Initiation à la programmation

SL25Y031

Cours/TD – Chapitre 5

Université Paris Cité

Objectifs :

- Comprendre le concept de mutabilité et d'immutabilité.
- Définir et décomposer des *n*-uplets.
- Définir des listes en extension et en intension/-compréhension.
- Savoir modifier des listes.
- Comprendre la différence entre opérations fonctionnelles et non fonctionnelles.

1 Immutabilité et *n*-uplets

Immutabilité [Cours]

- Une valeur *immutable* est une valeur que l'on ne peut pas altérer.
- Tous les types de valeurs vus jusqu'à présent (`int`, `str`, etc.) sont des types de valeurs immutables.
- Prenons l'exemple des chaînes de caractères.
  - Il est possible de réassigner une variable contenant une chaîne de caractères, mais cela ne modifie pas la chaîne elle-même (seulement la variable). Par exemple :

```
s1 = "Bonjour."
s2 = s1
s2 = "Au revoir."
print(s2) # Affiche "Au revoir.".
print(s1) # Affiche "Bonjour.".
```

  - Il est possible de créer une chaîne de caractères à partir d'une autre avec, par exemple, la méthode `lower`, mais cela ne modifie pas la chaîne originale. Par exemple :

```
s1 = "Bonjour."
s2 = s1.lower()
print(s2) # Affiche "bonjour.".
print(s1) # Affiche "Bonjour.".
```

- Rappelons-nous que durant l'exécution d'un programme Python, la machine maintient en mémoire les associations entre noms de variables et valeurs, et que l'on peut représenter visuellement ces associations en dessinant d'un côté un ensemble de noms de variables, de l'autre un ensemble de valeurs, et, pour chaque nom de variables, une flèche partant du nom de variable et pointant sur l'une des valeurs.
  - Si `s1` est un nom de variable non encore définie, alors l'exécution de `s1 = "Bonjour."` ajoute un `"Bonjour."` du côté des valeurs, et `s1` du côté des variables avec une flèche de ce `s1` vers cette valeur.
  - Ensuite, si `s2` est un nom de variable elle aussi non encore définie, l'exécution de `s2 = s1.lower()` ajoute un `"bonjour."` (la valeur de `s1.lower()`) du côté des valeurs, et `s2` du côté des

variables avec une flèche de ce s2 vers cette valeur.

— Comme nous le verrons dans la suite du cours, il existe en PYTHON des valeurs *mutables*, c.-à-d. que l'on peut modifier. Modifier une valeur mutable, ce n'est pas changer les associations entre variables et valeurs (les flèches), mais modifier directement une valeur.

— Imaginons qu'une fonction `mutable_str` permette d'accéder à un type de chaînes de caractères mutables. On pourrait imaginer que ce type possède une méthode `mutable_lower` qui modifie directement la valeur sur laquelle est appelée, en la passant en bas de casse. Voici ce que l'on pourrait observer :

```
s1 = mutable_str("Bonjour.")
s2 = s1
s1.mutable_lower()
print(s2) # Affiche "bonjour.".
```

Exercice 1 (Valeurs et variables, ⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
x = 1
y = x
x += 1
x += 1
```

□

Le type `tuple` [COURS]

— Le type `tuple` est le type des structures de données appelées « *n*-uplets » en PYTHON. Un *n*-uplet est une suite finie et ordonnée de valeurs appelées « éléments ». Les *n*-uplets constitués de deux éléments sont appelés « paires ».

— Le *n*-uplet vide s'écrit « `()` » ou, de manière équivalente, `tuple()`.

— Le *n*-uplet contenant uniquement la valeur de `x1` s'écrit « `(x1,)` ». La virgule est très importante, car `(x1)` n'est pas un *n*-uplet mais vaut tout simplement `x1`. En revanche, `x1` et `(x1,)` sont deux objets différents. Par exemple :

```
print(3) # Affiche "3".
print((3)) # Affiche "3".
print(3 == (3)) # Affiche "True".
print((3,)) # Affiche "(3,)".
print(3 == (3,)) # Affiche "False".
```

— La paire constituée des valeurs `x1` et `x2` s'écrit soit « `(x1, x2,)` » soit « `(x1, x2)` ».

— De manière générale, on accède au *n*-uplet constitué des valeurs de `x1`, `x2`, ..., `xn` (dans cet ordre) avec l'expression suivante : `(x1, x2, ..., xn,)`. Dans cette expression, la dernière virgule est facultative lorsqu'il y a au moins deux éléments.

— Les *n*-uplets sont immutables (on ne peut pas les altérer).

— Les *n*-uplets sont des itérables, il est donc possible de s'en servir pour construire des boucles *for*. Cela dit, il est plutôt rare que l'on ait besoin de le faire.

— Il est possible de concaténer deux *n*-uplets avec l'opérateur +. Cela dit, il est plutôt rare que l'on ait besoin de le faire.

— Les *n*-uplets sont surtout utilisés comme valeurs de retour de certaines fonctions. Par exemple :

