17]: # avec une fonction à deux argument,  
# je peux en fixer un, et n'avoir qu'une réglette  
# pour fixer celui qui est libre  
interact(sinus2, freq=fixed(1.),  
         phase=FloatSlider(min=0., max=2*np.pi, step=np.pi/6),  
         );
```



7.29.3 Widgets

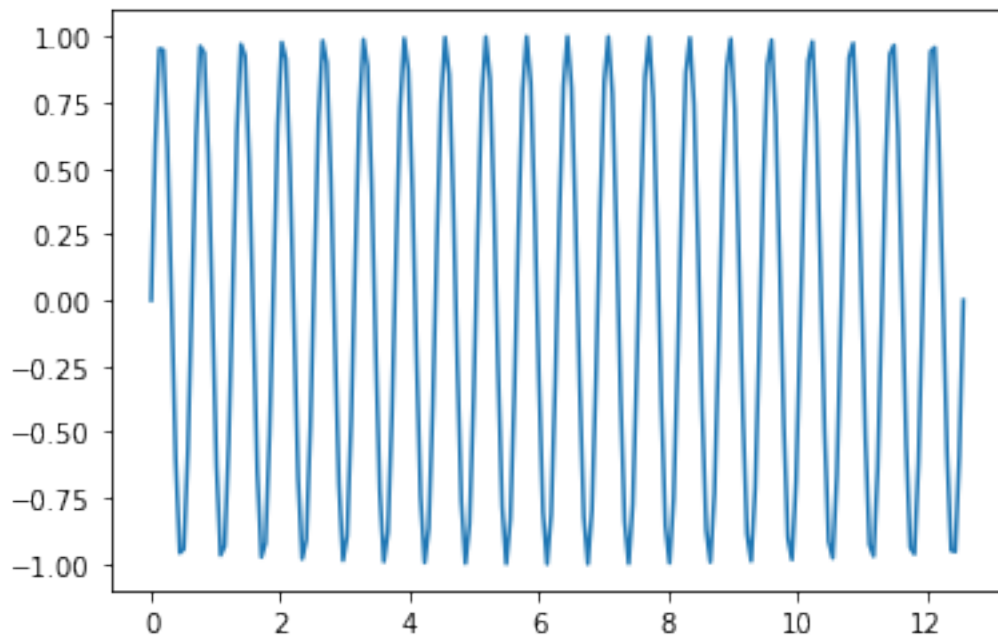
Il existe toute une famille de widgets, dont `FloatSlider` est l'exemple le plus courant, mais vous pouvez aussi :

- créer des radio bouton pour entrer un paramètre booléen ;
- ou une saisie de texte pour entrer un paramètre de type `str` ;
- ou une liste à choix multiples...

Bref, vous pouvez créer une mini interface-utilisateur avec des objets graphiques simples choisis dans une palette assez complète pour ce type d'application.

Voyez [les détails complets sur readthedocs.io](https://readthedocs.io)

```
[18]: # de même qu'un tuple était ci-dessus un raccourci pour un FloatSlider  
# une liste ou un dictionnaire est transformé(e) en un Dropdown  
interact(sinus, freq={'rapide': 10., 'moyenne': 1., 'lente': 0.1});
```



Voyez la [liste complète des widgets ici](#).

Dashboards

Lorsqu'on a besoin de faire une interface un peu plus soignée, on peut créer sa propre disposition de boutons et autres réglages.

Voici un exemple de dashboard, uniquement pour vous donner une meilleure idée, qui pour changer agit sur une visualisation réalisée avec plot.ly plutôt que matplotlib :

```
[19]: ###
# EDIT - juillet 2019 -
###
# en faisant simplement
# pip install plotly
# j'obtiens la version 4.0.0 de plotly
# qui de toute évidence a changé depuis
# la rédaction initiale de ce document
#
# c'est pourquoi je m'arrange pour charger plotly-3.10.0
# dont je sais qu'elle fonctionne avec ce code:
# pip install plotly==3.10.0
#
# si quelqu'un sait faire marcher la nouvelle version
# je suis intéressé pour l'intégrer, n'hésitez pas à
# m'envoyer un pull request
#

import plotly
plotly.__version__
```

[19]: '4.7.1'

```
[20]: # on importe la bibliothèque plot.ly
import chart_studio.plotly as py
import plotly.graph_objs as go
```

```
[21]: # il est impératif d'utiliser plot.ly en mode 'offline'
# pour in mode interactif,
# car sinon les affichages sont beaucoup trop lents
import plotly.offline as pyoff

pyoff.init_notebook_mode()
```

```
[22]: # les widgets pour construire le tableau de bord
from ipywidgets import (interactive_output,
                        IntSlider, Dropdown, Layout, HBox, VBox, Text)
from IPython.display import display
```

```
[23]: # une fonction sinus à 4 réglages
# qu'on réalise pour changer avec plot.ly
# et non pas avec matplotlib
def sinus4(freq, phase, amplitude, domain):

    X = np.linspace(0., domain*np.pi, 500)
    Y = amplitude * np.sin(freq*(X+phase))

    data = [ go.Scatter(x=X, y=Y, mode='lines', name='sinus') ]
    # je fixe l'amplitude à 10 pour que les animations
    # soient plus parlantes
    layout = go.Layout(
        yaxis = {'range' : [-10, 10]},
        title="Exemple de graphique interactif avec dashboard",
        height=500,
        width=500,
    )
    figure = go.Figure(data=data, layout=layout)
    pyoff.iplot(figure)
```

```
[24]: def my_dashboard():
    """
    create and display a dashboard
    return a dictionary name->widget suitable for interactive_output
    """

    # dashboard pieces as widgets
    l_75 = Layout(width='75%')
    l_50 = Layout(width='50%')
    l_25 = Layout(width='25%')

    w_freq = Dropdown(options=list(range(1, 10)),
                      value = 1,
                      description = "fréquence",
                      layout=l_50)
    w_phase = FloatSlider(min=0., max = 2*np.pi, step=np.pi/12,
                          description="phase",
                          value=0., layout=l_75)
```

```

w_amplitude = Dropdown(options={"micro" : .1,
                                "mini" : .5,
                                "normal" : 1.,
                                "grand" : 3.,
                                "énorme" : 10.},
                        value = 3.,
                        description = "amplitude",
                        layout = l_25)
w_domain = IntSlider(min=1, max=10, description="dom. n * ", layout=l_50)

# make up a dashboard
dashboard = VBox([HBox([w_amplitude, w_phase]),
                  HBox([w_domain, w_freq]),
                  ])
display(dashboard)
return dict(freq=w_freq, phase=w_phase,
            amplitude=w_amplitude, domain=w_domain)

```

Avec tout ceci en place on peut montrer un dialogue interactif pour changer tous les paramètres de sinus4.

```

[25]: # interactively call sinus4
      # attention il reste un bug:
      # au tout début rien ne s'affiche,
      # il faut faire bouger au moins un réglage
      interactive_output(sinus4, my_dashboard())

```

Output()

7.30 w7-s10-c4-animations-matplotlib

Animations interactives avec **matplotlib**

7.30.1 Complément - niveau avancé

Nous allons voir dans ce notebook comment créer une animation avec matplotlib et tirer parti des widgets dans un notebook pour rendre ces animations interactives.

Ce sera l'occasion de décortiquer un exemple un peu sophistiqué, où l'utilisation d'un générateur permet de rendre l'implémentation plus propre et plus élégante.

Le sujet

En guise d'illustration, nous allons créer :

- une animation matplotlib : disons que l'on veut faire défiler horizontalement une sinusoïde ;
- un widget interactif : disons que l'on veut pouvoir changer la vitesse de défilement.

Les outils

Pour fabriquer cela nous aurons besoin principalement :

- de la librairie d'animation de `matplotlib`, et spécifiquement le sous-package `animation`,
- et des widgets du module `ipywidgets`.

```
[1]: import numpy as np
      from matplotlib import pyplot as plt
      from matplotlib import animation
```

```
[2]: from IPython.display import display as display_widget
      from ipywidgets import IntSlider
```

La logique

Dans un notebook précédent nous avons abordé la fonction `interact`, de la librairie `ipywidgets`, qui nous permettait d'appeler interactivement une fonction avec des arguments choisis interactivement via une série de widgets.

Si on essaie d'utiliser `interact` pour faire des animations, l'effet visuel, qui revient à effacer/recalculer une visualisation à chaque changement de valeur des entrées, donne un rendu peu agréable à l'œil.

C'est pourquoi ici la logique va être un petit peu différente :

- c'est une fonction native de `matplotlib` qui implémente la boucle principale, en travaillant sur un objet `Figure`,
- et le widget est utilisé uniquement pour modifier une variable python ;
- pour simplifier notre code, l'échange d'informations entre ces deux morceaux se fait le plus simplement possible, via une variable globale.

Bien entendu cette dernière pratique n'est pas recommandée dans du code de production, et le lecteur intéressé est invité à améliorer ce point.

Version non interactive et basique

Pour commencer nous allons voir comment utiliser `matplotlib.animation` sans interactivité.

Cette version est inspirée du [tutorial matplotlib sur les animations](#), qui montre d'ailleurs d'autres animations plus complexes et convaincantes, comme le double pendule par exemple.

Mais avant tout choisissons ce mode de rendu :

```
[3]: %matplotlib notebook
```

Nous allons utiliser la fonction `animation.FuncAnimation` ; celle-ci s'attend à recevoir en argument, principalement :

- une figure,
- et une fonction d'affichage.

La logique est que la fonction d'affichage est appelée à intervalles réguliers par `FuncAnimation`, elle doit retourner un itérable d'objets affichable dans la figure.

Dans notre cas, nous allons créer une instance unique d'un objet `plot`; cette instance sera modifiée à chaque frame par la fonction d'affichage, qui le renverra dans un tuple à un élément (ceci parce qu'un itérable est attendu).

Version basique dite tout-en-un

```
[4]: import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

# on commence par créer une figure;
figure1 = plt.figure(figsize=(4, 2))

# en général pour une animation
# il est important de fixer les bornes en x et en y
# pour ne pas avoir d'artefacts de changement d'échelle
# pendant l'animation
ax1 = plt.axes(xlim=(0, 2), ylim=(-1.5, 1.5))

# on crée aussi un plot vide qui va être modifié à chaque frame
line1, = ax1.plot([], [], linewidth=2)

# la vitesse de défilement
speed = 1

# une globale; c'est vilain !
offset = 0.

# la fonction qui calcule et affiche chaque frame
def compute_and_display(n):
    global offset
    offset += speed / 100
    x = np.linspace(0, 2, 1000)
    y = np.sin(2 * np.pi * (x - offset))
    line1.set_data(x, y)
    return line1,

# la fonction magique pour animer une figure
# blit=True est une optimisation graphique
# pour ne rafficher que le nécessaire
anim = animation.FuncAnimation(figure1,
                                func=compute_and_display,
                                frames=50, repeat=False,
                                interval=40, blit=True)

plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Séparation calcul et affichage

```
[5]: plt.ion()
```

On voit qu'on a appelé `FuncAnimation` avec `frames=50` et `interval=40` (ms); ce qui correspond donc à 25 images par seconde, soit une durée de deux secondes.

Profitons-en pour voir tout de suite une amélioration possible. Il serait souhaitable de séparer :

- d'une part la logique du calcul - ou de l'acquisition, si on voulait par exemple faire du postprocessing temps réel d'images vidéo,
- et d'autre part l'affichage à proprement parler.

Pour cela, remarquez que le paramètre `frames` est documenté comme pouvant être un itérateur. La logique en fait à l'oeuvre dans `FuncAnimation` est que

- `frames` est un itérateur qui va énumérer des données,
- à chaque frame cet itérateur est avancé avec `next()`, et son retour est passé à la fonction d'affichage.

En guise de commodité, lorsqu'on passe comme ci-dessus `frames=200`, la fonction transforme cela en `frames=range(200)`. C'est pourquoi d'ailleurs il est important que `compute_and_display` accepte un paramètre unique, même si nous n'en avons pas eu besoin.

Cette constatation nous amène à une deuxième version, en concevant un générateur pour le calcul, qui est passé à `FuncAnimation` comme paramètre `frames`.

Version non interactive, mais avec séparation calcul / affichage

```
[6]: import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

figure2 = plt.figure(figsize=(4, 2))
ax2 = plt.axes(xlim=(0, 2), ylim=(-1.5, 1.5))
line2, = ax2.plot([], [], linewidth=2)

# la vitesse de défilement
speed = 1

# remarquez qu'on n'a plus besoin de globale ici
# une locale dans le générateur est bien plus propre

# la logique du calcul est conçue comme un générateur
def compute():
    offset = 0.
    # nous sommes dans un générateur, il n'y a pas
    # de contrindication à tourner indéfiniment
    while True:
        offset += speed / 100
        x = np.linspace(0, 2, 1000)
        y = np.sin(2 * np.pi * (x - offset))
        # on décide de retourner un tuple X, Y
        # qui est passé tel-qu'el à l'affichage
        yield x, y
```

```

# la fonction qui affiche
def display(frame):
    # on retrouve nos deux éléments x et y
    x, y = frame
    # il n'y a plus qu'à les afficher
    line2.set_data(x, y)
    return line2,

anim = animation.FuncAnimation.figure2,
                                func=display,
                                frames=compute(),
                                interval=40, blit=True)

plt.show()

```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Cette fois l'animation ne se termine jamais, mais dans le notebook vous pouvez cliquer le bouton bleu en haut à droite de la figure pour la faire cesser.

Avec interactivité

Pour rendre ceci interactif, nous allons simplement ajouter un widget qui nous permettra de régler la vitesse de défilement.

Version interactive avec widget pour choisir la vitesse

```

[7]: import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

from IPython.display import display as display_widget
from ipywidgets import IntSlider

figure3 = plt.figure(figsize=(4, 2))
ax3 = plt.axes(xlim=(0, 2), ylim=(-1.5, 1.5))
line3, = ax3.plot([], [], linewidth=2)

# un widget pour choisir la vitesse de défilement
speed_slider = IntSlider(min=1, max=10, value=3,
                        description="Vitesse:")

def compute():
    offset = 0.
    # nous sommes dans un générateur, il n'y a pas
    # de contraindication à tourner indéfiniment
    while True:
        # on accède à la vitesse via le widget
        offset += speed_slider.value / 100
        x = np.linspace(0, 2, 1000)
        y = np.sin(2 * np.pi * (x - offset))

```

```

        # on décide de retourner un tuple X, Y
        # qui est passé tel-qu'el à l'affichage
        yield x, y

# la fonction qui affiche
def display(frame):
    # on retrouve nos deux éléments x et y
    x, y = frame
    # il n'y a plus qu'à les afficher
    line3.set_data(x, y)
    return line3,

anim = animation.FuncAnimation.figure3,
                                func=display,
                                frames=compute(),
                                interval=40, blit=True)

display_widget(speed_slider)
plt.show()

```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

IntSlider(value=3, description='Vitesse:', max=10, min=1)

Conclusion

Avec une approche de programmation plus traditionnelle, on pourrait penser avoir besoin de recourir à plusieurs threads pour implémenter ce genre de visualisation interactive.

En effet, vous remarquerez que dans cette dernière version, en termes de parallélisme, on peut avoir l'impression que 3 choses ont lieu principalement en même temps :

- la logique de calcul, qui en substance est décrite comme un unique `while True:`,
- la logique d'affichage, qui est cadencée par `FuncAnimation`,
- et la logique interactive, qui gère le widget sur interaction de l'utilisateur.

