Chapitre 5

Itération, importation et espace de nommage

5.1 Les instructions `break` et `continue`

5.1.1 Complément - niveau basique

`break` **et** `continue`

En guise de rappel de ces deux notions que nous avons déjà rencontrées dans la séquence consacrée aux 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 :

```
In [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 Une limite de la boucle `for`

5.2.1 Complément - niveau basique

Pour ceux qui veulent suivre le cours au niveau basique, retenez seulement que dans une boucle `for` sur un objet mutable, **il ne faut pas modifier le sujet** de la boucle.

Ainsi par exemple il ne **faut pas faire** quelque chose comme ceci :

```
In [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
        for valeur in ensemble:
            if 'bert' not in valeur:
                ensemble.discard(valeur)


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

        RuntimeError                              Traceback (most recent call last)

        <ipython-input-1-966b4462b0c4> in <module>()
          4
          5 # ceci semble une bonne idée mais ne fonctionne pas
    ----> 6 for valeur in ensemble:
          7     if 'bert' not in valeur:
          8         ensemble.discard(valeur)


        RuntimeError: Set changed size during iteration
```

Comment faire alors ?

Première remarque, votre premier réflexe pourrait être de penser à une compréhension d'ensemble :

```
In [2]: ensemble2 = {valeur for valeur in ensemble if 'bert' in valeur}
        ensemble2

Out[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*.

```
In [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 :

```
In [4]: liste = [[1], [2], [3]]
        print('avant', liste)
```

```
avant [[1], [2], [3]]
```

```
In [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 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.

```
In [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** :

```
In [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.

```
In [3]: import string
        support = string.ascii_lowercase
        print(f'support={support}')

support=abcdefghijklmnopqrstuvwxyz
```

```
In [4]: # range
        for x in range(3, 8):
            print(x)
```

```
3
4
5
6
7
```

```
In [5]: # islice
        for x in itertools.islice(support, 3, 8):
            print(x)
```

```
d
e
f
g
h
```

5.4 Programmation fonctionnelle

5.4.1 Complément - niveau basique

Pour résumer

La notion de programmation fonctionnelle consiste essentiellement à pouvoir manipuler les fonctions comme des objets à part entière, et à les passer en argument à d'autres fonctions, comme cela est illustré dans la vidéo.

On peut créer une fonction par l'intermédiaire de :

— l'*expression* `lambda:`, on obtient alors une fonction *anonyme* ;

— l'*instruction* `def` et dans ce cas on peut accéder à l'objet fonction par son nom.

Pour des raisons de syntaxe surtout, on a davantage de puissance avec `def`.

On peut calculer la liste des résultats d'une fonction sur une liste (plus généralement un itérable) d'entrées par :

— `map`, éventuellement combiné à `filter` ;

— une compréhension de liste, éventuellement assortie d'un `if`.

Nous allons revoir les compréhensions dans la prochaine vidéo.

5.4.2 Complément - niveau intermédiaire

Pour les curieux qui ont entendu le terme de *map - reduce* , voici la logique derrière l'opération `reduce`, qui est également disponible en Python au travers du module `functools`.

`reduce`

La fonction `reduce` permet d'appliquer une opération associative à une liste d'entrées. Pour faire simple, étant donné un opérateur binaire $\otimes$ on veut pouvoir calculer

$$x_1 \otimes x_2 ... \otimes x_n$$

De manière un peu moins abstraite, on suppose qu'on dispose d'une **fonction binaire** `f` qui implémente l'opérateur $\otimes$, et alors

$$\texttt{reduce}\ (f, [x_1, ..x_n]) = f(...f(f(x_1, x_2), x_3), .., x_n)$$

En fait `reduce` accepte un troisième argument - qu'il faut comprendre comme l'élément neutre de l'opérateur/fonction en question - et qui est retourné lorsque la liste en entrée est vide.

Par exemple voici - encore - une autre implémentation possible de la fonction `factoriel`.

On utilise ici le module `operator`, qui fournit sous forme de fonctions la plupart des opérateurs du langage, et notamment, dans notre cas, `operator.mul` ; cette fonction retourne tout simplement le produit de ses deux arguments.

```
In [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` :

```
In [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 Tri de listes

5.5.1 Complément - niveau intermédiaire

Nous avons vu durant une semaine précédente comment faire le tri simple d'une liste, en utilisant éventuellement le paramètre `reverse` de la méthode `sort` sur les listes. Maintenant que nous sommes familiers avec la notion de fonction, nous pouvons approfondir ce sujet.

Cas général

Dans le cas général, on est souvent amené à trier des objets selon un critère propre à l'application. Imaginons par exemple que l'on dispose d'une liste de tuples à deux éléments, dont le premier est la latitude et le second la longitude :

```
In [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 :

```
In [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` to module `operator`

```
In [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.getitem
        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` :

```
In [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 :

```
In [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

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

Écrivez une fonction qui :

— accepte en argument une liste de listes,

— et qui retourne **la même liste**, mais avec toutes les sous-listes **triées en place**.

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

Écrivez votre code ici :

```
In [ ]: def multi_tri(listes):
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_multi_tri.correction(multi_tri)
```

5.5.3 Exercice - niveau intermédiaire

Tri de plusieurs listes, dans des directions différentes

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

Modifiez votre code pour qu'il accepte cette fois **deux** arguments listes que l'on suppose de tailles égales.

Comme tout à l'heure le premier argument est une liste de listes à trier.

À présent le second argument est une liste (ou un tuple) de booléens, de même cardinal que le premier argument, et qui indiquent l'ordre dans lequel on veut trier la liste d'entrée de même rang. `True` signifie un tri descendant, `False` un tri ascendant.

Comme dans l'exercice `multi_tri`, il s'agit de modifier en place les données en entrée, et de retourner la liste de départ.