```
# x: int
def f(x):
  return ((x**2), (x**3))

print(f(2)) # Affiche "(4, 8)".
```

— Python ne contraint pas les éléments d'un *n*-uplet à être tous du même type. C'est quelque chose qui arrive très fréquement, notamment quand il s'agit de valeurs de retour de fonctions, et n'est pas considéré problématique. Par exemple :

```
# x: int
def g(x):
  return ((x**2), f"x = {x}")

print(g(2)) # Affiche "(4, "x = 2")".
```

— Le nombre d'éléments d'un *n*-uplet est sa *longueur*. Comme pour une chaîne de caractères, on peut accéder à la longueur d'un *n*-uplet avec la fonction `len`. Par exemple, si `u=("Hello", "world", "!")`, `len(u)` vaut 3.
— On est rarement (bien que parfois) amené à utiliser la fonction `len` sur des *n*-uplet car ils sont surtout utilisés comme valeurs de retour de certaines fonctions retournant toujours des *n*-uplets de longueurs connues. Par exemple, la fonction `f` ci-dessus retourne toujours des paires (de longueur 2).
— Le système d'indexation des éléments d'un *n*-uplet est exactement le même que celui des caractères d'une chaîne de caractères. En particulier, si `u` est un *n*-uplet non vide, il est possible d'accéder à son premier élément avec `u[0]` et `u[-len(u)]`, et à son dernier élément avec `u[len(u)-1]` et `u[-1]`.
— Si l'on cherche à accéder aux différents éléments d'un *n*-uplet `u` de longueur k *connue au moment de la programmation*, on va souvent effectuer une assignation multiple des k éléments à k variables `x1`, `x2`, ..., `xk` avec l'instruction `x1, x2, ..., xk = u`. Par exemple :

```
# x: int
def f(x):
  return ((x**2), (x**3))

a, b = f(3) # Assignation multiple.
print(a) # Affiche "9".
print(b) # Affiche "27".
```

2 Les listes

Le type `list` [Cours]

— Le type `list` est le type des structures de données appelées « listes » en Python. Une liste est une suite finie et ordonnée de valeurs appelées « éléments ».
— Comme nous le verrons par la suite, la différence majeure entre listes et *n*-uplets est que les listes sont mutables.
— Pour accéder à une liste constituée des valeurs de `x1`, `x2`, ..., `xn` (dans cet ordre), il est possible d'utiliser l'expression suivante, dite « en extension » : `[x1, x2, ..., xn]`. Par exemple :

```
l = [39, 0, 0, 17, - 29]
```

— La liste vide s'écrit `[]` ou, de manière équivalente, `list()`.
— Le nombre d'éléments d'une liste est sa *longueur*. Comme pour une chaîne de caractères, on peut accéder à la longueur d'une liste avec la fonction `len`. Par exemple, si `l=[8, 0, -4, 12, 0]`, `len(l)` vaut 5.
— Le système d'indexation des éléments d'une liste est exactement le même que celui des caractères d'une chaîne de caractères et des éléments d'un *n*-uplet. En particulier, si `l` est un liste non vide, il est possible d'accéder à son premier élément avec `l[0]` et `l[-len(l)]`, et à son dernier élément avec `l[len(l)-1]` et `l[-1]`.
— Une liste est un itérable. On peut donc s'en servir pour écrire une boucle *for*. Par exemple, le code suivant affiche chaque élément d'une liste sur une ligne (par élément), du premier au dernier :