Le point à retenir ici est que, grâce à la fois au générateur et au notebook, on n'a pas du tout besoin de gérer soi-même cet aspect de programmation parallèle.

Nous verrons d'ailleurs dans la semaine suivante comment le paradigme de programmation asynchrone de Python a été bâti, au dessus de cette capacité qu'offre le générateur, pour utiliser ce type d'approche de manière systématique, afin de faire tourner dans un seul thread et de manière transparente, un grand nombre de logiques.

Pour en savoir plus

Voyez :

- [la documentation du module animation](#),

- ainsi que [le tutoriel dont on s'est inspiré pour ce notebook](#), surtout pour voir d'autres animations plus élaborées.

7.31 w7-s10-c5-bokeh-et-al

Autres bibliothèques de visualisation

7.31.1 Complément - niveau basique

Pour conclure cette séquence sur les outils de visualisation, nous allons très rapidement évoquer des alternatives à la bibliothèque `matplotlib`, sachant que le domaine est en pleine expansion.

Le poids du passé

On a vu que `matplotlib` est un outil relativement complet. Toutefois, on peut lui reprocher deux défauts majeurs.

- D'une part, `matplotlib` a choisi d'offrir une interface aussi proche que possible de ce qui existait préalablement en MatLab. C'est un choix tout à fait judicieux dans l'optique d'attirer la communauté MatLab à des outils open source basés sur Python et numpy. Mais en contrepartie, cela implique d'adopter tels quels des choix de conception.
- Et notamment, en suivant cette approche on hérite d'un modèle mental qui est plus orienté vers la sortie vers du papier que vers la création de documents interactifs.

Ceci, ajouté à l'explosion du domaine de l'analyse et de la visualisation de données, explique la largeur de l'offre en matière de bibliothèques de visualisation alternatives.

Dans ce complément nous allons explorer notamment quelques techniques qui permettent de faire des visualisations interactives ; c'est-à-dire où l'on peut modifier la visualisation en fonction de paramètres, réglables facilement.

C'est quelque chose qui demande un peu de soin car, si on utilise `interact()` brutalement, on obtient des visualisations qui "flashent", car à chaque changement du contexte on recalcule toute une image, plutôt que de modifier l'image précédente. Ça semble un détail, mais l'oeil est très sensible à ce type d'artefact, et à l'expérience ce détail a plus d'impact qu'on ne pense.

bokeh

Commençons par signaler notamment la bibliothèque [bokeh](#), qui est développée principalement par Anaconda, dans un modèle open source.

bokeh présente quelques bonnes propriétés qui nous semblent mériter d'être signalées.

Pour commencer cette bibliothèque utilise une architecture qui permet de penser la visualisation comme quelque chose d'interactif (disons une page html), et non pas de figé comme lorsqu'on pense en termes de feuille de papier. Notamment elle permet de faire collaborer du code Python avec du code JavaScript, qui offre immédiatement des possibilités bien plus pertinentes lorsqu'il s'agit de créer des interactions utilisateur qui soient attractives et efficaces. Signalons en passant, à cet égard, qu'elle utilise [la librairie JavaScript d3.js](#), qui est devenu un standard de fait plus ou moins incontournable dans le domaine de la visualisation.

En tout état de cause, elle offre une interface de programmation qui tient compte d'environnements comme les notebooks, ce qui peut s'avérer un atout précieux si vous utilisez massivement ce support, comme on va le voir, précisément, dans ce notebook.

Il peut aussi être intéressant de savoir que **bokeh** offre des possibilités natives de [visualisation de graphes](#) et de [données géographiques](#).

Par contre à ce stade du développement, la visualisation en 3D n'est sans doute pas le point fort de **bokeh**. C'est une option qui reste possible (voir [par exemple ceci](#)), mais cela est pour l'instant considéré comme une extension de la librairie, et donc n'est accessible qu'au prix de l'écriture de code javascript.

Pour une présentation plus complète, je vous renvoie à [la documentation utilisateur](#).

bokeh dans les notebooks

Nous allons rapidement illustrer ici comment **bokeh** s'interface avec l'environnement des notebooks pour créer une visualisation interactive. Vous remarquerez que dans le code qui suit, on n'a pas eu besoin de mentionner de `magic ipython`, comme lorsqu'on avait du faire dans le complément sur les notebooks interactifs :

```
%matplotlib notebook
```

```
[1]: import numpy as np

[2]: # l'attirail de notebooks interactifs
from ipywidgets import interact, fixed, FloatSlider, Dropdown

[3]: # les imports pour bokeh
from bokeh.plotting import figure, show
# dans la rubrique entrée-sortie, on trouve
# les outils pour produire du html
# (le mode par défaut)
# ou pour interagir avec un notebook
from bokeh.io import push_notebook, output_notebook

[4]: # c'est cette déclaration qui remplace
# si on veut la magic '%matplotlib notebook'
output_notebook()

[5]: # on crée un objet figure
figure1 = figure(
    title="fonctions trigonométriques",
    plot_height=300, plot_width=600,
    # c'est là notamment qu'on précise
    # l'intervalle en y
    y_range=(-5, 5),
)

[6]: # on initialise la figure en créant
# un objet courbe
x = np.linspace(0, 2*np.pi, 2000)
y = np.sin(x)
courbe_trigo = figure1.line(x, y, color="#2222aa", line_width=3)
```

```
[7]: # la fonction de mise à jour, qui sera connectée
# à interact
def update_trigo(function, frequency=1,
                  amplitude=1, phase=0,
                  # l'objet handle correspond
                  # à une figure à mettre à jour
                  *, handle):
    # c'est ici qu'on modifie les données
    # utilisées pour produire la courbe
    courbe_trigo.data_source.data['y'] = \
        amplitude * function(frequency * x + phase)
    # et c'est ici qu'on provoque la mise à jour
    push_notebook(handle=handle)
```

```
[8]: # au moment où on matérialise l'objet figure
# on récupère une `handle` qui lui correspond
handle1 = show.figure1, notebook_handle=True)
```

```
[9]: # maintenant on peut créer un interacteur
interact(update_trigo,
          # on peut définir les options sont des tuples (label, valeur)
          # et ici nos valeurs sont des fonctions
          function=Dropdown(options =(("sinus", np.sin),
                                      ("cosinus", np.cos),
                                      ("tangeante", np.tan))),
          frequency=(1,20),
          amplitude=[0.5, 1, 3, 5],
          phase=(0, 2*np.pi, 0.05),
          handle=fixed(handle1),
          );
```

```
interactive(children=(Dropdown(description='function', options=(('sinus', <ufunc 'sin'>), ('cosinus',
```

7.31.2 Complément : niveau intermédiaire

Une classe pour ce genre d'usages

En termes de conception, notre approche jusqu'ici est améliorable.

En effet par construction, nous devons partager des données entre l'initialisation et la mise à jour - cf. les variables globales comme `handle1` - et c'est, comme toujours, une pratique qu'on cherche à éviter.

Voici une approche qui va réaliser exactement la même fonction, mais basée sur une classe; on va tirer profit de l'instance pour ranger proprement toutes les données.

```
[10]: # première version d'une classe d'animation

class Animation:

    # la fonction doit être vectorisée
    def display(self, function, title, *,
                y_range=(-5, 5), height=300, width=600):
        self.figure = figure(
            title=title, y_range=y_range,
```



```

        plot_height=height, plot_width=width)
self.x = np.linspace(0, 2*np.pi, 200)
y = function(self.x)
self.courbe = self.figure.line(self.x, y, color="#2222aa", line_width=3)
self.handle = show(self.figure, notebook_handle=True)

# on passe directement la fonction en paramètre
def update(self, function, frequency, amplitude, phase):
    new_y = amplitude * function(frequency * self.x + phase)
    self.courbe.data_source.data['y'] = new_y
    push_notebook(handle=self.handle)

def interact(self):
    # interact nous impose de passer une simple fonction
    # pour passer 'self' à cette fonction on crée une cloture
    def closure(function, frequency, amplitude, phase):
        self.update(function, frequency, amplitude, phase)
    interact(closure,
            function = Dropdown(
                options=(('sinus', np.sin), ('cosinus', np.cos), ('tangeante',
→np.tan))),
            frequency=(1, 20),
            amplitude=[0.5, 1, 3, 5],
            phase=(0, 2*np.pi, 0.05),
            )

```

```
[11]: a1 = Animation()
a1.display(np.sin, "fonctions trigonométriques")
```

```
[12]: a1.interact()
```

```
interactive(children=(Dropdown(description='function', options=(('sinus', <ufunc 'sin'>), ('cosinus',
```

Remarque

Je vous recommande cette pratique car, à nouveau, cela permet d'éviter les variables globales qui sont toujours une mauvaise idée ; tous les morceaux interdépendants sont regroupés, ainsi on limite la possibilité de casser le code en ne modifiant qu'un morceau ; la classe matérialise les interdépendances entre les objets `figure`, `handle` et `courbe` ; remarquez qu'en fait on n'a pas strictement besoin de `self.figure` comme attribut de l'instance.

Exemple : distribution uniforme

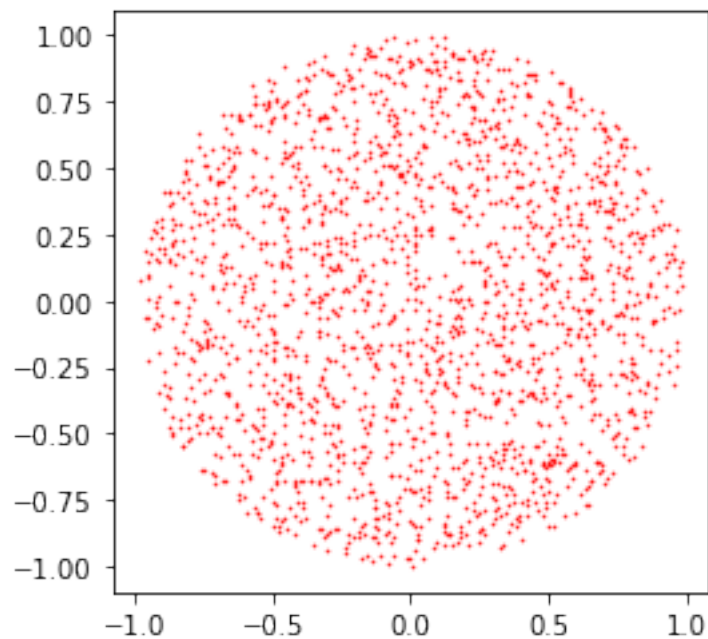
Voyons un deuxième exemple avec `bokeh`. Vous pouvez prendre ceci comme un exercice, et le faire de votre côté avant de lire la suite du notebook.

On veut ici écrire un outil qui déplace et déforme une distribution de points ; on part d'une distribution de N points calculée aléatoirement une bonne fois au début dans le cercle unité ; grâce aux réglages on pourra déformer ce nuage de points, qui va devenir une ellipse, grâce aux réglages suivants :

- `dx` et `dy`, les coordonnées du centre de l'ellipse,
- `rx` et `ry` les rayons en x et en y de l'ellipse,
- et enfin `alpha` l'angle de rotation de l'ellipse.

```
[13]: # petit utilitaire pour calculer la distribution
# uniforme de départ
def uniform_distribution(N):
    # on tire au hasard un rho et un rayon
    rhos = 2 * np.pi * np.random.sample(N)
    rads = np.random.sample(N)
    # il faut prendre la racine carrée du rayon
    # sinon ce n'est pas uniforme dans le plan
    circle_x = np.sqrt(rads) * np.cos(rhos)
    circle_y = np.sqrt(rads) * np.sin(rhos)
    return circle_x, circle_y

[14]: # regardons ça rapidement, - avec matplotlib
# pour vérifier que la répartition est bien homogène
import matplotlib.pyplot as plt
plt.figure(figsize=(4, 4))
X, Y = uniform_distribution(2000)
plt.scatter(X, Y, marker='.', s=1, color='red');
```



un peu de variété

```
[15]: # et aussi: pour que ce soit plus joli
# et surtout plus facile à suivre visuellement
# je tire au hasard des couleurs
# et des tailles pour les points
def enhanced_uniform_distribution(N):
    # on calcule la distribution initiale
    # (celle-ci est vraiment uniforme)
    # dans le cercle de rayon 1
    x, y = uniform_distribution(N)
```

```

# le rouge entre 50 et 250
reds = 50 + 200 * np.random.random(size=N)
# le vert entre 30 et 250
greens = 30 + 220 * np.random.random(size=N)
# la mise en forme des couleurs
# le bleu est constant à 150
colors = [
    f"#{int(red):02x}-{int(green):02x}-{150:02x}"
    for red, green in zip(reds, greens)
]

# les rayons des points; entre 0.05 et 0.25
radii = 0.05 + np.random.random(size=N) * .20

return x, y, colors, radii

```

```

[16]: # c'est ici qu'on commence à faire du bokeh

# j'applique la technique qu'on vient de voir
# en créant une classe
# pour éviter les variables globales

class AnimatedDistribution:

    def __init__(self, N):
        self.N = N

    def show(self):
        # les choix des bornes sont très arbitraires
        # dans une version plus élaborée tous ces détails pourraient
        # être passés en paramètre au constructeur
        self.figure = figure(
            title="distribution pseudo-uniforme",
            plot_height=300, plot_width=300,
            x_range=(-10, 10),
            y_range=(-10, 10),
        )

        # on range x0 et y0 dans des attributs de l'instance
        # pour pouvoir faire les mises à jour
        self.x0, self.y0, colors, radii = enhanced_uniform_distribution(self.N)

        # le paquets de cercles
        self.cloud = self.figure.circle(
            self.x0, self.y0,
            radius = radii,
            fill_color=colors, fill_alpha=0.6,
            line_color=None, line_width=.1,
        )

        # et enfin la poignée qui, à nouveau, sera nécessaire
        # pour les mises à jour
        self.handle = show(self.figure, notebook_handle=True)

    def update(self, rx, ry, dx, dy, alpha):

```

```

    # on recalcule les x et y
    # à partir des valeurs initiales
    s, c = np.sin(alpha), np.cos(alpha)
    x = dx + c * rx * self.x0 - s * ry * self.y0
    y = dy + s * rx * self.x0 + c * ry * self.y0
    self.cloud.data_source.data['x'] = x
    self.cloud.data_source.data['y'] = y
    push_notebook(handle=self.handle)

def interact(self):
    def closure(rx, ry, dx, dy, alpha):
        self.update(rx, ry, dx, dy, alpha)
    interact(closure,
            rx=FloatSlider(min=.5, max=8,
                           step=.1, value=1.),
            ry=FloatSlider(min=.5, max=8,
                           step=.1, value=1.),
            dx=(-3, +3, .2),
            dy=(-3, +3, .2),
            alpha=FloatSlider(min=0., max=np.pi,
                              step=.05, value=0.))

```

```
[17]: dist = AnimatedDistribution(1000)
      dist.show()
```

```
[18]: # pour déformer / déplacer
      dist.interact()
```

```
interactive(children=(FloatSlider(value=1.0, description='rx', max=8.0, min=0.5), FloatSlider(value=
```

le point étant ici de montrer que toutes les modifications sont lisses, sans l'effet de flickering qu'on obtiendrait en redessinant toute l'image à chaque fois

Autres bibliothèques

Pour terminer cette digression sur les solutions alternatives à `matplotlib`, j'aimerais vous signaler enfin rapidement quelques autres options disponibles actuellement.