```
In [ ]: # Pour être un peu plus clair, voici à quoi on s'attend
        exo_multi_tri_reverse.example()
```

À vous de jouer :

```
In [ ]: def multi_tri_reverse(listes, reverses):
            "<votre_code>"
```

```
In [ ]: # et pour vérifier votre code
        exo_multi_tri_reverse.correction(multi_tri_reverse)
```

5.5.4 Exercice - niveau 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.

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

On vous demande d'écrire une fonction qui prend en argument :

— une fonction `f`, dont vous savez seulement que le premier argument est numérique, et qu'elle ne prend **que des arguments positionnels** (sans valeur par défaut) ;

— un nombre quelconque - mais au moins 1 - d'arguments positionnels `args`, dont on sait qu'ils pourraient être passés à `f`.

Et on attend en retour le résultat de `f` appliqués à tous ces arguments, mais avec le premier d'entre eux multiplié par deux.

Formellement : doubler_premier(f, x_1, x_2,..., x_n) = f($2 * x_1$, x_2,..., x_n)

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

$$distance\ (x_1, \ldots, x_n) = \sqrt{\sum x_i^2}$$

```
In [ ]: # rappel sur la fonction distance:
        from corrections.exo_distance import distance
        distance(3.0, 4.0)
```

```
In [ ]: distance(4.0, 4.0, 4.0, 4.0)
```

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

```
In [ ]: # ATTENTION vous devez aussi définir les arguments de la fonction
        def doubler_premier(votre, signature):
            return "votre code"
```

```
In [ ]: exo_doubler_premier.correction(doubler_premier)
```

5.5.5 Exercice - niveau intermédiaire

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

Vous devez maintenant écrire une deuxième version qui peut fonctionner avec une fonction quelconque (elle peut avoir des arguments nommés avec valeurs par défaut).

La fonction `doubler_premier_kwds` que l'on vous demande d'écrire maintenant prend donc un premier argument `f` qui est une fonction, un second argument positionnel qui est le premier argument de `f` (et donc qu'il faut doubler), et le reste des arguments de f, qui donc, à nouveau, peuvent être nommés ou non.

```
In [ ]: # quelques exemples de ce qui est attendu
        # avec ces deux fonctions

        def add3(x, y=0, z=0):
            return x + y + z

        def mul3(x=1, y=1, z=1):
            return x * y * z

        exo_doubler_premier_kwds.example()
```

Vous remarquerez que l'on n'a pas mentionné dans cette liste d'exemples

```
doubler_premier_kwds (muln, x=1, y=1)
```

que l'on ne demande pas de supporter puisqu'il est bien précisé que doubler_premier a deux arguments positionnels.

```
In [ ]: # ATTENTION vous devez aussi définir les arguments de la fonction
        def doubler_premier_kwds(votre, signature):
            "<votre code>"

In [ ]: exo_doubler_premier_kwds.correction(doubler_premier_kwds)
```

5.6 Comparaison de fonctions

5.6.1 Exercice - niveau avancé

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

À présent nous allons écrire une version très simplifiée de l'outil qui est utilisé dans ce cours pour corriger les exercices. Vous aurez sans doute remarqué que les fonctions de correction prennent en argument la fonction à corriger.

Par exemple un peu plus bas, la cellule de correction fait

```
exo_compare_all.correction(compare_all)
```

dans lequel `compare_all` est l'objet fonction que vous écrivez en réponse à cet exercice.

On vous demande d'écrire une fonction `compare` qui prend en argument :

— deux fonctions `f` et `g`; imaginez que l'une d'entre elles fonctionne et qu'on cherche à valider l'autre; dans cette version simplifiée toutes les fonctions acceptent exactement un argument;

— une liste d'entrées `entrees`; vous pouvez supposer que chacune de ces entrées est dans le domaine de `f` et de `g` (dit autrement, on peut appeler `f` et `g` sur chacune des entrées sans craindre qu'une exception soit levée).

Le résultat attendu pour le retour de `compare` est une liste qui contient autant de booléens que d'éléments dans `entrees`, chacun indiquant si avec l'entrée correspondante on a pu vérifier que `f(entree) == g(entree)`.

Dans cette première version de l'exercice vous pouvez enfin supposer que les entrées ne sont pas modifiées par `f` ou `g`.

Pour information dans cet exercice :

— `factorial` correspond à `math.factorial`

— `fact` et `broken_fact` sont des fonctions implémentées par nos soins, la première est correcte alors que la seconde retourne 0 au lieu de 1 pour l'entrée 0.

```
In [ ]: # par exemple
        exo_compare_all.example()
```

Ce qui, dit autrement, veut tout simplement dire que `fact` et `factorial` coïncident sur les entrées 0, 1 et 5, alors que `broken_fact` et `factorial` ne renvoient pas la même valeur avec l'entrée `0`.