```python
l = [8, 0, -4, 12, 0]
for e in l:
  print(e)
```

— Pour visualiser une liste `l` dans son ensemble, utiliser `print(l)`. Pour visualiser seulement l'élément d'indice `i`, utiliser `print(l[i])`.
— Comme pour une chaîne de caractères ou un n-uplet, une *tranche* d'une liste `l` est une sous-liste contiguë de `l`. On accède à la tranche commençant à la position `i` (incluse) et se terminant à la position `j` (exclue) d'une liste `l` avec `l[i:j]`. Par exemple, si l'on a `l=[8, 0, -4, 12, 0]`, `l[1:3]` vaut `[0, -4]`, la tranche composée des éléments aux positions 1 et 2.
— Comme pour une chaîne de caractères, si $0 \leq$ `i` $<$ `j` $\leq$ `len(l)`, `len(l[i:j])` vaut toujours `j-i`.
— Comme pour une chaîne de caractères, `l[i:i]` vaut toujours `[]`. Si $0 \leq$ `i` $<$ `len(l)`, alors `l[i:(i+1)]` vaut `[l[i]]`. `l[0:len(l)]` vaut toujours `l`.
— Le mot-clef `in` permet de construire des expressions booléennes dénotant si un élement est contenu dans une liste. Par exemple, `(3 in [-4, 8, 3, 9])` vaut `True`.
— Python ne contraint pas les éléments d'une même liste à être tous du même type. Par exemple, `[1, 3.4, True, None, "coucou", [], [2]]` est une expression valide qui désigne une certaine liste de longueur 7. Cependant, il est une très bonne pratique d'éviter de manipuler de telles listes hétérogènes lorsque cela est possible.

Exercice 2 (Comprendre la taille et les indices, $\star$)
Supposons que `l1=[1, 3, 5, 7, 9, 11, 13, 15, 17]`.

1. *Que vaut* `len(l1)` *?*
2. *À quels indices (positifs et négatifs) trouve-t-on la valeur* 1 *?*
3. *À quels indices trouve-t-on la valeur* 17 *?*
4. *À quels indices trouve-t-on la valeur* 9 *?*

□

Exercice 3 (Comprendre la taille et les indices, $\star$)
Supposons que `l2=[2, 2, 3, 3, 4, 5, 7]`.

1. *Que vaut* `len(l2)` *?*
2. *Que vaut l'expression* `l2[0]` *?*
3. *Que vaut l'expression* `l2[4]` *?*
4. *Pourquoi l'évaluation de* `l2[7]` *génère-t-elle une erreur ?*

□

Exercice 4 (Parcourt à l'envers, $\star$)
Étant donnée une liste `l`*, écrire une boucle permettant d'afficher chaque élément de* `l` *sur une ligne (par élément), du dernier au premier.* □

Exercice 5 (Listes et expressions, $\star\star$)
Supposons que `l=[1, 2, 4]`*. Que valent les expressions suivantes :*

1. `l[l[0]]`
2. `l[l[2] - l[1]]`

□

Exercice 6 (Listes d'itérables, $\star$)
Soit `l1=[[25, 10, 1917], [14, 7, 1789]]` *et* `l2=["Sabine", "Fred", "Jamy"]`.

1. *Que vaut* `l1[1]` *? Quel est son type ?*
2. *Que vaut* `l1[1][0]` *? Quel est son type ?*
3. *Que vaut* `l2[1]` *? Quel est son type ?*
4. *Que vaut* `l2[1][0]` *? Quel est son type ?*

□

Exercice 7 (Génération de phrases, ⋆⋆)
Soit `verbes=["parle avec", "voit"]` *et* `noms_propres=["Sabine", "Fred", "Jamy"]`.

1. *Écrire, à l'aide de boucles for, un code affichant toutes les phrases possibles constituées d'un sujet, d'un verbe et d'un objet à partir de* `verbes` *et* `noms_propres`.
2. *Modifier le code proposé à la question précédente pour ne générer que les phrases dont le sujet et l'objet sont distincts.*

□

Exercice 8 (Minimum et maximum, ⋆⋆⋆)

1. *Définir une fonction* `min` *prenant en arguments une liste* `l` *(qu'on supposera composée de valeurs numériques) et renvoyant le minimum de* `l` *si celle-ci est non vide et* `None` *sinon. La fonction doit être construite autour d'une boucle for.*
2. *Question similaire avec une fonction* `max` *renvoyant l'élément maximal de son argument s'il existe.*
3. *Définir une fonction* `min_and_max` *prenant en arguments une liste* `l` *(qu'on supposera composée de valeurs numériques) et renvoyant la paire composée du minimum et du maximum* `l` *si celle-ci est non vide et* `None` *sinon. La fonction ne doit pas faire appel aux deux fonctions* `min` *et* `max` *précédentes mais doit être construite autour d'une boucle for.*

□

Création en intension/compréhension [Cours]

— Étant donnée une liste `l=[x1, x2, ..., xn]` et une fonction à un argument `f`, il est possible d'accéder à la liste `[f(x1), f(x2), ..., f(xn)]` en utilisant l'expression suivante, dite « en intension » (ou « en compréhension ») : `[f(x) for x in l]`. Par exemple :

```
l1 = [1, -1, 2, -2, 3, -3]