Comme on l'a dit en introduction, l'offre dans ce domaine est pléthorique, aussi si vous avez un témoignage à apporter sur une expérience que vous avez eue dans ce domaine, nous serons ravis de vous voir la partager dans le forum du cours.

plotly [la bibliothèque plotly](#).

Cette bibliothèque est disponible en open source, et l'offre commerciale de plotly est tournée vers le conseil autour de cette technologie. Comme pour `bokeh`, elle est conçue comme un hybride entre Python et JavaScript, au dessus de `d3.js`. En réalité, elle présente même la particularité d'offrir une API unique disponible depuis Python, JavaScript, et R.

mpld3 <https://mpld3.github.io/>

Je n'ai pas d'expérience à partager avec cette librairie, mais sur la papier l'approche semble prometteuse, puisqu'il s'agit (aussi) de concilier matplotlib avec d3.js.

k3d J'ai utilisé récemment la librairie k3d et j'ai trouvé le résultat assez bluffant pour les visualisations 3d. C'est un outil assez spartiate en termes de documentation, mais très performant.

Cette librairie se prête bien à la technique d'interactions que nous avons développée dans ce notebook.

7.31.3 Complément - niveau avancé (voire oiseux)

Simplement pour finir, j'aimerais revenir sur notre classe `Animation`.

On pourrait même considérer qu'une instance de notre classe `Animation` est une figure, et donc envisager de la faire hériter d'une classe `bokeh.figure`; sauf qu'en fait `bokeh.figure` n'est pas une classe mais une fonction (une factory, c'est-à-dire une fonction qui contruit des instances) :

```
[19]: # l'objet bokeh.figure est une factory, est pas une classe
      # comme on le devine grâce aux minuscules
      type(figure())
```

[19]: function

```
[20]: # la classe c'est celle-ci:
      type(figure())
```

[20]: bokeh.plotting.figure.Figure

```
[21]: # qu'on peut importer comme ceci
      from bokeh.plotting import Figure

      type(figure()) is Figure
```

[21]: True

Exercice (niveau avancé) :

vous semble-t-il possible de récrire la classe `Animation` comme une classe qui hérite cette fois de `Figure` ; quels seraient les bénéfices de cette approche ?

7.32 w7-s10-c6-fourier-k3d

Application à la transformée de Fourier

7.32.1 Complément - niveau avancé

On va appliquer ce qu'on a appris jusqu'ici, au cas de la transformée de Fourier.

Mon angle c'est d'essayer de vous faire intuituer à quoi correspond cette fameuse formule, dans le cas d'une fonction périodique en tous cas, et pourquoi dans ce cas-là on trouve un résultat non nul seulement sur les fréquences harmoniques de la fonction de base.

En guise de bonus, on va en profiter pour représenter aussi la fonction complexe en 3D, c'est surtout un prétexte pour faire au moins un exemple avec `k3d`, qui est très efficace, et qui à mon humble avis gagne à être connue.

Mais commençons par importer ce qui va nous servir.

```
[1]: import numpy as np
      # mostly we use bokeh in here, but the first glimpse is made with mpl
      import matplotlib.pyplot as plt
      %matplotlib notebook
```

```
[2]: from bokeh.plotting import figure, show
      from bokeh.io import push_notebook, output_notebook

      output_notebook()
```

```
[3]: # install with - unsurprisingly (from the terminal)
      # pip install ipywidgets

      from ipywidgets import interact, fixed
      from ipywidgets import SelectionSlider, IntSlider
```

```
[4]: # ditto w/
      # pip install k3d

      import k3d
      from k3d.plot import Plot
```

7.32.2 Une fonction périodique

On considère donc une fonction périodique, comme celle-ci :

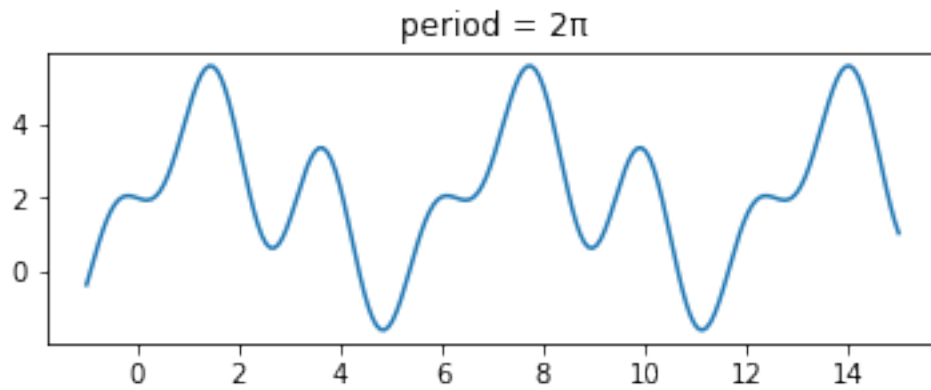
```
[5]: # a vectorized function is required here

      def my_periodic_2pi(t):
          '2sin(x) + sin(2x) - 3/2 sin(3x) + 2'
          return 2*np.sin(t) + np.sin(2*t) - 1.5*np.sin(3*t) + 2
```

Pour un aperçu, on la plotte rapidement avec `matplotlib`

```
[6]: def plot_functions(domain, title, *functions):
      plt.figure(figsize=(6, 2))
      for function in functions:
          plt.plot(domain, function(domain))
          # notice how to retrieve the function's docstring
          plt.title(title)
          plt.show()

      # period is 2 pi, let us plot between 0 and 15 with a .001 step
      plot_functions(np.linspace(-1, 15, 200), "period = 2 ", my_periodic_2pi)
```



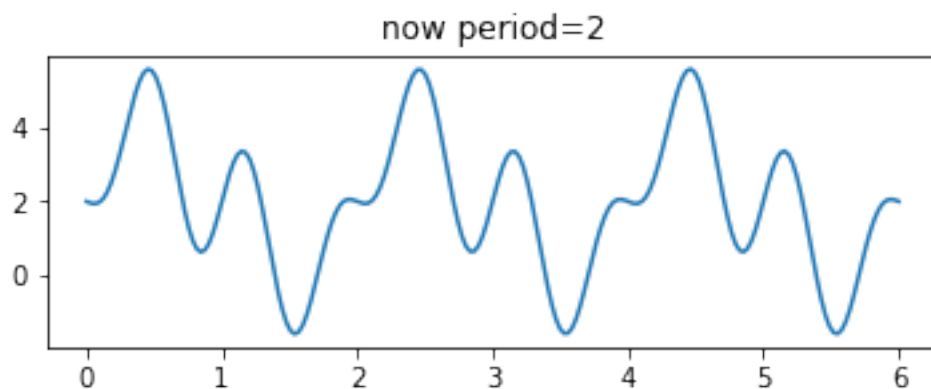
changement d'échelle

Comme on le voit, la période est de 2 , évidemment ;
pour nous simplifier la vie nous allons changer l'échelle des x , pour travailler avec une période entière,
ce sera plus facile pour faire les calculs mentalement ;
je choisis arbitrairement une période = 2 :

```
[7]: # this one has a period of 2
def my_periodic(t):
    "addition of 3 sinus - period = 2"
    return my_periodic_2pi(t*np.pi)
```

```
[8]: D1 = np.linspace(0, 6, 200)

plot_functions(D1, "now period=2", my_periodic)
```



les morceaux

En ignorant la constante additive 2, on sait donc que notre fonction d'entrée est la superposition des 3 fonctions

```
[9]: def H1(t):
      "fundamental"
      return 2*np.sin(t*np.pi)
    def H2(t):
      "fundamental"
      return np.sin(2*t*np.pi)
    def H3(t):
      "fundamental"
      return -1.5*np.sin(3*t*np.pi)

    plot_functions(D1, "the 3 pieces", H1, H2, H3)
```



La transformée de Fourier permet de retrouver ces 3 morceaux, donc par construction de `my_periodic` on doit retrouver :

- la fondamentale (bleu) : period = 2, frequency = 1/2
- l'harmonique de 2nd ordre (orange) : period = 1, frequency = 1
- l'harmonique de 3eme ordre (vert) : period = 2/3, frequency = 3/2

les plages de fréquence

Du coup on va avoir envie de s'intéresser à plusieurs plages de fréquence :

```
[10]: from ipywidgets import FloatSlider, Dropdown, Layout
      # for building sliders
      full_width = Layout(width='100%')

      # la fréquence fondamentale
      FUNDAMENTAL = 1/2

      # un widget à large spectre, pour choisir une fréquence entre 1/4 et 3
      def full_spectrum():
          return FloatSlider(min=0.25, max=3., step=0.01,
                             layout=full_width,
                             value=FUNDAMENTAL,
                             )

      # quand on voudra faire un zoom autour d'une fréquence précise
      def closeup_around(freq):
```



```

return FloatSlider(min=freq * 0.98, max=freq * 1.02,
                   # 400 steps
                   step = freq/10_000,
                   layout=full_width,
                   value=FUNDAMENTAL,
                   readout_format='.4f',
                   )

```

7.32.3 La formule de fourier

Je rappelle la formule magique, la transformée de Fourier de f est la fonction F qui associe à une fréquence ϕ la valeur :

$$F : \phi \rightarrow \int_{-\infty}^{\infty} f(t) e^{2i\pi\phi t} dt$$

Pour un ϕ donné, il s'agit donc de calculer l'intégrale sur \mathbb{R} de la fonction complexe

$$F_{\phi}(t) = f(t) e^{2i\pi\phi t}$$

On va commencer par représenter cette courbe en 3D : * sur l'axe des x, on représente le temps t * et sur les axes y et z, on représente les partie réelle et imaginaire de $F_{\phi}(t)$

7.32.4 représentation 3D

Voici une classe permettant de visualiser la courbe de F_{ϕ} en 3D.

En plus de la courbe, on matérialise par une ligne rouge l'emplacement du barycentre de la distribution complexe (plus de détails plus bas).

Une différence par rapport à ce qu'on avait pu voir avec `bokeh`, c'est qu'ici la librairie `k3d` expose une classe `Plot`, qui peut s'afficher directement dans le notebook ; du coup il semble raisonnable ici d'hériter de cette classe ; sinon l'idée générale est la même.

Vous avez sans doute déjà remarqué que chaque librairie de visualisation s'attend à recevoir les données d'entrée sous un format spécifique - ce qui a tendance à rendre l'utilisation de toutes ces techniques un peu fastidieuse parfois..

En tous cas notez que de manière opportuniste, la méthode centrale ici, à savoir `compute_dots_and_center()`, retourne ses données sous un format qui est propice pour `k3d` - qui aime les tableaux de `shape (n,3)`, d'où l'appel à `np.stack()`.

```

[11]: class FourierAnimator3D(Plot):

    DOTS_PER_UNIT = 50

    def __init__(self, function, phi,
                 domain=10, **kwds):
        self.function = function
        self.phi = phi
        self.domain = domain
        # pass along named parameters, like e.g. height
        super().__init__(**kwds)

        # returns the format expected by k3d line
        dots, center = self.compute_dots_and_center()
        # create line and add in plot

```

```

        self.line = k3d.line(dots)
        self += self.line
        # the line that materializes the barycenter
        self.center_line = k3d.line(center, color=0xff0000, width=0.5)
        self += self.center_line

    def update(self, phi):
        self.phi = phi
        new_dots, new_center = self.compute_dots_and_center()
        self.line.vertices = new_dots
        self.center_line.vertices = new_center

    def compute_dots_and_center(self):
        """
        returns an array of shape (nb_points, 3) suitable for k3d.line
        """
        nb_points = self.DOTS_PER_UNIT * self.domain
        t, dt = np.linspace(0, self.domain, nb_points, retstep=True)
        # a complex value
        rotating = self.function(t) * np.exp(2j * np.pi * self.phi * t)
        x = t
        y = np.real(rotating)
        z = np.imag(rotating)
        # the format expected by k3d line
        dots = np.stack([x, y, z], axis=1)
        # compute barycenter - as a complex average
        center_complex = np.sum(rotating) / nb_points
        # the format expected by k3d points
        # here 1 point at each end of the cylinder
        center = np.array([(0, center_complex.real, center_complex.imag),
                          (self.domain, center_complex.real, center_complex.imag)])
        return dots, center

    def interact(self, phi_widget):
        interact(lambda phi: self.update(phi), phi=phi_widget)

```

```

[12]: a3d = FourierAnimator3D(my_periodic, phi=1.)
      display(a3d)
      a3d.interact(full_spectrum())

```

```

/Users/tparment/miniconda3/envs/flotpython-course/lib/python3.7/site-packag
es/traitlets/traitlets.py:101: UserWarning: Given trait value dtype "f
loat64" does not match required type "float32". A coerced copy has been
created.
  np.dtype(self.dtype).name))

```

```

FourierAnimator3D(antialias=3, axes=['x', 'y', 'z'], axes_helper=1.0, background_color=16777215, cam

```

```

interactive(children=(FloatSlider(value=0.5, description='phi', layout=Layout(width='100%'), max=3.0

```

calculer et visualiser l'intégrale : le barycentre

Rappelez-vous que : * on commence par fixer ϕ ;
 et ϕ correspond à la vitesse de rotation de la courbe de f autour de l'axe $y=z=0$

Et là vous vous dites, c'est bien joli mais comment je calcule l'intégrale de cette fonction complexe ?

En fait c'est assez simple à faire mentalement, et ça le truc crucial à comprendre, c'est que cette intégrale se déduit du barycentre de la courbe qu'on observe si on se met "au bout" de l'axe du temps et qu'on observe le signal tourner.

Intuitivement, pour évaluer l'intégrale d'une fonction usuelle, on peut estimer la moyenne de f entre les bornes, on n'a plus qu'à multiplier par la longueur du segment.

De la même façon, le barycentre de ce dessin - à nouveau quand on regarde le long de l'axe du temps - donne une bonne indication de la valeur de l'intégrale ; bien sûr, pour obtenir les coordonnées du barycentre il faut normaliser (diviser par la longueur du segment, comme pour la moyenne en dimension 1) ; aussi si on a N points dans notre échantillon :

$$\begin{aligned} \int_a^b F_\phi(t) dt &\approx \sum \text{rotating}[i] * dt \\ &\approx \sum \frac{\text{rotating}[i] * (b-a)}{N} \end{aligned}$$

et du coup le barycentre, qui est obtenu (souvenez-vous le cas de la dimension 1) en divisant cette valeur par la longueur du domaine $(b-a)$ s'obtient par notre unique ligne de code

```
center_complex = np.sum(rotating) / nb_points
```

On peut essayer de faire le calcul mentalement, mais c'est parfois délicat ; d'une part on ne voit pas la différence entre les points où f est positive ou négative ; d'autre part il faut noter que ça ne marche bien en fait que parce la vitesse de rotation est uniforme.

En tous cas le barycentre dans la visualisation donne, lui, une indication fiable de la valeur de l'intégrale.

Ce qu'on observe sur cette première visualisation, - on va le voir encore mieux en 2D - c'est que le barycentre est souvent nul, sauf au voisinage des fameuse fréquences de f - ouf, ça marche !