```
In [ ]: # c'est à vous
        def compare_all(f, g, entrees):
            "<votre code>"
```

```
In [ ]: # pour vérifier votre code
        exo_compare_all.correction(compare_all)
```

5.6.2 Exercice optionnel - niveau avancé

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

`compare` revisitée

Nous reprenons ici la même idée que `compare`, mais en levant l'hypothèse que les deux fonctions attendent un seul argument. Il faut écrire une nouvelle fonction `compare_args` qui prend en entrée :

— deux fonctions `f` et `g` comme ci-dessus ;

— mais cette fois une liste (ou un tuple) `argument_tuples` de **tuples** d'arguments d'entrée.

Comme ci-dessus on attend en retour une liste `retour` de booléens, de même taille que `argument_tuples`, telle que, si `len(argument_tuples)` vaut n :

$\forall i \in \{1, ..., n\}$, si `argument_tuples[i]` == [a_1,...,a_j], alors

`retour(i) == True` $\iff$ f (a_1,...,a_j) == g (a_1,...,a_j)

Pour information, dans tout cet exercice :

— `factorial` correspond à `math.factorial` ;

— `fact` et `broken_fact` sont des fonctions implémentées par nos soins, la première est correcte alors que la seconde retourne 0 au lieu de 1 pour l'entrée 0 ;

— `add` correspond à l'addition binaire `operator.add` ;

— `plus` et `broken_plus` sont des additions binaires que nous avons écrites, l'une étant correcte et l'autre étant fausse lorsque le premier argument est nul.

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

```
In [ ]: # ATTENTION vous devez aussi définir les arguments de la fonction
        def compare_args(votre, signature):
            "<votre_code>"
```

```
In [ ]: exo_compare_args.correction(compare_args)
```

5.7 Construction de liste par compréhension

5.7.1 Révision - niveau basique

Ce mécanisme très pratique permet de construire simplement une liste à partir d'une autre (ou de **tout autre type itérable** en réalité, mais nous y viendrons).

Pour l'introduire en deux mots, disons que la compréhension de liste est à l'instruction `for` ce que l'expression conditionnelle est à l'instruction `if`, c'est-à-dire qu'il s'agit d'une **expression à part entière**.

Cas le plus simple

Voyons tout de suite un exemple :

```
In [1]: depart = (-5, -3, 0, 3, 5, 10)
        arrivee = [x**2 for x in depart]
        arrivee
```

```
Out[1]: [25, 9, 0, 9, 25, 100]
```

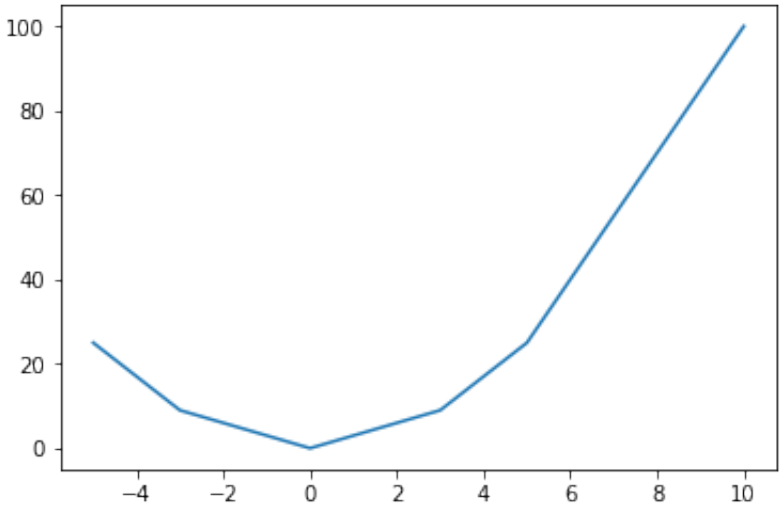
Le résultat de cette expression est donc une liste, dont les éléments sont les résultats de l'expression `x**2` pour `x` prenant toutes les valeurs de `depart`.

Remarque : si on prend un point de vue un peu plus mathématique, ceci revient donc à appliquer une certaine fonction (ici $x \to x^2$) à une collection de valeurs, et à retourner la liste des résultats. Dans les langages fonctionnels, cette opération est connue sous le nom de `map`, comme on l'a vu dans la séquence précédente.

Digression

```
In [2]: # profitons de cette occasion pour voir
        # comment tracer une courbe avec matplotlib
        %matplotlib inline
        import matplotlib.pyplot as plt
        plt.ion()
```

```
In [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 :

```
In [4]: [x**2 for x in depart if x%2 == 0]

Out[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 :

```
In [5]: [ord(x) for x in 'abc']

Out[5]: [97, 98, 99]

In [6]: [chr(x) for x in (97, 98, 99)]

Out[6]: ['a', 'b', 'c']
```

Autres types (2)

On peut également construire par compréhension des dictionnaires et des ensembles :

```
In [7]: d = {x: ord(x) for x in 'abc'}
        d

Out[7]: {'a': 97, 'b': 98, 'c': 99}

In [8]: e = {x**2 for x in (97, 98, 99) if x %2 == 0}
        e

Out[8]: {9604}
```

Pour en savoir plus

Voyez la section sur les compréhensions de liste dans la documentation python.

5.8 Compréhensions imbriquées

5.8.1 Compléments - niveau intermédiaire

Imbrications

On peut également imbriquer plusieurs niveaux pour ne construire qu'une seule liste, comme par exemple :