# y: int
def f(y):
  return ((2 * y) + 1)

l2 = [f(x) for x in l1]
print(l2) # Affiche "[3, -1, 5, -3, 7, -5]".
```

— Il est à noter que bien qu'une expression en intension s'écrive avec le mot clef `for`, une expression en intension n'est pas une boucle *fort*, ni ne contient, au sens propre, de boucle *for*. Il y a bien un lien conceptuel fort entre les deux concepts, mais ça n'en reste pas moins deux concepts distincts, correspondant à deux constructions syntaxiques distinctes.

— Plus généralement, on peut construire par intension une liste à partir d'une expression `exp`, d'un nom de variable `x` et d'un itérable de taille finie `s` (ex : une liste, une chaîne de caractères) avec l'expression `[exp for x in s]`. Dans l'exemple suivant, l'expression `exp` est `(i**2)`, le nom de variable `x` est `i` et l'itérable `s` est `range(7)` :

```
l = [(i**2) for i in range(7)]
print(l) # Affiche "[0, 1, 4, 9, 16, 25, 36]".
```

— La syntaxe précédente crée une liste contenant un élément pour chaque élément de l'itérable `s`. Il est possible d'intégrer une condition à une définition en intension pour ne générer des élements que pour certains éléments de `s`. Par exemple :

```
l1 = [(i**2) for i in range(7) if((i % 3) == 0)]
print(l1) # Affiche "[0, 9, 36]".

l2 = [(i**2) for i in range(7) if((i % 3) != 0)]
print(l2) # Affiche "[1, 4, 16, 25]".
```

Exercice 9 (Création par intension, ⋆)

1. *Définir par intension une liste* `l1` *valant* `[0, 2, 4, 6, 8, 10, 12, 14]`.
2. *Définir par intension une liste* `l2` *valant* `[2, 4, 6, 8, 10, 12, 14, 16]`.

□

Exercice 10 (Double ou triple, ⋆⋆)
Écrire une fonction `f` *prenant en argument une liste d'entiers* `l` *et renvoyant une liste de la même taille que* `l` *contenant à chaque indice* `i` *le double de* `l[i]` *si* `l[i]` *est positif et le triple sinon. La fonction* `f` *doit renvoyer une liste créée par intension à l'aide d'une fonction auxiliaire* `f_aux` *à définir.*

Contrat :
l = [1, -1, 2, -2] → *retour : [2, -3, 4, -6]*

□

Exercice 11 (Sommes d'entiers, ⋆⋆)
Écrire une fonction `f` *prenant en argument une liste d'entiers* `l` *supposés tous positifs et renvoyant une liste de la même taille que* `l` *contenant à chaque indice* `i` *la somme des entiers de 0 à* `l[i]`. *La fonction* `f` *doit renvoyer une liste créée par intension à l'aide d'une fonction auxiliaire* `f_aux` *à définir.*

Contrat :
l = [1, 4, 0, 3] → *retour : [1, 10, 0, 6]*

□

Exercice 12 (Pas de nombre pair, ⋆)
Écrire une fonction prenant en argument une liste d'entiers `l` *et retournant la liste des éléments de* `l` *qui ne sont pas pairs.* □

Exercice 13 (Pas d'espace, ⋆)
Écrire une fonction prenant en argument une chaîne `s` *de caractères et retournant la liste des caractères de* `s` *qui ne sont pas des espaces.* □

Exercice 14 (Pas de lettre en bas de casse, ⋆⋆⋆)
Écrire une fonction prenant en argument une chaîne `s` *de caractères et retournant la liste des caractères de* `s` *qui ne sont pas des lettres en bas de casse. (Une fois une première version de la fonction écrite, vérifier que les caractères qui ne sont pas des lettres ne sont pas écartés à tort.)* □

3 Opérations sur les listes

Opérations fonctionnelles [Cours]

— Il est possible de concaténer deux listes grâce à l'opérateur +. Par exemple, `([4, 3] + [1, 2])` est égale à `[4, 3, 1, 2]`.
— L'opérateur `*` permet répéter une liste, c.-à-d. de concaténer plusieurs copies de cette liste, pour former une nouvelle liste. Par exemple, `([1, 2] * 3)` est égale à `[1, 2, 1, 2, 1, 2]`.
— La fonction `enumerate` permet d'obtenir, à partir d'une liste `l`, un itérable associant à chaque élément de `l` sa position dans `l`. Soit `l=[x1, x2, ..., xn]`, `enumerate(l)` désigne une séquence d'éléments `(0, x1)`, `(1, x2)`, ..., `((n-1), xn)` ; ses éléments sont donc des paires de valeurs, composées d'un entier et d'un élément de `l`. Par exemple :