7.32.5 animation en 2D

Il ne nous reste plus qu'à représenter la même chose, mais cette fois en 2D en regardant le long de l'axe des x ; la logique est la même, sauf pour le format de retour de `compute_dots_and_center`, qui est adapté pour `bokeh` :

```
[13]: DEFAULT_RANGE = (-6, 6)
      DEFAULT_DOMAIN = 100

      class FourierAnimator2D:

          DOTS_PER_UNIT = 50

          def __init__(self, function, phi, domain=DEFAULT_DOMAIN, **kwargs):
              self.function = function
              self.phi = phi
              self.domain = domain

          def compute_dots_and_center(self):
              """
              returns X, Y for the curve in 2D
              and xc, yc the coordinates of the (bary)center
              """
              nb_points = self.DOTS_PER_UNIT * self.domain
              t, dt = np.linspace(0, self.domain, nb_points, retstep=True)
              # a complex value
```

```

    rotating = self.function(t) * np.exp(2j * np.pi * self.phi * t)
    X = np.real(rotating)
    Y = np.imag(rotating)
    # compute barycenter - as a complex average
    center_complex = np.sum(rotating) / nb_points
    return X, Y, center_complex.real, center_complex.imag

def display(self, x_range=DEFAULT_RANGE, y_range=DEFAULT_RANGE):
    self.figure = figure(
        title=self.function.__name__,
        x_range=x_range, y_range=y_range)

    X, Y, xc, yc = self.compute_dots_and_center()

    self.courbe = self.figure.line(X, Y, color='blue', line_width = 1)
    self.center = self.figure.circle([xc], [yc], size=5, color="red")
    self.handle = show(self.figure, notebook_handle=True)

def update(self, phi):
    self.phi = phi

    X, Y, xc, yc = self.compute_dots_and_center()
    self.courbe.data_source.data['x'] = X
    self.courbe.data_source.data['y'] = Y
    self.center.data_source.data['x'] = [xc]
    self.center.data_source.data['y'] = [yc]
    push_notebook(handle=self.handle)

def interact(self, phi_widget):
    interact(lambda phi: self.update(phi), phi=phi_widget)

```

```

[14]: a2d = FourierAnimator2D(my_periodic, FUNDAMENTAL)
      a2d.display()
      a2d.interact(full_spectrum())

```

```
interactive(children=(FloatSlider(value=0.5, description='phi', layout=Layout(width='100%')), max=3.0
```

On peut même zoomer autour des fréquences critiques :

```

[15]: a2d.interact(closeup_around(FUNDAMENTAL))

```

```
interactive(children=(FloatSlider(value=0.5, description='phi', layout=Layout(width='100%')), max=0.5
```

Discussion

Vous observez la forte discontinuité de F qui vaut 0 presque partout ; vous pouvez comprendre que lorsque la fréquence ϕ n'est pas en résonance avec celle de f , le dessin qu'on obtient est presque parfaitement centré sur $(0, 0)$ et que donc le barycentre est nul.

En mode zoom autour de la fondamentale, on observe mieux la mise en résonance ; par contre cette visualisation peut donner l'illusion que F est continue, ce n'est pas le cas, c'est un artefact lié à la longueur finie de notre domaine.

On a choisi pour la 2D un domaine par défaut qui est $[0..100]$, ce qui fait donc 50 périodes. C'est pour cela qu'on a l'illusion qu'au voisinage d'une fréquence sensible, le barycentre s'écarte petit à petit ; en fait ce n'est pas le cas, c'est réellement une fonction discontinue, mais pour le voir il faut faire le calcul sur un domaine plus long ; lorsque vous choisissez par exemple $\phi = 0.501$, vous voyez seulement les 50 premiers pas de la figure qui commencent à diverger ; vous pouvez imaginer qu'en augmentant le domaine, on verra une décroissance plus rapide.

Par contre ce sont les vitesses de calcul qui vont commencer à nous limiter :

```
[16]: # la discontinuité est plus forte qu'on ne pourrait le penser
# mais pour le voir il faut augmenter le domaine
# et donc les calculs sont plus lents
a2dzoom = FourierAnimator2D(my_periodic, FUNDAMENTAL, domain=500)
a2dzoom.display()
a2dzoom.interact(closeup_around(FUNDAMENTAL))
```

```
interactive(children=(FloatSlider(value=0.5, description='phi', layout=Layout(width='100%'), max=0.5
```

7.32.6 voir aussi

Une vidéo de 3BlueBrown, sur le même sujet ; bon ses animations sont autrement plus sophistiquées :-)

mais ici au moins on les a faites nous-mêmes ;)

<https://www.youtube.com/watch?v=spUNpyF58BY>

7.33 w7-s10-x1-taylor

Le théorème de Taylor illustré

7.33.1 exercice : niveau avancé

En guise d'application de ce qu'on a vu jusqu'ici, je vous invite à réaliser une visualisation du théorème de Taylor ; je vous renvoie à votre cours d'analyse, [ou à wikipedia](#) pour une présentation détaillée de ce théorème, mais ce que nous en retenons se résume à ceci.

On peut approximer une fonction "suffisamment régulière" - disons C^∞ pour fixer les idées - par un polynôme d'ordre n , dont les coefficients dépendent uniquement des n dérivées successives au voisinage d'un point :

$$f_n(x) = \sum_{i=0}^n \frac{f^{(i)}(0) \cdot x^i}{i!}$$

Sans perte de généralité nous avons ici fixé le point de référence comme étant égal à 0, il suffit de translater f par changement de variable pour se ramener à ce cas-là.

Le théorème de Taylor nous dit que la suite de fonctions (f_n) converge vers f .

On pourrait penser - c'était mon cas la première fois que j'ai entendu parler de ce théorème - que l'approximation est valable au voisinage de 0 seulement ; si on pense en particulier à sinus, on peut accepter l'idée que ça va nous donner une période autour de 0 peut-être.

En fait, c'est réellement bluffant de voir que ça marche vraiment incroyablement bien et loin.

7.33.2 mon implémentation

Je commence par vous montrer seulement le résultat de l'implémentation que j'ai faite.

Pour calculer les dérivées successives j'utilise la librairie `autograd`.

Ce code est relativement générique, vous pouvez visualiser l'approximation de Taylor avec une fonction que vous passez en paramètre - qui doit avoir tout de même la bonne propriété d'être vectorisée, et d'utiliser la couche `numpy` exposée par `autograd` :

```
[1]: # to compute derivatives
import autograd
import autograd.numpy as np
```

Sinon pour les autres dépendances, j'ai utilisé les `ipywidgets` et `bokeh`

```
[2]: from math import factorial

from ipywidgets import interact, IntSlider, Layout
```

```
[3]: from bokeh.plotting import figure, show
from bokeh.io import push_notebook, output_notebook

output_notebook()
```

la classe `Taylor`

J'ai défini une classe `Taylor`, je ne vous montre pas encore le code, je vais vous inviter à en écrire une vous même; nous allons voir tout de suite comment l'utiliser, mais pour la voir fonctionner il vous faut l'évaluer :

↓↓↓↓↓ ↓↓↓↓↓ assurez-vous de bien évaluer la cellule cachée ici ↓↓↓↓↓ ↓↓↓↓↓

```
[4]: # @BEG@ name=taylor
class Taylor:
    """
    provides an animated view of Taylor approximation
    where one can change the degree interactively

    derivatives are computed on X=0, translate as needed
    """
    def __init__(self, function, domain):
        """
        initialized from

        Parameters:
            function: an autograd-friendly vectorized function
            domain: the X domain (typically a linspace)
        """
        self.function = function
        self.domain = domain
        self.regular_y = function(self.domain)

    def display(self, y_range):
        """
```

```

        create initial drawing with degree=0

Parameters:
    y_range: there is a need to display all degrees with a fixed
        range on the y-axis for smooth transitions;
        pass this as a (ymin, ymax) tuple
    """
x_range = (self.domain[0], self.domain[-1])
self.figure = figure(title=self.function.__name__,
                      x_range=x_range, y_range=y_range)
# each of the 2 curves is a bokeh line object
self.line_exact = self.figure.line(
    self.domain, self.regular_y, color='green')
self.line_approx = self.figure.line(
    self.domain, self._approximated(0), color='red', line_width=2)
# that's what allows for smooth updates down the road
self.handle = show(self.figure, notebook_handle=True)
# @END@

# @BEG@ name=taylor continued=true

def _approximated(self, degree):
    """
    Computes and returns the Y array, the images of the domain
    through Taylor approximation

    Parameters:
        degree: the degree for Taylor approximation
    """
    # initialize with a constant f(0)
    # 0 * self.domain allows to create an array
    # with the right length
    result = 0 * self.domain + self.function(0.)
    # f'
    derivative = autograd.grad(self.function)
    for n in range(1, degree+1):
        # the term in f(n)(x)/n!
        result += derivative(0.)/factorial(n) * self.domain**n
        # higher-orders derivatives
        derivative = autograd.grad(derivative)
    return result

def _update(self, degree):
    # update the second curve only, of course
    self.line_approx.data_source.data['y'] = self._approximated(degree)
    push_notebook(handle=self.handle)

def interact(self, degree_widget):
    """
    displays a widget for iteratively modifying degree

    Parameters:
        degree_widget: a ipywidget, typically an IntSlider
        styled at your convenience
    """
    interact(lambda degree: self._update(degree), degree=degree_widget)
# @END@

```

↑↑↑↑↑ ↑↑↑↑↑ assurez-vous de bien évaluer la cellule cachée ici ↑↑↑↑↑ ↑↑↑↑↑

```
[5]: # check the code was properly loaded
help(Taylor)
```

Help on class Taylor in module __main__:

```
class Taylor(builtins.object)
|   Taylor(function, domain)
|
|   provides an animated view of Taylor approximation
|   where one can change the degree interactively
|
|   derivatives are computed on X=0, translate as needed
|
|   Methods defined here:
|
|   __init__(self, function, domain)
|       initialized from
|
|       Parameters:
|       function: an autograd-friendly vectorized function
|       domain: the X domain (typically a linspace)
|
|   display(self, y_range)
|       create initial drawing with degree=0
|
|       Parameters:
|       y_range: there is a need to display all degrees with a fixed
|       range on the y-axis for smooth transitions;
|       pass this as a (ymin, ymax) tuple
|
|   interact(self, degree_widget)
|       displays a widget for iteratively modifying degree
|
|       Parameters:
|       degree_widget: a ipywidget, typically an IntSlider
|       styled at your convenience
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
```

sinus

Ma classe Taylor s'utilise comme ceci : d'abord on crée une instance à partir d'une fonction et d'un domaine, i.e. l'intervalle des X qui nous intéresse.

```
[6]: # between -4 and 4
DOMAIN = np.linspace(-4*np.pi, 4*np.pi, 250)
```



```
# an instance
sinus_animator = Taylor(np.sin, DOMAIN)
```

Remarquez bien qu'ici la fonction que je passe au constructeur est en réalité `autograd.numpy.sin` et non pas `numpy.sin`, vu la façon dont on a défini notre symbole `np` lors des imports (et ça ne marcherait pas du tout avec `numpy.sin`).

Ensuite on crée un `ipywidget` qui va nous permettre de choisir le degré n ; dans le cas de sinus, qui est impaire, les degrés intéressants sont impairs (vous pouvez vérifier que les coefficients de Taylor pairs sont nuls lorsque f est impaire).

```
[7]: # the widget to select a degree
sinus_widget = IntSlider(
    min=1, max=33, step=2,          # sinus being odd we skip even degrees
    layout=Layout(width='100%')) # more convenient with the whole page width
```

Pour lancer l'interaction, on n'a plus qu'à :

- afficher le diagramme avec la méthode `display()` ; on a besoin pour cela de préciser les bornes en Y , qui resteront constantes au cours de l'animation (sinon la visualisation est vilaine)

puis lancer l'interaction en passant en paramètre le widget qui choisit le degré, ce qui donne :

```
[8]: # fixed limits in Y
sinus_animator.display((-1.5, 1.5))

sinus_animator.interact(sinus_widget)
```

```
interactive(children=(IntSlider(value=1, description='degree', layout=Layout(width='100%'), max=33,
```

cosinus

La même chose avec cosinus nous donnerait ceci :

```
[9]: # allows to select a degree
sinus_widget = IntSlider(
    min=0, max=34, step=2,          # only even degrees
    layout=Layout(width='100%'))

### ready to go
sinus_animator = Taylor(np.cos, DOMAIN)
sinus_animator.display((-1.5, 1.5))
sinus_animator.interact(sinus_widget)
```

```
interactive(children=(IntSlider(value=0, description='degree', layout=Layout(width='100%'), max=34,
```

exponentielle

```
[10]: # allows to select a degree
exp_widget = IntSlider(min=0, max=17,
    layout=Layout(width='100%'))

### ready to go
exp_animator = Taylor(np.exp, np.linspace(-5, 10, 200))
exp_animator.display((-15_000, 25000))
exp_animator.interact(exp_widget)
```

```
interactive(children=(IntSlider(value=0, description='degree', layout=Layout(width='100%'), max=17),
```

7.33.3 quelques indices

affichage

Ici j'ai utilisé `bokeh`, mais on peut tout à fait arriver à quelque chose de similaire avec `matplotlib` sans aucun doute

conception

Ma classe `Taylor` s'inspire très exactement de la technique décrite dans le Complément #6 "Autres bibliothèques de visualisation", et notamment la classe `Animation`, modulo quelques renommages.

calcul de dérivées avec `autograd`

La seule fonction que j'ai utilisée de la bibliothèque `autograd` est `grad` :

```
[11]: from autograd import grad
```

```
[12]: # dans le cas de sinus par exemple
# les dérivées successives en 0 se retrouvent comme ceci
f = np.sin # à nouveau cette fonction est autograd.numpy.sin
f(0.)
```

```
[12]: 0.0
```

```
[13]: # ordre 1
f1 = grad(f)
f1(0.)
```

```
[13]: 1.0
```

```
[14]: # ordre 2
f2 = grad(f1)
f2(0.)
```

[14]: -0.0

7.33.4 votre implémentation

Je vous invite à écrire votre propre implémentation, qui se comporte en gros comme notre classe `Taylor`.

Vous pouvez naturellement simplifier autant que vous le souhaitez, ou modifier la signature comme vous le sentez (pensez alors à modifier aussi la cellule de test).

À titre indicatif ma classe `Taylor` fait environ 30 lignes de code utile, i.e. sans compter les lignes blanches, les docstrings et les commentaires.

```
[15]: # à vous de jouer

class MyTaylor:
    def __init__(self, function, domain):
        ...
    def display(self, y_range):
        # là on veut créer le dessin original, c'est à dire
        # la figure, la courbe de f qui ne changera plus,
        # et la courbe approchée avec disons n=0 (donc y=f(0))
        ...
    def _update(self, n):
        # modifier la courbe approximative avec Taylor à l'ordre n
        # je vous recommande de créer cette méthode privée
        # pour pouvoir l'appeler dans interact()
        ...
    def interact(self, widget):
        # là clairement il va y avoir un appel à
        # interact() de ipywidgets
        print("inactive for now")
```

```
[16]: # testing MyTaylor on cosinus

sinus_widget = IntSlider(
    min=0, max=34, step=2,      # only even degrees
    layout=Layout(width='100%'))

### ready to go
sinus_animator = MyTaylor(np.cos, DOMAIN)
sinus_animator.display((-1.5, 1.5))
sinus_animator.interact(sinus_widget)
```

inactive for now

7.34 w7-s10-x2-coronavirus

Coronavirus

7.34.1 Exercice - niveau intermédiaire

Où on vous invite à visualiser des données liées au coronavirus.

Bon honnêtement à ce stade je devrais m'arrêter là, et vous laisser vous débrouiller complètement :)

Mais juste pour vous donner éventuellement des idées - voici des suggestions sur comment on peut s'y prendre.