```
In [1]: [n + p for n in [2, 4] for p in [10, 20, 30]]
```

```
Out[1]: [12, 22, 32, 14, 24, 34]
```

Bien sûr on peut aussi restreindre ces compréhensions, comme par exemple :

```
In [2]: [n + p for n in [2, 4] for p in [10, 20, 30] if n*p >= 40]
```

```
Out[2]: [22, 32, 14, 24, 34]
```

Observez surtout que le résultat ci-dessus est une liste simple (de profondeur 1), à comparer avec :

```
In [3]: [[n + p for n in [2, 4]] for p in [10, 20, 30]]
```

```
Out[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 :

```
In [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
```

```
Out[4]: [22, 32, 14, 24, 34]
```

Ordre d'évaluation de `[[ .. for .. ] .. for .. ]`

Pour rappel, on peut imbriquer des compréhensions de compréhensions. Commençons par poser

```
In [5]: n = 4
```

On peut alors créer une liste de listes comme ceci :

```
In [6]: [[(i, j) for i in range(1, j + 1)] for j in range(1, n + 1)]

Out[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 :

```
In [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

Out[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 :

```
In [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]

Out[8]: [[(2, 2)], [(2, 4), (4, 4)]]
```

ce qui est équivalent à :

```
In [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

Out[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 :

```
In [10]: i, j
```

```
Out[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

```
In [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 :

```
In [12]: [(x, y) for x in [1, 2] for y in [1, 2]]
```

```
Out[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* :

```
In [13]: [ (i, j) for i in range(1, j + 1) for j in range(1, n + 1) ]
```

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

<ipython-input-13-ab78d593de5b> in <module>()
----> 1 [ (i, j) for i in range(1, j + 1) for j in range(1, n + 1) ]


NameError: name 'j' is not defined
```

On obtient une erreur, l'interpréteur se plaint à propos de la variable `j` (c'est pourquoi nous l'avons effacée de l'environnement au préalable).

Ce qui se passe ici, c'est que, comme nous l'avons déjà mentionné en semaine 3, le code que nous avons écrit est en fait équivalent à :

```
In [ ]: # la version bavarde de cette imbrication à plat, à nouveau :
        # [ (i, j) for i in range(1, j + 1) for j in range(1, n + 1) ]
        # serait
        resultat = []
        for i in range(1, j + 1):
            for j in range(1, n + 1):
                resultat.append((i, j))
```

Et dans cette version * dépliée* on voit bien qu'en effet on utilise `j` avant qu'elle ne soit définie.

Conclusion

La possibilité d'imbriquer des compréhensions avec plusieurs niveaux de `for` dans la même compréhension est un trait qui peut rendre service, car c'est une manière de simplifier la structure des entrées (on passe essentiellement d'une liste de profondeur 2 à une liste de profondeur 1).

Mais il faut savoir ne pas en abuser, et rester conscient de la confusion qui peut en résulter, et en particulier être prudent et prendre le temps de bien se relire. N'oublions pas non plus ces deux phrases du Zen de Python : "*Flat is better than nested*" et surtout "*Readability counts*".

5.9 Compréhensions

5.9.1 Exercice - niveau basique

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

Il vous est demandé d'écrire une fonction `aplatir` qui prend *un unique* argument `l_conteneurs` qui est une liste (ou plus généralement un itérable) de conteneurs (ou plus généralement d'itérables), et qui retourne la liste de tous les éléments de tous les conteneurs.

```
In [ ]: # par exemple
        exo_aplatir.example()
```

```
In [ ]: def aplatir(conteneurs):
            "<votre_code>"
```

```
In [ ]: # vérifier votre code
        exo_aplatir.correction(aplatir)
```

5.9.2 Exercice - niveau intermédiaire

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

À présent, on passe en argument deux conteneurs (deux itérables) `c1` et `c2` de même taille à la fonction `alternat`, qui doit construire une liste contenant les éléments pris alternativement dans `c1` et dans `c2`.

```
In [ ]: # exemple
        exo_alternat.example()
```

Indice pour cet exercice il peut être pertinent de recourir à la fonction *built-in* `zip`.

```
In [ ]: def alternat(c1, c2):
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_alternat.correction(alternat)
```

5.9.3 Exercice - niveau intermédiaire

On se donne deux ensembles A et B de tuples de la forme

```
(entier, valeur)
```

On vous demande d'écrire une fonction `intersect` qui retourne l'ensemble des objets `valeur` associés (dans A ou dans B) à un entier qui soit présent dans (un tuple de) A *et* dans (un tuple de) B.

```
In [ ]: # un exemple
        from corrections.exo_intersect import exo_intersect
        exo_intersect.example()
```

```
In [ ]: def intersect(A, B):
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_intersect.correction(intersect)
```

5.10 Expressions génératrices

5.10.1 Complément - niveau basique

Comment transformer une compréhension de liste en itérateur ?

Nous venons de voir les fonctions génératrices qui sont un puissant outil pour créer facilement des itérateurs. Nous verrons prochainement comment utiliser ces fonctions génératrices pour 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.