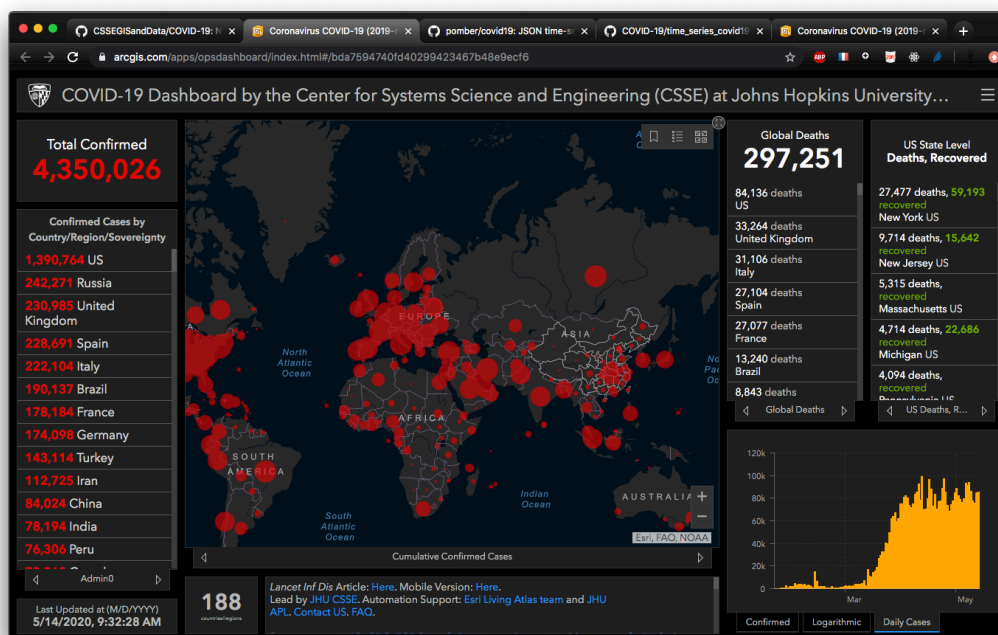
```
[1]: import matplotlib.pyplot as plt
      %matplotlib notebook
```

7.34.2 Le dashboard de Johns Hopkins

Le département Center for Systems Science and Engineering (CSSE), de l'Université Johns Hopkins, publie dans un dépôt github <https://github.com/CSSEGISandData/COVID-19> les données dans un format assez brut. C'est très détaillé et touffu :

```
[2]: # le repo github
      official_url = "https://github.com/CSSEGISandData/COVID-19"
```

Le README mentionne aussi un dashboard, qui permet de visualiser les données en question. Il me semble que l'URL change tous les jours au fur et à mesure des updates, mais voici une capture d'écran pour donner une idée :



Ce qu'on vous propose de faire, pour s'amuser, c'est quelque chose d'analogue - en version beaucoup plus modeste naturellement - pour pouvoir visualiser facilement telle ou telle courbe.

7.34.3 Exercice 1 : un jeu de données intéressant

Pour ma part j'ai préféré utiliser un dépôt de seconde main, qui consolide en fait les données du CSSE, pour les exposer dans un seul fichier au jformat JSON. Cela est disponible dans ce second dépôt github <https://github.com/pomber/covid19>; la sortie de ce processus est mise à jour quotidiennement - à l'heure où j'écris ce texte en Mai 2020 - et est disponible (voir le README) à cette URL <https://pomber.github.io/covid19/timeseries.json>.

```
[3]: abridged_url = "https://pomber.github.io/covid19/timeseries.json"
```

Comme c'est du JSON, on peut charger ces données en mémoire comme ceci

```
[4]: # pour aller chercher l'URL
import requests

# pour charger le JSON en objets Python
import json
```

```
[5]: # allons-y
req = requests.get(abridged_url)
# en utilisant la property `text` on decode en Unicode
encoded = req.text
# que l'on peut decoder
decoded = json.loads(encoded)
```

```
[6]: ## un peu de vérification
# si ceci n'est pas True, il y a un souci
# avec le réseau ou cette URL
req.ok
```

```
[6]: True
```

Les données sont indexées par pays

```
[7]: # voici ce qu'on obtient
type(decoded)
```

```
[7]: dict
```

```
[8]: # une clé
list(decoded.keys())[0]
```

```
[8]: 'Afghanistan'
```

Les données d'un pays sont dans un format très simple, une liste

```
[9]: france_data = decoded['France']
type(france_data)
```

```
[9]: list
```

```
[10]: france_data[0]
```

```
[10]: {'date': '2020-1-22', 'confirmed': 0, 'deaths': 0, 'recovered': 0}
```

```
[11]: france_data[-1]
```

```
[11]: {'date': '2020-5-24', 'confirmed': 182709, 'deaths': 28370, 'recovered': 64735}
```

Homogénéité par pays

Ce que j'ai constaté, et je suppose qu'on peut plus ou moins compter sur cette bonne propriété, c'est que * pour chaque pays on trouve un enregistrement par jour * tous les pays ont la même plage de temps - quitte à rajouter des enregistrements à 0, comme ci-dessus pour la France le 22 janvier

```
[12]: us_data = decoded['US']
      us_data[0]
```

```
[12]: {'date': '2020-1-22', 'confirmed': 1, 'deaths': 0, 'recovered': 0}
```

```
[13]: us_data[-1]
```

```
[13]: {'date': '2020-5-24',
      'confirmed': 1643246,
      'deaths': 97720,
      'recovered': 366736}
```

```
[14]: len(france_data) == len(us_data)
```

```
[14]: True
```

```
[15]: # nombre de pays
      len(decoded)
```

```
[15]: 188
```

```
[16]: # nombre de jours
      len(france_data)
```

```
[16]: 124
```

Un sujet possible (#1)

Vous pourriez interpréter ces données pour créer un dashbord dans lequel on peut choisir : * la donnée à laquelle on s'intéresse (confirmed, deaths ou recovered) * le pays auquel on s'intéresse (idéalement dans une liste triée) * la période (en version simple : les n derniers jours)

et en fonction, afficher deux courbes qui montrent sur cette période * la donnée brute (une fonction croissante donc) * sa dérivée (la différence avec le jour précédent)

Selon votre envie, toutes les variantes sont possibles, pour simplifier (commencez sans dashboard), ou complexifier, comme vous le sentez. Pour revenir sur le dashboard de CSSE, on pourrait penser à utiliser un package comme `folium` pour afficher les résultats sur une carte du Monde ; cela dit je vous recommande de bien réfléchir avant de vous lancer là-dedans, car c'est facile de se perdre, et en plus la valeur ajoutée n'est pas forcément majeure...

un mot par rapport à pandas

Il y a plein d'approches possibles, et toutes raisonnables :

- si vous êtes à l'aise avec **pandas**, vous allez avoir le réflexe de construire immédiatement une grosse dataframe avec toutes ces données, et utiliser la puissance de **pandas** pour faire tous les traitements de type tris, assemblages, moyennes, roulements, etc.. et les affichages (et si vous êtes dans ce cas, vous préférerez l'exercice #2) ;
- si au contraire vous êtes réfractaire à pandas, vous pouvez absolument tout faire sans aucune dataframe ;
- entre ces deux extrêmes, on peut facilement imaginer des hybrides, où on construit des dataframes de manière opportuniste selon les traitements.

La courbe d'apprentissage de pandas est parfois jugée un peu raide ; c'est à vous de voir ce qui vous convient le mieux. Ce qui est clair c'est que quand on maîtrise bien, et une fois qu'on a construit une grosse dataframe avec toutes les données, on dispose avec d'un outil surpuissant pour faire plein de choses en très peu de lignes.

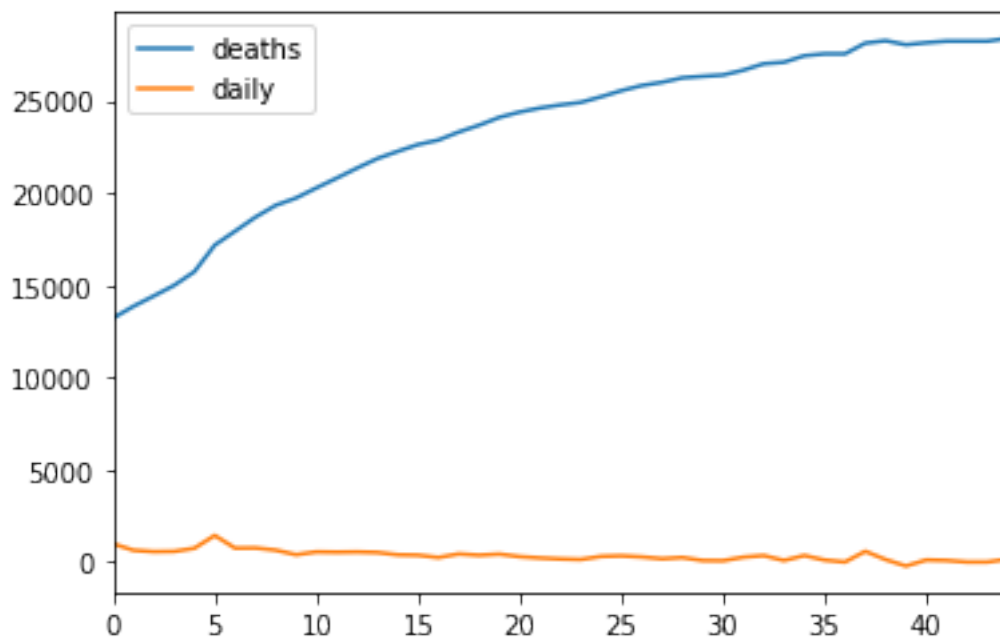
Mais pour bien maîtriser il faut avoir l'occasion de pratiquer fréquemment, ce n'est pas forcément le cas de tout le monde (ce n'est pas le mien par exemple), donc à chacun de choisir son approche.

Pour illustrer une approche disons hybride, voici ce qui pourrait être un début de mise en forme des données pour un pays et une caractéristique (parmi les 3 exposées dans ce jeu de données)

```
[17]: import numpy as np
import pandas as pd

def extract_last_days(countryname, value, days):
    country = decoded[countryname]
    cropped = country[-(days):]
    dates = np.array([chunk['date'] for chunk in cropped])
    # take one more than requested for computing deltas including
    # for the first day (we need the value the day before the first day)
    cropped = country[-(days+1):]
    values = np.array([chunk[value] for chunk in cropped])
    # shift one day so we get the value from the day before
    shifted = np.roll(values, 1)
    # the daily increase; ignore first value which is wrong
    deltas = (values - shifted)[1:]
    relevant = values[1:]
    # all 3 arrays dates, deltas and relevant have the same shape
    data = {'dates': dates, value: relevant, 'daily': deltas}
    return pd.DataFrame(data=data)

df1 = extract_last_days('France', 'deaths', 45)
df1.plot();
```



7.34.4 Exercice 2 : idem mais à partir d'un autre jeu de données

Je vous signale une autre source de données, dans ce repo git <https://github.com/owid/covid-19-data/tree/master/public/data>; les données cette fois-ci sont au format excel, et publiées à cette adresse

```
[18]: alt_url = 'https://covid.ourworldindata.org/data/owid-covid-data.csv'
```

Dans ces cas-là il faut avoir le réflexe d'utiliser pandas; voici un aperçu (ayez de la patience pour le chargement)

```
[19]: import pandas as pd

df = pd.read_csv(alt_url)
```

```
[20]: df.head(2)
```

```
[20]:  iso_code location      date  total_cases  new_cases  total_deaths  \
0      ABW  Aruba  2020-03-13           2           2           0
1      ABW  Aruba  2020-03-20           4           2           0

   new_deaths  total_cases_per_million  new_cases_per_million  \
0           0           18.733           18.733
1           0           37.465           18.733

   total_deaths_per_million  ...  aged_65_older  aged_70_older  \
0              0.0  ...       13.085       7.452
1              0.0  ...       13.085       7.452

   gdp_per_capita  extreme_poverty  cvd_death_rate  diabetes_prevalence  \
```


0	35973.781	NaN	NaN	11.62
1	35973.781	NaN	NaN	11.62

	female_smokers	male_smokers	handwashing_facilities	hospital_beds_per_100k
0	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN

[2 rows x 32 columns]

Un sujet possible (#2)

Le sujet à la base est le même bien entendu, essayer de visualiser ces données sous une forme où on y perçoit quelque chose :)

Les données sont bien entendu beaucoup plus riches, a contrario cela va demander davantage de mise en forme avant de pouvoir visualiser quoi que ce soit.

Je vous propose ce second point de vue si vous souhaitez vous entraîner avec **pandas**, puisqu'ici on a déjà une dataframe (ce qui ne veut pas dire qu'on ne peut pas traiter le premier exercice en utilisant **pandas**).

Explorons un peu

Voici quelques éléments sur la structure de ces données :

```
[21]: # beaucoup plus de détails
df.columns
```

```
[21]: Index(['iso_code', 'location', 'date', 'total_cases', 'new_cases',
            'total_deaths', 'new_deaths', 'total_cases_per_million',
            'new_cases_per_million', 'total_deaths_per_million',
            'new_deaths_per_million', 'total_tests', 'new_tests',
            'total_tests_per_thousand', 'new_tests_per_thousand',
            'new_tests_smoothed', 'new_tests_smoothed_per_thousand', 'tests_units',
            'stringency_index', 'population', 'population_density', 'median_age',
            'aged_65_older', 'aged_70_older', 'gdp_per_capita', 'extreme_poverty',
            'cvd_death_rate', 'diabetes_prevalence', 'female_smokers',
            'male_smokers', 'handwashing_facilities', 'hospital_beds_per_100k'],
            dtype='object')
```

```
[22]: # la colonne iso_code représente le pays :
df.iso_code.unique()[:5]
```

```
[22]: array(['ABW', 'AFG', 'AGO', 'AIA', 'ALB'], dtype=object)
```

```
[23]: # rien que sur la france, on a ce nombre d'enregistrements
df_france = df[df.iso_code == 'FRA']
```

```
len(df_france)
```

[23]: 147

```
[24]: # manifestement c'est un par jour
df_france.head(2)
```

```
[24]:
```

	iso_code	location	date	total_cases	new_cases	total_deaths	\
6236	FRA	France	2019-12-31	0	0	0	
6237	FRA	France	2020-01-01	0	0	0	

	new_deaths	total_cases_per_million	new_cases_per_million	\
6236	0	0.0	0.0	
6237	0	0.0	0.0	

	total_deaths_per_million	...	aged_65_older	aged_70_older	\
6236	0.0	...	19.718	13.079	
6237	0.0	...	19.718	13.079	

	gdp_per_capita	extreme_poverty	cvd_death_rate	diabetes_prevalence	\
6236	38605.671	NaN	86.06	4.77	
6237	38605.671	NaN	86.06	4.77	

	female_smokers	male_smokers	handwashing_facilities	\
6236	30.1	35.6	NaN	
6237	30.1	35.6	NaN	

	hospital_beds_per_100k
6236	5.98
6237	5.98

[2 rows x 32 columns]

```
[25]: df_france.tail(2)
```

```
[25]:
```

	iso_code	location	date	total_cases	new_cases	total_deaths	\
6381	FRA	France	2020-05-24	144806	240	28332	
6382	FRA	France	2020-05-25	144921	115	28367	

	new_deaths	total_cases_per_million	new_cases_per_million	\
6381	43	2218.450	3.677	
6382	35	2220.211	1.762	

	total_deaths_per_million	...	aged_65_older	aged_70_older	\
6381	434.050	...	19.718	13.079	
6382	434.587	...	19.718	13.079	

	gdp_per_capita	extreme_poverty	cvd_death_rate	diabetes_prevalence	\
6381	38605.671	NaN	86.06	4.77	
6382	38605.671	NaN	86.06	4.77	

```

        female_smokers male_smokers  handwashing_facilities  \
6381             30.1         35.6                    NaN
6382             30.1         35.6                    NaN

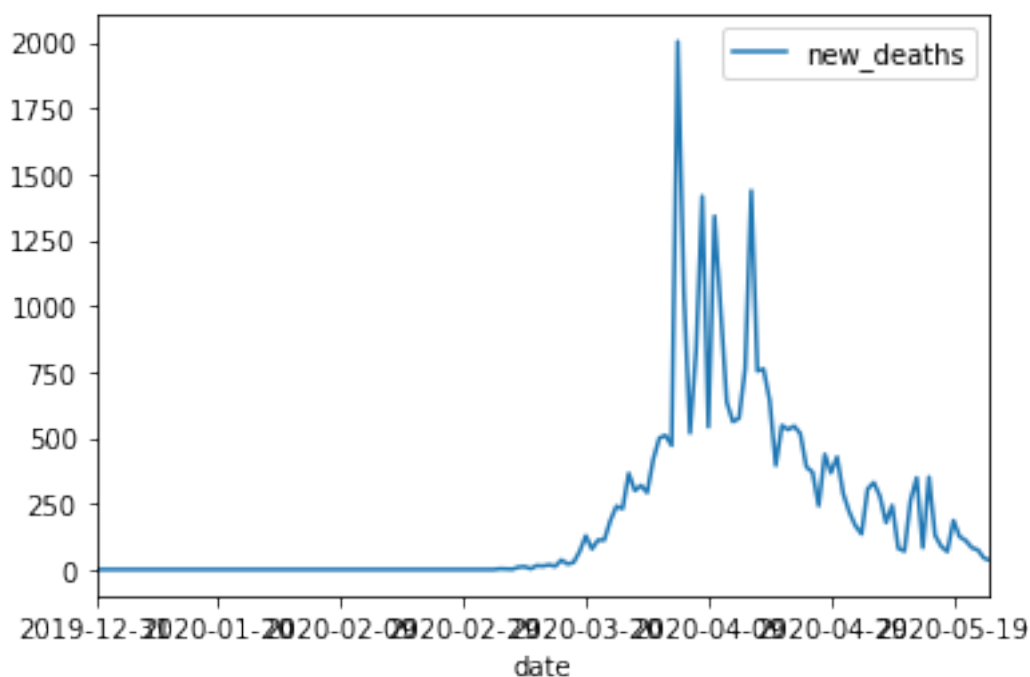
        hospital_beds_per_100k
6381                5.98
6382                5.98

[2 rows x 32 columns]

```

Pour afficher, disons les décès par jour en France depuis le début de la pandémie, on pourrait faire :

```
[26]: df_france.plot(x='date', y='new_deaths');
```



```
[27]: # n'hésitez pas à installer des packages
      # supplémentaires au besoin
      !pip install plotly-express
```

```

Requirement already satisfied: plotly-express in /Users/tparment/miniconda3
  /envs/flotpython-course/lib/python3.7/site-packages (0.4.1)
Requirement already satisfied: pandas>=0.20.0 in /Users/tparment/miniconda3
  /envs/flotpython-course/lib/python3.7/site-packages (from plotly-express
  ) (0.25.0)
Requirement already satisfied: plotly>=4.1.0 in /Users/tparment/miniconda3/
  envs/flotpython-course/lib/python3.7/site-packages (from plotly-express)
  (4.7.1)
Requirement already satisfied: statsmodels>=0.9.0 in /Users/tparment/minico
  nda3/envs/flotpython-course/lib/python3.7/site-packages (from plotly-exp
  ress) (0.11.1)

```

Requirement already satisfied: patsy>=0.5 in /Users/tparment/miniconda3/envs/flotpython-course/lib/python3.7/site-packages (from plotly-express) (0.5.1)

Requirement already satisfied: scipy>=0.18 in /Users/tparment/miniconda3/envs/flotpython-course/lib/python3.7/site-packages (from plotly-express) (1.4.1)

Requirement already satisfied: numpy>=1.11 in /Users/tparment/miniconda3/envs/flotpython-course/lib/python3.7/site-packages (from plotly-express) (1.17.0)

Requirement already satisfied: python-dateutil>=2.6.1 in /Users/tparment/miniconda3/envs/flotpython-course/lib/python3.7/site-packages (from pandas>=0.20.0->plotly-express) (2.8.0)

Requirement already satisfied: pytz>=2017.2 in /Users/tparment/miniconda3/envs/flotpython-course/lib/python3.7/site-packages (from pandas>=0.20.0->plotly-express) (2019.1)

Requirement already satisfied: six in /Users/tparment/miniconda3/envs/flotpython-course/lib/python3.7/site-packages (from plotly>=4.1.0->plotly-express) (1.12.0)

Requirement already satisfied: retrying>=1.3.3 in /Users/tparment/miniconda3/envs/flotpython-course/lib/python3.7/site-packages (from plotly>=4.1.0->plotly-express) (1.3.3)

```
[28]: # plusieurs courbes en une
# avec plotly express, pour changer un peu
import plotly.express as px

selection = ['USA', 'FRA']

start = '2020-03-15'
date_start = start
#date_start = pd.to_datetime(start)
sel = df[(df.iso_code.isin(selection)) & (df.date > date_start)]
fig1 = px.line(sel, x="date", y="total_deaths_per_million", color="location")
fig1.update_layout(height= 800, title_text="Décès Covid, cumulés (par million_
    ↪d'habitants)")
fig1.show()
```

7.34.5 Comment partager ?

Je ne publie pas de corrigés pour cet exercice.

J'invite ceux d'entre vous qui le souhaitent à nous faire passer leur code; le plus simple étant de les ajouter dans le repo github dit de récréation, à cet endroit <https://github.com/flotpython/recreation/tree/master/corona-dashboards>.

Chapitre 8

Programmation asynchrone avec asyncio

8.1 w8-s1-c1-thread-safety

Pourquoi les threads c'est délicat ?

8.1.1 Complément - niveau avancé

À nouveau dans ce cours nous nous intéressons aux applications qui sont plutôt I/O intensive. Cela dit et pour mettre les choses en perspective, on pourrait se dire que qui peut le plus peut le moins, et que le multi-threading qui est bien adapté au calcul parallèle CPU-intensive, pourrait aussi bien faire l'affaire dans le contexte de l'I/O-intensive.

Il se trouve qu'en fait le multi-threading présente un inconvénient assez notable, que nous allons tenter de mettre en évidence dans ce complément ; sur un exemple hyper-simple, nous allons illustrer la notion de section critique, et montrer pourquoi on doit utiliser parfois - trop souvent - la notion de lock ou verrou lorsqu'on utilise des threads.

8.1.2 avertissement : pas que pour Python

Je dois préciser avant d'aller plus loin que pour cette discussion, nous allons oublier le cas spécifique de Python ; les notions que nous abordons tournent autour des relations entre l'OS et les applications, qui sont valables en général.

En fait c'est même pire que ça, et nous verrons les implications pour Python à la fin du complément ; vous avez peut-être déjà entendu parler du GIL, mais on va avoir besoin d'appréhender cette histoire de section critique pour mieux comprendre les tenants et les aboutissements du GIL en Python.

8.1.3 processus et threads

On rappelle que, pour écrire des programmes parallèles, l'Operating System nous offre principalement deux armes :

- les processus
- les threads

Il faut se souvenir que la première fonction de l'OS est justement que plusieurs programmes puissent s'exécuter en même temps, c'est-à-dire partager les ressources physiques de l'ordinateur, et notamment le CPU et la mémoire, sans pouvoir se contaminer l'un l'autre.

Aussi, c'est par construction que deux processus différents se retrouvent dans des espaces totalement étanches, et qu'un processus ne peut pas accéder à la mémoire d'un autre processus.

On peut naturellement utiliser des processus pour faire du calcul parallèle, mais cette contrainte de naissance rend l'exercice fastidieux, surtout lorsque les différents programmes sont très dépendants les uns des autres, car dans ce cas bien sûr ils ont besoin d'échanger voire de partager des données (je m'empresse de préciser qu'il existe des mécanismes pour faire ça - notamment : bibliothèques de mémoire partagée, envoi de messages - mais qui induisent leur propre complexité...).

Par contraste un processus peut contenir plusieurs threads, chacun disposant pour faire court, d'une pile et d'un pointeur de programme - en gros donc, où on en est dans la logique de une exécution séquentielle ; l'intérêt étant que de tous les threads partagent à présent la mémoire du processus ; c'est donc un modèle a priori très attractif pour notre sujet.

8.1.4 le scheduler

Comme ces notions de processus et de threads sont fournies par l'OS, c'est à lui également que revient la responsabilité de les faire tourner ; cela est fait dans le noyau par ce qu'on appelle le scheduler.

Comment ça marche ? dans le détail, c'est un sujet très copieux, il existe une littérature hyper-abondante sur le sujet, et donc une extrême variété de stratégies et de réglages possibles.

Mais pour ce qui nous intéresse, nous n'allons retenir que ces caractéristiques de haut niveau très simples :

- le scheduler maintient une liste de processus et de threads à faire tourner ;
- il décide - à une fréquence assez élevée - de leur donner la main à tour de rôle ;
- simplement il faut bien réaliser qu'à ce stade, ce que manipule le scheduler, c'est essentiellement du code binaire, très proche du processeur, après toutes les phases de compilation et optimisation.

context switches

L'instant où le scheduler décide de suspendre l'exécution d'un processus - ou thread - pour donner la main à un autre, s'appelle un context switch ; on parle de process switch* lorsqu'on passe d'un processus à un autre, et de task switch ou thread switch lorsqu'on passe d'un thread à un autre à l'intérieur d'un processus.

Le point important pour nous, c'est que le scheduler est un morceau de code générique, il fait donc son travail de manière neutre pour tous les processus ou threads, indépendamment du langage par exemple, ou du domaine d'application ; et que le découpage du temps en slots alloués aux différents joueurs se fait bien évidemment sur la base des instructions élémentaires du processeur - ce qu'on appelle les cycles.

On s'intéresse davantage aux threads dans la suite, et nous allons voir que dans ce cas, cela crée parfois de mauvaises surprises.

8.1.5 une simple opération d'addition

Pour illustrer notre propos, nous allons étudier une opération extrêmement banale qui consiste à incrémenter la valeur d'une variable.

Il se trouve qu'en pratique cette opération se décompose en réalité en 3 opérations élémentaires, comme le montre la figure suivante ; à nouveau le langage utilisé dans toutes ces illustrations n'est pas du Python - typiquement une opération comme celle-ci en Python va occasionner bien plus d'instructions élémentaires que cela - disons pour fixer les idées que c'est quelque chose comme du C ; peu importe en fait, c'est l'idée qui est importante.



Figure 1 : une instruction du genre de $a = a + 1$ dans un langage compilé, avec un seul thread

On voit sur cette figure la logique des trois opérations * dans un premier temps on va chercher la valeur de la variable **a** qu'on range disons dans un registre - ou un cache ; * on réalise l'incrémement de cette valeur dans le registre * puis on recopie le résultat dans la case mémoire originelle, qui correspond à la variable **a**

Ce programme fait donc bien ce qu'on veut ; si la valeur de **a** était 10 au début, on y trouve 11 à la fin, tout va bien.

8.1.6 dans deux threads, un scénario favorable

À présent, nous allons imaginer le cas de deux threads qui s'exécutent en parallèle, avec un seul processeur ; et admettons que chacun des deux threads exécute une fois $a = a + 1$ sur une variable globale **a**.

En admettant comme tout à l'heure que **a** valait 10 en commençant, on s'attend donc naturellement à ce qu'à la fin **a** vaille 12 puisqu'on l'aura incrémenté deux fois.

Voyons d'abord un scénario qui se passe bien ; le scheduler qui, donc, donne la main alternativement à l'un et l'autre de nos deux threads, a la bonne idée de laisser intègres les deux blocs de 3 instructions, sans y insérer de context switching.

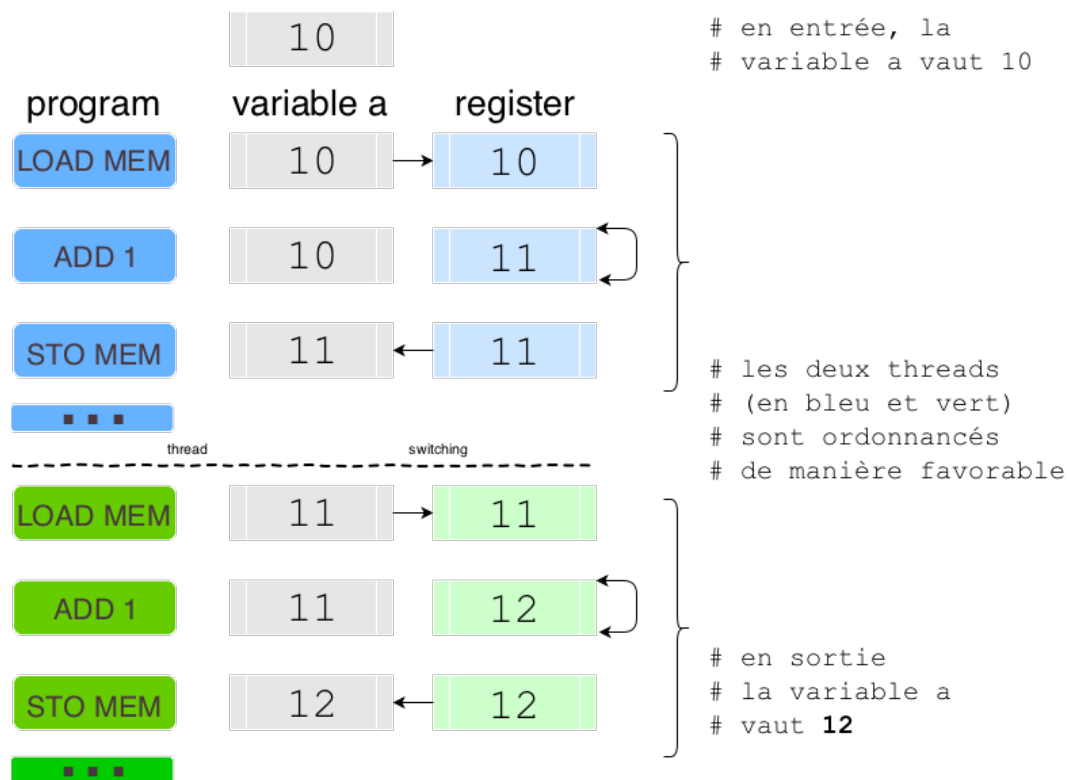


Figure 2 : deux threads, un processeur, dans un scénario favorable

Dans ce scénario, à l'issue des deux threads on a bien, comme attendu, `a == 12`.

8.1.7 toujours 2 threads, mais pas de chance

Mais en fait, il y a un souci avec cette façon de faire.

Ce qu'il faut bien comprendre c'est que du point de vue du scheduler, toutes ces instructions sont pareilles et il n'y a pas, à ce stade, de moyen pour le scheduler, de traiter ce bloc de 3 instructions de manière particulière.

Aussi le scheduler, qui a déjà un travail assez compliqué si on tient compte du fait qu'il doit être fair (donner autant de temps à tout le monde), choisit les points de context switching comme il le peut au milieu de ce qui, pour lui, n'est qu'une longue liste d'instructions.