```
In [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]
```

```
In [2]: # c'est une expression génératrice
        generator = (x**2 for x in range(100) if x%17 == 0)
        print(generator)
```

```
<generator object <genexpr> at 0x04AF0D50>
```

Ensuite pour utiliser une expression génératrice, c'est très simple, on l'utilise comme n'importe quel itérateur.

```
In [3]: generator is iter(generator) # generator est bien un itérateur
```

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

```
In [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 :

```
In [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, 311381316, 3617541316, 4536561316}
```

5.10.2 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`

```
In [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 :

```
In [7]: # mais en fait la fonction range ne renvoie PAS une liste (depuis Python 3)
        range(4)
```

```
Out[7]: range(0, 4)
```

```
In [8]: # et en effet ce n'est pas une liste
        isinstance(range(4), list)
```

```
Out[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 temps que telle** mais seulement de cet artefact pour pouvoir itérer sur la liste des résultats, il est préférable d'utiliser une **expression génératrice**.

Voyons tout de suite sur un exemple à quoi cela ressemblerait.

```
In [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
```

```
Out[9]: <generator object <genexpr> at 0x04B014B0>
```

Comme pour `range`, le résultat de l'expression génératrice ne se laisse pas regarder avec `print`, mais comme pour `range`, on peut itérer sur le résultat :

```
In [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 :

```
In [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.

5.11 `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.

5.11.1 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 :

```
In [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 :

```
In [2]: divs(28)
```

```
Out[2]: <generator object divs at 0x04F34660>
```

Mais lorsqu'on l'utilise dans une boucle `for` :

```
In [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 :

```
In [4]: def divdivs(n):
            for i in divs(n):
                divs(i)
```

```
In [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'interieur du `try`), qui donc échoue.

Deuxième idée

Si on utilise juste `yield`, ça ne fait pas du tout ce qu'on veut :

```
In [6]: def divdivs(n):
            for i in divs(n):
                yield divs(i)

In [7]: try:
            for i in divdivs(28):
                print(i)
        except Exception as e:
            print(f"OOPS {e}")
```

```
<generator object divs at 0x051F8330>
<generator object divs at 0x051F8360>
<generator object divs at 0x051F8330>
<generator object divs at 0x051F8360>
```

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`;

```
In [8]: def divdivs(n):
            for i in divs(n):
                yield from divs(i, verbose=True)

In [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`.

5.12 Les boucles `for`

5.12.1 Exercice - niveau intermédiaire

Produit scalaire

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

On veut écrire une fonction qui retourne le produit scalaire de deux vecteurs. Pour ceci on va matérialiser les deux vecteurs en entrée par deux listes que l'on suppose de même taille.

On rappelle que le produit de X et Y vaut $\sum_i X_i * Y_i$.

On posera que le produit scalaire de deux listes vides vaut `0`.

Naturellement puisque le sujet de la séquence est les expressions génératrices, on vous demande d'utiliser ce trait pour résoudre cet exercice.

NOTE remarquez bien qu'on a dit **expression** génératrice et pas nécessairement **fonction génératrice**.

```
In [ ]: # un petit exemple
        exo_produit_scalaire.example()
```

Vous devez donc écrire :

```
In [ ]: def produit_scalaire(X, Y):
            """retourne le produit scalaire de deux listes de même taille"""
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_produit_scalaire.correction(produit_scalaire)
```

5.13 Précisions sur l'importation

5.13.1 Complément - niveau basique

Importations multiples - rechargement

Un module n'est chargé qu'une fois

De manière générale, à l'intérieur d'un interpréteur python, un module donné n'est chargé qu'une seule fois. L'idée est naturellement que si plusieurs modules différents importent le même module, (ou si un même module en importe un autre plusieurs fois) on ne paie le prix du chargement du module qu'une seule fois.

Voyons cela sur un exemple simpliste, importons un module pour la première fois :

```
In [1]: import multiple_import

chargement de multiple_import
```

Ce module est très simple, comme vous pouvez le voir

```
In [2]: from modtools import show_module
        show_module(multiple_import)

Fichier E:\python_pdf_wip\flotpython\w5\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 remarquez qu'il ne produit aucune impression

```
In [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`

```
In [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!
```

```
In [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 :

```
In [6]: from importlib import reload
        reload(multiple_import)

chargement de multiple_import


Out[6]: <module 'multiple_import' from 'E:\\python_pdf_wip\\flotpython\\w5\\multiple_import.p
```

Remarquez bien que `importlib.reload` est une fonction et non une instruction comme `import` - d'où la syntaxe avec les parenthèses qui n'est pas celle de `import`.

Notez également que la fonction `importlib.reload` a été introduite en python3.4, avant, il fallait utiliser la fonction `imp.reload` qui est dépréciée depuis python3.4 mais qui existe toujours. Évidemment, vous devez maintenant exlusivement utiliser la fonction `importlib.reload`.

NOTE spécifique à l'environnement des **notebooks** (en fait, à l'utilisation de ipython) :

À l'intérieur d'un notebook, vous pouvez faire comme ceci pour recharger le code importé automatiquement :

```
In [7]: # charger le magic 'autoreload'
        %load_ext autoreload

In [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.

5.13.2 Complément - niveau avancé

Revenons à python standard. Pour ceux qui sont intéressés par les détails, signalons enfin les deux variables suivantes.