Imaginons du coup un scénario moins favorable que le précédent, dans lequel le scheduler, pas de chance, choisit de faire un context switching juste après le premier LOAD du premier thread ; ça nous donne alors l'exécution décrite dans cette figure :

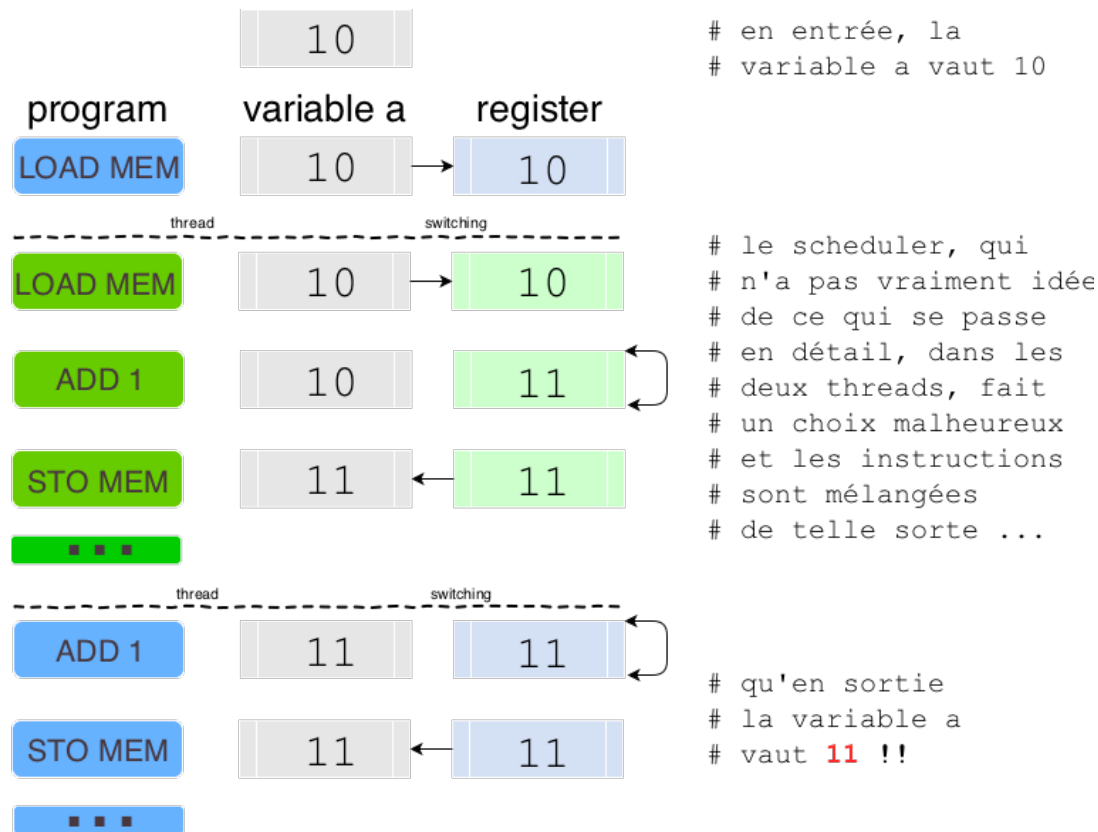


Figure 3 : deux threads, un processeur, mais un choix de scheduling malheureux

Du coup ce qui se passe ici, c'est que le deuxième fil fait son LOAD à partir de la variable `a` qui n'a pas encore été modifiée, et du coup les deux threads incrémentent tous les deux la valeur 10, et à l'issue de l'exécution des deux threads, on a maintenant `a == 11` !!

Pour résumer donc : on part de `a == 10`, on exécute 2 threads qui font tous les deux `a = a + 1` et au final, on se retrouve avec `a == 11` ; gros souci donc !

8.1.8 phénomène général

À ce stade vous pourriez vous dire que j'ai triché, et que j'ai choisi un scénario irréaliste ; par exemple qu'en pratique l'incrément de 1 ça se fait en hardware en une seule instruction.

Oui bien sûr, l'exemple est choisi pour rester aussi simple que possible ; mais si vous n'êtes pas convaincu avec `a = a + 1`, prenez simplement `a = a + b`, vous verrez que c'est exactement le même souci.

En fait le souci que l'on a, de manière générale, c'est que :

- dans un langage de programmation un tout petit peu évolué, un fragment de code (même réduit à une instruction) se traduit presque toujours en plusieurs instructions binaires pour le processeur
- pour que le programme fonctionne correctement dans un mode multi-thread, certains fragments de code, et notamment ceux qui accèdent à de la mémoire partagée, doivent être exécutés de façon atomique (c-à-d ne pas être interrompus en plein milieu par le scheduler)
- et sans aide du programmeur, le scheduler n'a aucun moyen de savoir où, dans le flot d'instruction binaires, il est légitime ou pas de faire un context switching.

Et avec quelque chose d'un tout petit peu plus compliqué comme `a = 2 * a`; `a = a + 1`, on n'a même pas besoin de descendre au niveau du code machine pour exhiber le problème...

8.1.9 verrou et exclusion mutuelle

Du coup, pour rendre la programmation par thread utilisable en pratique, il faut lui adjoindre des mécanismes, accessibles au programmeur, pour rendre explicite ce type de problèmes.

La notion la plus simple de ces mécanismes est celle de verrou pour implémenter une exclusion mutuelle ; pour en donner une illustration très rapide, voyons cela sur notre exemple.

Nous allons remplacer ceci :

```
# thread A
a = a + 1

# thread B
a = a + 1

par ceci

# thread A

get_lock(lock)
a = a + 1
release_lock(lock)

# thread B

get_lock(lock)
a = a + 1
release_lock(lock)
```

Dans cette nouvelle version, un nouvel objet global `lock` est introduit, qui peut être dans deux états libre ou occupé.

De cette façon, celui des deux threads qui arrive à ce stade en premier obtient le verrou (le met dans l'état occupé), et fait son traitement avant de le relâcher ; du coup l'autre doit attendre que le premier ait fini tout le traitement de sa section critique pour pouvoir commencer le sien.

Comme on le voit, l'idée consiste à permettre au programmeur de rendre explicite l'exclusion mutuelle qu'il est nécessaire d'assurer pour que le programme fonctionne comme prévu, et de façon déterministe.

8.1.10 ce qu'il faut retenir

Pour conclure cette partie, retenons que l'on peut écrire du code multi-thread dont le comportement est déterministe, mais au prix de l'ajout dans le code d'annotations qui limitent les modes d'exécution ; ce qui a tendance à rendre les choses complexes, et donc coûteuses.

Et retenons que le problème principal ici est lié à l'absence de contrôle, par le programmeur, sur les context switchings ; et du coup ceux-ci peuvent intervenir à n'importe quel moment.

Nous verrons que la situation est très différente avec le paradigme `async/await/asyncio`.

8.1.11 le cas de Python : le GIL

Dans ce contexte, le cas des programmes Python est un peu spécial ; ce n'est pas un langage compilé, ce qui signifie que du point de vue de l'OS et du scheduler, le processus qui tourne est en fait l'interpréteur Python.

Et il se trouve que l'interpréteur Python est un exemple de programme qui pourrait être sensible au type de problèmes que nous venons d'étudier.

Voyons un exemple pour vous faire entrevoir la complexité du sujet. Vous vous souvenez qu'on a parlé de garbage collection, et de compteur de références. Voyons comment le fait de maintenir un compteur de références crée le besoin d'écrire dans la mémoire, alors qu'en lisant le code Python on ne voit que des accès en lecture.

```
[1]: # on est bien d'accord que ce code ne fait que lire
      # le contenu de x et ne modifie pas sa valeur

      def foo(x, max_depth, depth=1):
          print(f"in {depth}th function call we see {x}")
          if depth < max_depth:
              foo(x, max_depth, depth+1)
```

```
[2]: # j'exécute ce code sur un objet tout neuf
      a = []

      # et je confirme bien qu'on n'y touche jamais
      foo(a, 3)
```

```
in 1th function call we see []
in 2th function call we see []
in 3th function call we see []
```

```
[3]: # mais en fait pendant toute l'exécution de ce code
      # il y a des changements qui sont faits dans l'objet a
      # en tous cas dans sa représentation interne,
      # regardons notamment le compteur de références

      # pour importer getrefcount() qui permet de
      # lire le compteur de références d'un objet
      import sys

      # la même logique exactement que plus haut,
      # mais ici affiche aussi le compteur de références
      def bar(x, max_depth, depth=1):
          print(f"in {depth}th function call we see {x} that has {sys.getrefcount(x)}_↪refs")
          if depth < max_depth:
              bar(x, max_depth, depth+1)

      bar(a, 3)
```

```
in 1th function call we see [] that has 4 refs
in 2th function call we see [] that has 6 refs
in 3th function call we see [] that has 8 refs
```

Du coup, et précisément pour protéger son fonctionnement intime, l'interpréteur Python est implémenté de telle sorte à empêcher l'exécution simultanée de plusieurs threads dans un même processus Python ! Cela est fait au travers d'un verrou central pour tout l'interpréteur, qui s'appelle le GIL - le Global Interpreter Lock.

Aussi, bien qu'il est possible - notamment au travers de la librairie **threading** - de concevoir des programmes multi-threadés en Python, par construction, ils ne peuvent pas s'exécuter en parallèle, et notamment ne peuvent pas tirer profit d'une architecture multi-processeur (pour cela en Python, il ne reste que l'option multi-processus). Ce qui, il faut bien l'admettre, ruine un peu l'intérêt...

8.1.12 pour en savoir plus

Cette présentation est juste une mise en perspective, elle est volontairement superficielle, d'autant que nous sommes clairement à côté de notre sujet. Si vous souhaitez approfondir certains de ces points (malheureusement sur ces sujets pointus les sources anglaises, même sur wikipedia, sont souvent préférables...) :

- sur la notion de thread : [Thread\(computing\) on wikipedia](#)
- sur le mécanisme de verrou :
 - [une illustration en vidéo \(1'17\) du verrou, très proche de notre exemple](#)
 - [wikipedia en anglais](#)
- sur le GIL : <https://realpython.com/python-gil/>

remarque à propos des verrous

Le lecteur attentif remarquera une contradiction apparente, car dans notre présentation des verrous, on a introduit ... un nouvel objet global `lock` ; on pourrait craindre de n'avoir fait ici que de reporter le problème. N'aurait-on pas seulement déplacé le souci qu'on avait avec globale `a` sur la globale `lock` ?

En réalité ça n'est pas le cas, cette approche fonctionne vraiment et est massivement utilisée. Elle fonctionne notamment parce que le mécanisme de verrou est cette fois connu du scheduler, qui par conséquent peut garantir le comportement de `get_lock()` et `release_lock()`.

8.2 w8-s1-c2-warning-python37

Avertissement relatif à **asyncio** et Python-3.7

8.2.1 Complément - niveau intermédiaire

Puisque cette semaine est consacrée à **asyncio**, il faut savoir que cette brique technologique est relativement récente, et qu'elle est du coup, plus que d'autres aspects de Python, sujette à des évolutions.

8.2.2 Les vidéos utilisent Python-3.6

Comme on l'a dit en préambule du cours, notre version de référence est Python-3.6. C'est la version utilisée dans les vidéos. Par contre les notebooks sur FUN-MOOC utilisent à présent la version 3.7.

8.2.3 Un résumé des nouveautés

Vous trouverez à la fin de la semaine, dans la séquence consacrée aux bonnes pratiques, un résumé des améliorations apportées depuis la version 3.6.

8.2.4 L'essentiel est toujours d'actualité

Cela étant dit, nos buts ici étaient principalement :

- de vous faire découvrir ce nouveau paradigme,
- de vous faire sentir dans quelles applications cela peut avoir un apport très précieux,
- de bien vous faire comprendre ce qui se passe à l'exécution,
- et de vous donner un aperçu de la façon dont tout cela est implémenté.

8.2.5 Les différences les plus visibles

Les plus grosses différences concernent la prise en main. Comme nous allons bientôt le voir, le “hello world” de `asyncio` était en Python-3.6 un peu awkward, cela nécessitait pas mal de circonlocutions.

C'est-à-dire que pour faire fonctionner la coroutine :

```
[1]: # un exemple de coroutine
import asyncio

async def hello_world():
    await asyncio.sleep(0.2)
    print("Hello World")
```

En Python-3.6

Pour exécuter cette coroutine dans un interpréteur Python-3.6, la syntaxe est un peu lourdingue :

```
# pour exécuter uniquement cette coroutine en Python-3.6
loop = asyncio.get_event_loop()
loop.run_until_complete(hello_world())
```

En Python-3.7

En 3.7, on arrive au même résultat de manière beaucoup plus simple :

```
# c'est beaucoup plus simple en 3.7
asyncio.run(hello_world())
```

Avec IPython 7

Notez qu'avec IPython (et donc aussi dans les notebooks) c'est encore plus simple ; en effet IPython s'est débrouillé pour autoriser la syntaxe suivante :

```
[2]: # depuis ipython, ou dans un notebook, vous pouvez faire simplement
      await hello_world()
```

Hello World

Mise en garde attention toutefois, je vous mets en garde contre le fait que ceci est une commodité pour nous faciliter la vie, mais elle est spécifique à IPython et ne va pas fonctionner tel quel dans un programme exécuté directement par l'interpréteur Python standard.

```
[3]: # un code cassé
      !cat data/broken-await.py
```

```
import asyncio

async def hello_world():
    await asyncio.sleep(0.2)
    print("Hello World")

# ceci ne fonctionne pas
# en Python standard
await hello_world()
```

```
[4]: # la preuve
      !python data/broken-await.py
```

```
File "data/broken-await.py", line 9
    await hello_world()
    ^
```

SyntaxError: 'await' outside function

Nous avons choisi de ne pas utiliser ce trait dans les notebooks, car cela pourrait créer de la confusion, mais n'hésitez pas à l'utiliser de votre côté une fois que tout ceci est bien acquis.

À propos de Python-3.8

Avec Python 3.8 - pas encore disponible à l'heure où j'écris ceci en Avril 2020 - il y a peu de changements concernant `asyncio`, ils sont décrits ici :

<https://docs.python.org/3/whatsnew/3.8.html#asyncio>

Notez toutefois l'apparition d'une REPL (read-eval-print-loop) qui supporte justement `await` au toplevel

8.2.6 Conclusion

Pour conclure cet avertissement, ne vous formalisez pas si vous voyez dans le cours des pratiques qui sont dépassées. Les différences par rapport aux pratiques actuelles - même si on les voit très visibles dans ce cours introductif - sont en réalité mineures au niveau de ce qu'il est important de comprendre quand on aborde d'un oeil neuf ce nouveau paradigme de programmation.

8.3 w8-s4-cl-async-http

Essayez vous-même

8.3.1 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 dans 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, on voit ici un réel gain à l'utilisation de communications asynchrones (à cause de conditions réseau un peu erratiques lors de la vidéo, on n'y voit pas bien le gain obtenu) :

```
$ 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.

8.4 w8-s8-c1-players

asyncio - un exemple un peu plus réaliste

8.4.1 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 bibliothèque - 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 matière de gestion de sous-processus, car c'est quelque chose que l'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-processus) 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) :



configurations prédéfinies

Pour éviter de nous noyer dans des configurations compliquées, on a embarqué dans `players` plusieurs (4) configurations prédéfinies - voyez la globale `predefined`.

Dans tous les cas chacune de ces configurations crée deux joueurs.

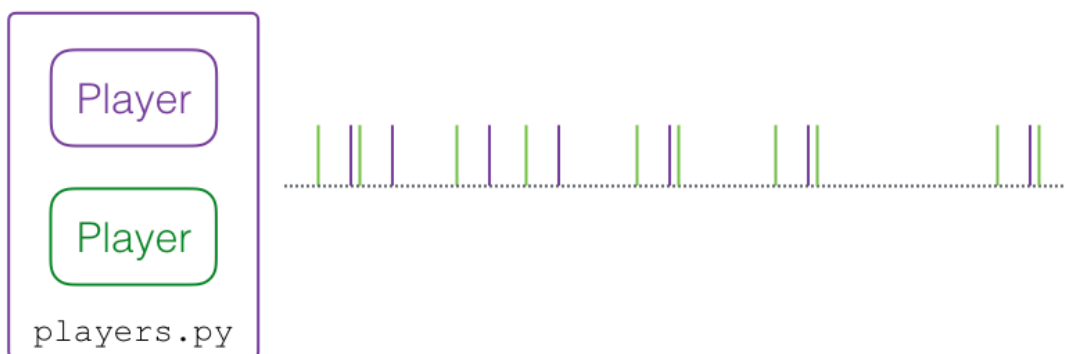
```
[1]: # par exemple la config. prédéfinie # 1
# ressemble à ceci
from data.players import predefined

for predef, players in predefined.items():
    print(f"predefined {predef}: {players}")
```

```
predefined 1: [Players john: 3x0.8s + mary: 7x0.4s]
predefined 2: [Players bill: 5x0.5s + jane: 4x0.7s]
predefined 3: [Players augustin: 8x0.8s + randalphe: 10x0.6s]
predefined 4: [Players bertrand: 12x0.5s + juliette: 8x0.7s]
```

Ce qui signifie qu'avec la config. #1, on génère 3 événements pour `john`, et 7 pour `mary`; et la durée entre les événements de `john` est tirée au hasard entre 0 et 0.8s.

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 :



```
[2]: # je peux lancer un sous-processus
# depuis le notebook
# ici la config #1
!data/players.py
```

```
S john
S mary
S mary
E mary
```

```
W mary
E mary
S john
W mary
S mary
W john
```

```
[3]: # ou une autre configuration
!data/players.py 2
```

```
S bill
S jane
S bill
N jane
S bill
E bill
S bill
W jane
W jane
```

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.

8.5 w8-s8-c2-game

Gestion de sous-process

8.5.1 Complément - niveau avancé

Dans ce second notebook, nous allons étudier un deuxième programme Python, que j'appelle `game.py` (en fait c'est le présent notebook).

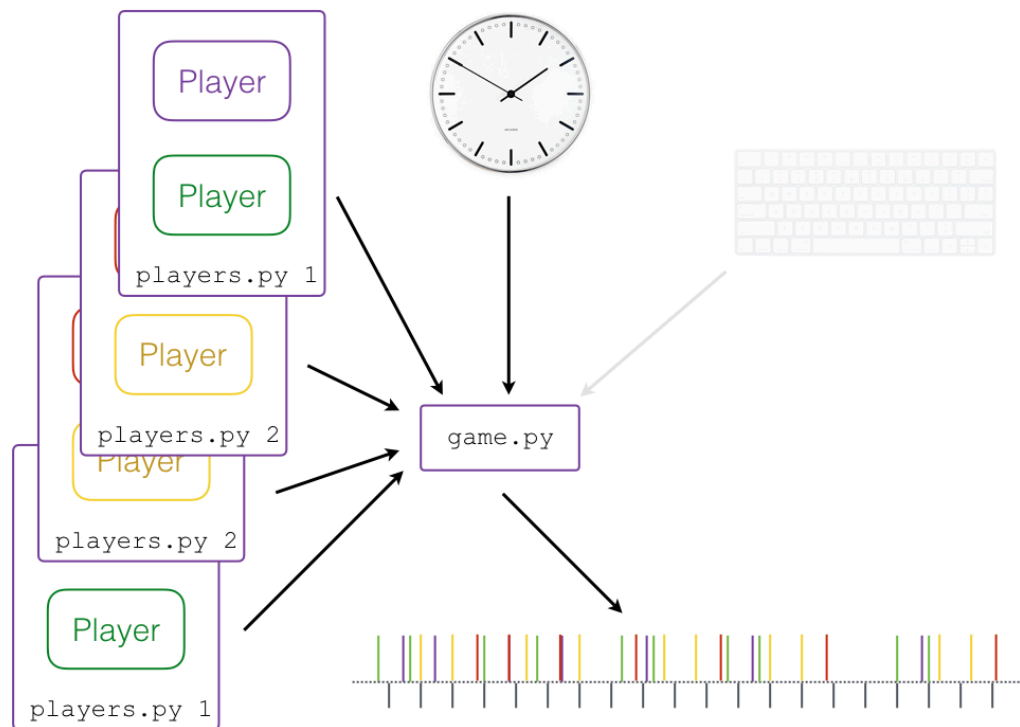
Fonctions de `game.py`

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-process) de `players.py` ;
- multiplexer (agrégateur) : on veut lire et imprimer au fur et à mesure les messages émis par les sous-processus ;
- 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 navigateur 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; il vous suffira de taper au clavier un numéro de 1 à 4 (suivi de Entrée) au moment où voulez ajouter une paire de joueurs, dans une des 4 configurations prédéfinies de `players.py`.

Terminaison

On choisit de terminer le programme `game.py` lorsque le dernier sous-processus `players.py` se termine.

Le programme `game.py`

C'est ce notebook qui va jouer pour nous le rôle du programme `game.py`.

```
[1]: import asyncio
import sys
```

```
[2]: # cette constante est utile pour déclarer qu'on a l'intention
# de lire les sorties (stdout 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-process selon un scénario configurable. Pour ne pas trop se compliquer la vie on choisit de représenter un scénario (un script) comme une liste de tuples de la forme :

```
script = [ (time, predef), ...]
```

qui signifie de lancer, un délai de `time` secondes après le début du programme, le programme `players.py` dans la configuration `predef` - toujours de 1 à 4 donc.

```
[3]: class Scheduler:

    def __init__(self, script):

        # on trie le script par ordre chronologique
        self.script = list(script)
        self.script.sort(key = lambda time_predef : time_predef[0])

        # juste pour donner un numéro à chaque processus
        self.counter = 1
        # combien de processus sont actifs
        self.running = 0

    async def run(self):
        """
        fait tout le travail, c'est-à-dire :
        * lance tous les sous-processus 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.py avec cette config
```

```

    puis
    écoute à la fois stdout et stderr, et les imprime
    (bon c'est vrai que players n'écrit rien sur stderr)
    attend la fin du sous-processus (avec wait())
    et retourne son code de retour (exitcode) du sous-processus

    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
    # l'option python -u sert à désactiver le buffering sur stdout
    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
    elle retourne lorsque le processus 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

        # bien qu'ici players n'écrit que de l'ASCII
        # readline() nous renvoie un objet `bytes`
        # qu'il faut convertir en str
        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 :

```

[4]: 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 :

```
[5]: class Game:

    def __init__(self, script):
        self.script = script

    def mainloop(self):
        loop = asyncio.get_event_loop()

        # on met ensemble une clock et un scheduler
        clock = Clock()
        scheduler = Scheduler(self.script)

        # et on fait tourner le tout
        asyncio.ensure_future(clock.run())
        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

```
[6]: # nous allons juxtaposer 2 instances de players.py
# et donc avoir 4 joueurs dans le jeu
game = Game( [(0.5, 1), (1., 2)])
```

```
[ ]: # si vous êtes dans un notebook
# cette exécution fonctionne, mais pour de sombres raisons
# liées à des évolutions de IPython, le kernel va mourir
# à la fin; ce n'est pas important..
game.mainloop()

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

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 avons 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 claviers, et ajouter à la volée des sous-process qui n'étaient pas prévus initialement dans le scénario.

8.5.2 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...](#)

8.6 w8-s9-c1-news-python37

Nouveautés par rapport aux vidéos

8.6.1 Complément - niveau intermédiaire

Comme on l'a signalé au début de la semaine, `asyncio` a subi quelques modifications dans Python-3.7, que nous allons rapidement illustrer dans ce complément.

Nous verrons aussi par ailleurs une curiosité liée à la dernière version de IPython, qui vise à faciliter le débogage et la mise au point de code asynchrone.

Python-3.7 et `asyncio`

Documentation L'évolution la plus radicale est une refonte totale de la documentation.

C'est une très bonne nouvelle, car de l'aveu même de Guido van Rossum, la documentation en place pour les versions 3.5 et 3.6 était particulièrement obscure ; [voici comment il l'a annoncé](#) :

Finally the asyncio docs are not an embarrassment to us all.

Si vous avez déjà eu l'occasion de parcourir ces anciennes documentations, et que vous les avez trouvées indigestes, sachez que vous n'êtes pas seul dans ce cas ;) Dans tous les cas je vous invite à [parcourir la nouvelle version](#), qui a le mérite d'apporter plus de réponses qu'elle ne soulève d'interrogations. Ce qui n'était pas vraiment le cas avant, c'est donc un grand progrès :)

Accès plus facile Un certain nombre de changements ont été apportés à la librairie pour en rendre l'accès plus facile.

Notamment, comme on l'a évoqué en début de semaine, on peut maintenant faire fonctionner une simple coroutine à des fins pédagogiques en faisant plus simplement :


```
>>> import asyncio
>>> async def hello_world():
...     await asyncio.sleep(0.2)
...     print("hello world")
...
>>> asyncio.run(hello_world())
hello world
```

On a également créé des raccourcis, comme par exemple :

- `asyncio.create_task()` est un alias pour `asyncio.get_event_loop().create_task()` ;
- de même `asyncio.current_task()` et `asyncio.all_tasks()` font ce que vous imaginez ;

Commodité Changement un peu plus profond, la fonction `asyncio.get_running_loop()` permet d'accéder à la boucle courante.

Si vous avez lu du code `asyncio` plus ancien, vous avez peut-être remarqué une tendance prononcée à passer un objet `loop` en paramètre à peu près partout. Grâce à cette fonction, cela n'est plus nécessaire, on est garanti de pouvoir retrouver, à partir de n'importe quelle coroutine, l'objet boucle qui nous pilote.

De manière corollaire, une méthode `get_loop` a été ajoutée aux classes `Future` et `Task`.

Pas de changement de fond Sinon, en terme des concepts fondamentaux, tout le contenu du cours reste valide.

Pour en savoir plus Vous retrouverez tous les détails dans la page suivante :

<https://docs.python.org/3/whatsnew/3.7.html#whatsnew37-asyncio>

IPython7 et `asyncio`

`await` dans ipython-7

Cette section ne s'applique pas stricto sensu à Python-3.7, mais à la version 7 de IPython.

Le sujet, c'est ici encore de raccourcir le boilerplate nécessaire, lorsque vous avez écrit une coroutine et que vous voulez la tester.

Python standard

Voici d'abord ce qui se passe avec l'interpréteur Python standard (ici en 3.7) :

```
$ python3
Python 3.7.0 (default, Jun 29 2018, 20:14:27)
  <snip>
>>> import asyncio
>>>
>>> async def hello_world():
...     await asyncio.sleep(0.2)
...     print("hello world")
...
>>> await(hello_world())
```

```
File "<stdin>", line 1
SyntaxError: 'await' outside function
>>>
>>> asyncio.run(hello_world())
hello world
```

La syntaxe de Python nous interdit en effet d'utiliser `await` en dehors du code d'une coroutine, on l'a vu dans une des vidéos, et il nous faut faire appel à `asyncio.run()`.

IPython-7 : on peut faire `await` au toplevel!

Pour simplifier encore la mise en place de code asynchrone, depuis ipython-7, on peut carrément déclencher une coroutine en invoquant `await` dans la boucle principale de l'interpréteur :

```
$ ipython3
Python 3.7.0 (default, Jun 29 2018, 20:14:27)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.0.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import asyncio

In [2]: async def hello_world():
...:     await asyncio.sleep(0.2)
...:     print("hello world")

In [3]: await(hello_world())
hello world
```

Du coup, cette façon de faire fonctionnera aussi dans un notebook, si vous avez la bonne version de IPython en dessous de Jupyter.

Chapitre 9

Sujets avancés

9.1 w9-s2-c1-decorateurs

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 bibliothèque standard) pour des thèmes qui sont propices à la décoration de code.

Nous allons voir en détail 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 :

```
[1]: # un rappel du code montré dans la vidéo
class NbAppels:
    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)
```

```
[2]: # 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]
```

```
[3]: # pour se convaincre que nous sommes bien en log2(n)
      from math import log
```

```
[4]: n1 = 100

      log(n1)/log(2)
```

```
[4]: 6.643856189774725
```

```
[5]: fibo_log(n1)
```

```
1-ème appel à fibo_aux
2-ème appel à fibo_aux
3-ème appel à fibo_aux
4-ème appel à fibo_aux
5-ème appel à fibo_aux
6-ème appel à fibo_aux
7-ème appel à fibo_aux
8-ème appel à fibo_aux
```

```
[5]: 354224848179261915075
```

```
[6]: # on multiplie par 2**4 = 16,
      # donc on doit voir 4 appels de plus
      n2 = 1600

      log(n2)/log(2)
```

```
[6]: 10.643856189774725
```

```
[7]: fibo_log(n2)
```

```
9-ème appel à fibo_aux
10-ème appel à fibo_aux
11-ème appel à fibo_aux
12-ème appel à fibo_aux
13-ème appel à fibo_aux
14-ème appel à fibo_aux
15-ème appel à fibo_aux
16-ème appel à fibo_aux
```

17-ème appel à fibo_aux
 18-ème appel à fibo_aux
 19-ème appel à fibo_aux
 20-ème appel à fibo_aux

```
[7]: 107334514891896111031216090430387104771669252419256454134240993703556054568
      521697360339918760147628083408658484474761734261151621728188903238371381
      367829518650545384174940352297859710025879326389023114160189041561702693
      547204608963635581681290042311384152252047385825507207910615814639340927
      26107458349298577292984375276210232582438075
```

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

```
[8]: # 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 :

```
[9]: # 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 :

```
[10]: fibo_cache(300)
```

```
[10]: 222232244629420445529739893461909967206666939096499764990979600
```

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 :

```
[11]: len(fibo_cache.cache)
```

```
[11]: 301
```

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, `decorée` ; on s'arrange naturellement pour que `decorée` 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 façon 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 décorée 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 :

```
[12]: help(fibo_cache)
```

Help on function decoree in module __main__:

```
decoree(*args)
```

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 `+++` :

```
[13]: # une deuxième implémentation de memoize, avec la doc

import functools                                     # +++

# un décorateur 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

```

```

[14]: # 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 :

```
[15]: help(fibo_cache2)
```

Help on function fibo_cache2 in module __main__:

```

fibo_cache2(n)
  Un fibonacci hyper-lent (exponentiel) se transforme
  en temps linéaire une fois que les résultats sont cachés

```

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 métaclassse 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 comme 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 :

```
[16]: import time

# comme pour memoize, on est limité ici et on ne peut pas
# supporter les appels à la **kwargs, 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 décorateur 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 première 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 là on peut accéder à 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

```

```

[17]: @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)

```

```

[18]: fibo_cache_expire(300)

```

```

[18]: 222232244629420445529739893461909967206666939096499764990979600

```

```

[19]: fibo_cache_expire.cache[(200,)]

```

```

[19]: (280571172992510140037611932413038677189525, 1590419044.613409)

```

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` a 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 clôture](#), 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 techniques 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'inspection 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.