`sys.modules`

L'interpréteur utilise cette variable pour conserver la trace des modules actuellement chargés.

```
In [9]: import sys
        'csv' in sys.modules

Out[9]: False

In [10]: import csv
         'csv' in sys.modules

Out[10]: True

In [11]: csv is sys.modules['csv']

Out[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`.

```
In [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.

```
In [13]: 'gc' in sys.builtin_module_names

Out[13]: True
```

Pour en savoir plus

Pour aller plus loin, vous pouvez lire la documentation sur l'instruction `import`

5.14 Où sont cherchés les modules ?

5.14.1 Complément - niveau basique

Pour les débutants en informatique, le plus simple est de se souvenir que si vous voulez uniquement charger vos propres modules ou packages, il suffit de les placer dans le répertoire où vous lancez la commande python. Si vous n'êtes pas sûr de cet emplacement vous pouvez le savoir en faisant :

```
In [ ]: from pathlib import Path
        Path.cwd()
```

5.14.2 Complément - niveau intermédiaire

Dans ce complément nous allons voir, de manière générale, comment sont localisés (sur le disque dur) les modules que vous chargez dans python grâce à l'instruction `import` ; nous verrons aussi où placer vos propres fichiers pour qu'ils soient accessibles à python.

Comme expliqué ici, lorsque vous importez le module `spam`, python cherche dans cet ordre :

- un module *built-in* de nom `spam` - possiblement/probablement écrit en C,
- ou sinon un fichier `spam.py` (ou `spam/__init.py__` s'il s'agit d'un package) ; pour le localiser on utilise la variable `sys.path` (c'est-à-dire l'attribut `path` dans le module `sys`), qui est une liste de répertoires, et qui est initialisée avec, dans cet ordre :
  - le répertoire où se trouve le point d'entrée ;
  - la variable d'environnement `PYTHONPATH` ;
  - un certain nombre d'emplacements définis au moment de la compilation de python.

Ainsi sans action particulière de l'utilisateur, python trouve l'intégralité de la librairie standard, ainsi que les modules et packages installés dans le même répertoire que le fichier passé à l'interpréteur.

La façon dont cela se présente dans l'interpréteur des notebooks peut vous induire en erreur. Aussi je vous engage à exécuter plutôt, et sur votre machine, le programme suivant :

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

import sys
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 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 ;

alors vous devriez observer ceci :

```
$ cd /le/repertoire/ou/vous/etes/
/le/repertoire/ou/vous/etes $ python3 /le/repertoire/du/script/run.py
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 le chemin complet `/le/repertoire/du/script/run.py`,
— et le premier terme dans `sys.path` contient `/le/repertoire/du/script!`.

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 supplante `distutils` qui lui, fait partie de la distribution standard mais qui est tombé en déshérence au fil du temps.

`setuptools` permet au programmeur d'écrire - dans un fichier qu'on appelle traditionnellement `setup.py` - le contenu de son application ; grâce à quoi on peut ensuite de manière unifiée :

— installer l'application sur une machine à partir des sources ;

— préparer un package de l'application ;

— diffuser le package dans l'infrastructure PyPI ;

— installer le package depuis PyPI en utilisant `pip3`.

Pour installer `setuptools`, comme d'habitude vous pouvez faire simplement :

```
pip3 install setuptools
```

5.15 La clause `import as`

5.15.1 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 :

```
In [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 :

```
In [2]: from importlib import import_module
```

```
In [3]: loaded = import_module(modulename)
        type(loaded)
```

```
Out[3]: module
```

Nous avons maintenant bien chargé le module `math`, et on l'a rangé dans la variable `loaded`

```
In [4]: # loaded référence le même objet module que si on avait fait
        # import math
        import math
        math is loaded
```

```
Out[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 :

```
In [5]: # un import simple
        import math

In [6]: # peut se récrire
        math = import_module('math')
```

Et :

```
In [7]: # et un import from
        from pathlib import Path

In [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 débugger vous voulez utiliser la fonction *built-in* `globals`. Sauf que, en vertu de la règle LEGB, le symbole `globals` se trouve maintenant désigner votre module, et non la fonction.

À ce stade évidemment vous pouvez (devriez) renommer votre module, mais cela peut prendre du temps parce qu'il y a de nombreuses dépendances. En attendant vous pouvez tirer profit de la clause `import as` dont la forme générale est :

```
import monmodule as autremodule
```

ce qui, toujours à la grosse louche, est équivalent à :

```
autremodule = import_module('monmodule')
```

Un symbole dans un module

On peut aussi importer un symbole spécifique d'un module, sous un autre nom que celui qu'il a dans le module. Ainsi :

```
from monmodule import monsymbole as autresymbole
```

qui fait quelque chose comme :

```
temporaire = import_module('monmodule')
autresymbole = temporaire.monsymbole
del temporaire
```

Quelques exemples

J'ai écrit des modules 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.

```
In [9]: # changer le nom du module importé
        import un_deux as one_two
        one_two.un()

la fonction un dans le module un_deux


In [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


In [11]: # on peut mélanger tout ça
         from un_deux_trois_quatre import un as one, deux, trois as three

In [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.

5.16 Récapitulatif sur `import`

5.16.1 Complément - niveau basique

Nous allons récapituler les différentes formes d'importation, et introduire la clause `import *` - et voir pourquoi il est déconseillé de l'utiliser.

Importer tout un module

L'`import` le plus simple consiste donc à uniquement mentionner le nom du module

```
In [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 :

```
In [2]: # la fonction elle-même
        print(un_deux.un)

        un_deux.un()
```

```
<function un at 0x048C6540>
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` :

```
In [3]: # dans l'espace de nommage courant on peut accéder au module lui-même
        print(un_deux)
```

```
<module 'un_deux' from 'E:\\python_pdf_wip\\flotpython\\w5\\un_deux.py'>
```

```
In [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) :

```
In [5]: from un_deux_trois import un, deux
```

À présent nous avons deux nouvelles variables dans la portée locale :

```
In [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 :

```
In [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 :

```
In [8]: import un_deux_trois
```

alors nous avons maintenant **deux variables différentes** qui désignent la fonction `un` dans le module :

```
In [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 0x048C6BB8>
<function un at 0x048C6BB8>
ce sont deux façons d'accéder au même objet True
```

En on peut modifier l'une **sans affecter** l'autre :

```
In [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 0x048C6BB8>
1
```

5.16.2 Complément - niveau intermédiaire

`import .. as`

Que l'on importe avec la forme `import unmodule` ou avec la forme `from unmodule import unevariable`, on peut toujours ajouter une clause `as nouveaunom`, qui change le nom de la variable qui est ajoutée dans l'environnement courant.

Ainsi :

— `import foo` définit une variable `foo` qui désigne un module ;
— `import foo as bar` a le même effet, sauf que le module est accessible par la variable `bar` ;

Et :

— `from foo import var` définit une variable `var` qui désigne un attribut du module ;
— `from foo import var as newvar` définit une variable `newvar` qui désigne ce même attribut.

Ces deux formes sont pratiques pour éviter les conflits de nom.

```
In [11]: # par exemple
         import un_deux as mod12
         mod12.un()

la fonction un dans le module un_deux


In [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 :

```
In [13]: from un_deux_trois_quatre import *
```

Cette forme, pratique en apparence, va donc créer dans l'espace de nommage courant les variables

```
In [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...

5.16.3 Complément - niveau avancé

`import` de manière "programmative"

Étant donné la façon dont est conçue l'instruction `import`, on rencontre une limitation lorsqu'on veut, par exemple, **calculer le nom d'un module** avant de l'importer.

Si vous êtes dans ce genre de situation, reportez-vous au module `importlib` et notamment sa fonction `import_module` qui, cette fois, accepte en argument une chaîne.

Voici une illustration dans un cas simple. Nous allons importer le module `modtools` (qui fait partie de ce MOOC) de deux façons différentes et montrer que le résultat est le même :

```
In [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
```

```
Out[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.

5.17 La notion de package

5.17.1 Complément - niveau basique

Dans ce complément, nous approfondissons la notion de module, qui a été introduite dans les vidéos, et nous décrivons la notion de *package* qui permet de créer des bibliothèques plus structurées qu'avec un simple module.

Pour ce notebook nous aurons besoin de deux utilitaires pour voir le code correspondant aux modules et packages que nous manipulons :

```
In [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" :

```
In [2]: import module_simple
```

```
Chargement du module module_simple
```

Voyons à quoi ressemble ce module :

```
In [3]: show_module(module_simple)
```

```
Fichier E:\python_pdf_wip\flotpython\w5\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 :

```
In [4]: # on peut définir sans risque une variable globale 'spam'
        spam = 'eggs'
        print("spam globale", spam)
```

```
spam globale eggs
```

```
In [5]: # qui est indépendante de celle définie dans le module
        print("spam du module", module_simple.spam)
```

```
spam du module <function spam at 0x04886930>
```

Pour résumer, un module est donc un objet python qui correspond à la fois à :

- un (seul) **fichier** sur le disque ;
- et un **espace de nom** pour les variables du programme.

La notion de package

Lorsqu'il s'agit d'implémenter une très grosse bibliothèque, il n'est pas concevable de tout concentrer en un seul fichier. C'est là qu'intervient la notion de **package**, qui est un peu aux **répertoires** ce que que le **module** est aux **fichiers**.

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 :

```
In [6]: import package_jouet

chargement du package package_jouet
Chargement du module package_jouet.module_jouet dans le package 'package_jouet'
```

Voici le contenu de ces deux fichiers :

```
In [7]: show_module(package_jouet)

Fichier E:\python_pdf_wip\flotpython\w5\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 dÃ©finir des raccourcis
9|jouet = package_jouet.module_jouet.jouet


In [8]: show_module(package_jouet.module_jouet)

Fichier E:\python_pdf_wip\flotpython\w5\package_jouet\module_jouet.py
---------------------------------------
1|print("Chargement du module", __name__, "dans le package 'package_jouet'")
2|
3|jouet = 'une variable dÃ©finie 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` corrrespond au répertoire `package_jouet`.

Cependant, pour définir un package, il faut **obligatoirement** créer dans le répertoire (celui, donc, que l'on veut exposer à python), un fichier nommé `__init__.py`.

Comme on le voit, importer un package revient essentiellement à charger le fichier `__init__.py` dans le répertoire correspondant.

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 :

```
In [9]: type(package_jouet)

Out[9]: module

In [10]: type(package_jouet.module_jouet)

Out[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 :

```
In [11]: package_jouet.spam

Out[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` :

```
In [12]: package_jouet.module_jouet

Out[12]: <module 'package_jouet.module_jouet' from
         'E:\\python_pdf_wip\\flotpython\\w5\\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 correpond à un usage assez fréquent, où on veut exposer directement dans l'espace de nom du package des symboles qui sont en réalité définis dans un module.

Avec le code ci-dessus, après avoir importé `package_jouet`, nous pouvons utiliser

```
In [13]: package_jouet.jouet

Out[13]: 'une variable définie dans package_jouet.module_jouet'
```

alors qu'en fait il faudrait écrire en toute rigueur

```
In [14]: package_jouet.module_jouet.jouet

Out[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.

Au delà de cet usage permettant de définir une sorte de raccourcis, le code placé dans `__init__.py` est chargé d'initialiser la bibliothèque. Le fichier **peut être vide** mais **doit absolument exister**. Nous vous mettons en garde car c'est une erreur fréquente de l'oublier. Sans lui vous ne pourrez importer ni le package, ni les modules ou sous-packages qu'il contient.

À nouveau c'est ce fichier qui est chargé par l'interpréteur python lorsque vous importez le package. Comme pour les modules, le fichier n'est chargé qu'une seule fois par l'interpréteur python, s'il rencontre plus tard à nouveau le même `import`, il l'ignore silencieusement.

Pour en savoir plus

Voir la section sur les modules dans la documentation python, et notamment la section sur les packages.

5.18 Usage avançés de `import`

5.18.1 Complément - niveau avancé

```
In [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 :

```
In [2]: import package_jouet
```

```
chargement du package package_jouet
Chargement du module package_jouet.module_jouet dans le package 'package_jouet'
```

`__name__`

Le nom canonique du module :

```
In [3]: package_jouet.__name__
```

```
Out[3]: 'package_jouet'
```

```
In [4]: package_jouet.module_jouet.__name__
```

```
Out[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` :

```
In [5]: package_jouet.__file__
```

```
Out[5]: 'E:\\python_pdf_wip\\flotpython\\w5\\package_jouet\\__init__.py'
```

```
In [6]: package_jouet.module_jouet.__file__
```

```
Out[6]: 'E:\\python_pdf_wip\\flotpython\\w5\\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 :

```
In [7]: import package_relatif
```

```
In [ ]: # le code de main.py
        code = find_on_disk(package_relatif, "main.py")
        !cat $code
```

```
In [ ]: # 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 génére un string aléatoire en se basant sur le module standard `random` :

```
In [ ]: # le code de random.py
        code = find_on_disk(package_relatif, "random.py")
        !cat $code
```

```
In [ ]: 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 :

```
In [10]: import package_relatif.main
```

```
On charge le module random local package_relatif.random
On charge main.py
    __name__=package_relatif.main
    alea=[[2]]
```

```
In [11]: print(package_relatif.main.alea())
```

```
[[9]]
```

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
```

```
In [12]: # voyons le code de submodule:
         import package_relatif.subpackage
```

```
In [ ]: # le code de submodule/submodule.py
        code = find_on_disk(package_relatif.subpackage, "submodule.py")
        !cat $code
```

```
In [ ]: # 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()}>>"
```

```
In [14]: import package_relatif.subpackage.submodule

On charge package_relatif.subpackage.submodule


In [15]: print(package_relatif.subpackage.submodule.alea())

<<[[6]]>>
```

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 ultérieurment (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.

5.19 Décoder le module `this`

5.19.1 Exercice - niveau avancé

Le module `this` et le *Zen de Python*

Nous avons déjà eu l'occasion de parler du *Zen de Python* ; on peut lire ce texte en important le module `this` comme ceci

```
In [ ]: import this
```

Il suit du cours qu'une fois cet import effectué nous avons accès à une variable `this`, de type module :

```
In [ ]: this
```

But de l'exercice

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

Constatant que le texte du manifeste doit se trouver quelque part dans le module, le but de l'exercice est de deviner le contenu du module, et d'écrire une fonction `decode_zen`, qui retourne le texte du manifeste.

Indices

Cet exercice peut paraître un peu déconcertant ; voici quelques indices optionnels :

```
In [ ]: # on rappelle que dir() renvoie les noms des attributs
        # accessibles à partir de l'objet
        dir(this)
```

Vous pouvez ignorer `this.c` et `this.i`, les deux autres variables du module sont importantes pour nous.

```
In [ ]: # ici on calcule le résultat attendu
        resultat = exo_decode_zen.resultat(this)
```

Ceci devrait vous donner une idée de comment utiliser une des deux variables du module :

```
In [ ]: # ces deux quantités sont égales
        len(this.s) == len(resultat)
```

À quoi peut bien servir l'autre variable ?

```
In [ ]: # se pourrait-il que d agisse comme un code simple ?
        this.d[this.s[0]] == resultat[0]
```

Le texte comporte certes des caractères alphabétiques

```
In [ ]: # si on ignore les accents,
        # il y a 26 caractères minuscules
        # et 26 caractères majuscules
        len(this.d)
```

mais pas seulement ; les autres sont préservés.

À vous de jouer

```
In [ ]: def decode_zen(this):
            "<votre code>"
```

Correction

```
In [ ]: exo_decode_zen.correction(decode_zen)
```