```
l = [-2, 9, 0, 0, 25, 3]
for (i, x) in enumerate(l):
  print(f"The element at position {i} is {x}.")
```

— La fonction `reversed` permet d'obtenir, à partir d'une liste `l`, une séquence contenant les éléments de `l` dans l'ordre inverse. Par exemple, le code suivant affiche « 3 25 0 0 9 -2 » :

```
l = [-2, 9, 0, 0, 25, 3]
for x in reversed(l):
  print(x, end=" ")
```

— Les fonctions `enumerate` et `reversed` retournent des séquences qui ne sont pas du type `list` :

```
l = [1, -1, 2, -2]
print(enumerate(l)) # Affiche "<enumerate object at 0
    x7fd117371200>".
print(reversed(l)) # Affiche "<list_reverseiterator object at
    0x7fd11640d760>".
```

— Tout itérable peut cependant être converti en une liste à l'aide de la fonction `list`, à condition que l'itérable soit de taille finie (ce qui est toujours le cas pour des séquences obtenues par `enumerate` et `reversed` sur des listes) :

```
l = [1, -1, 2, -2]
print(list(enumerate(l))) # Affiche "[(0, 1), (1, -1), (2, 2),
    (3, -2)]".
print(list(reversed(l))) # Affiche "[-2, 2, -1, 1]".
```

— Une chaîne de caractères est une séquence de longueur finie et peut donc être convertie en une liste de caractères à l'aide de la fonction `list` :

```
s = 'Coucou'
print(list(s)) # Affiche "['C', 'o', 'u', 'c', 'o', 'u']".
```

— Il existe une fonction très similaire à `list` : `tuple`, qui convertit tout itérable de taille finie en un n-uplet. Par exemple :

```
print(tuple("Coucou")) # Affiche "('C', 'o', 'u', 'c', 'o', 'u
    ')".
print(tuple([0, 1, 2])) # Affiche "(0, 1, 2)".
```

— La fonction `sorted` permet d'obtenir, à partir d'une liste `l`, une séquence contenant les éléments de `l` triés par ordre croissant en utilisant l'opérateur `>` (qui désigne notamment l'ordre usuel sur les types numériques et une variante de l'ordre lexicographique pour les chaînes de caractères). Par exemple :

```
l1 = [1, -1, 2, -2]
print(sorted(l1)) # Affiche "[-2, -1, 1, 2]".

l2 = ["zèbre", "aliment", "aluminium", "université", "renard"]
print(sorted(l2)) # Affiche "['aliment', 'aluminium', 'renard
    ', 'université', 'zèbre']".
```

— Par défaut, la sortie de `sorted` est triée dans l'ordre croissant, mais `sorted` peut aussi produire une séquence d'éléments triés dans l'ordre décroissant si l'on spécifie son argument spécial `reverse` à `True`. Par exemple :

```
l = ["zèbre", "aliment", "aluminium", "université", "renard"]
print(sorted(l, reverse=True)) # Affiche "['zèbre', 'universit
    é', 'renard', 'aluminium', 'aliment']".
```

— Par défaut, la sortie de `sorted` est triée suivant l'ordre défini par l'opérateur `>` sur *les élements* de l'itérable d'entrée *eux-mêmes*, mais `sorted` peut aussi produire une séquence triée suivant l'ordre défini par `>` sur *une propriété* des éléments de l'itérable. Pour ce faire, il suffit de spécifier l'argument spécial `key` de `sorted` en indiquant une fonction renvoyant la propriété qui doit être utilisée pour le tri. Par exemple, le code suivant trie les chaînes de `l` en fonction de leur dernière lettre :

```python
# s: str
def get_last_char(s):
  if(len(s) >= 1): return s[-1]
  return ""

l = ["zèbre", "aliment", "aluminium", "université", "renard"]
print(sorted(l, key=get_last_char)) # Affiche "['renard', 'zèbre', 'aluminium', 'aliment', 'université']".
```

— Toutes les opérations vues ici sont *fonctionnelles*, c.-à-d. qu'elles génèrent de nouvelles listes ou séquences sans modifier la liste initiale. Par exemple, après exécution de la suite d'instructions suivante, seule la liste `l2` est triée, pas `l1` :

```python
l1 = [1, -1, 2, -2]
l2 = sorted(l1)
print(l2) # Affiche "[-2, -1, 1, 2]".
print(l1) # Affiche "[1, -1, 2, -2]".
```

— Les fonctions `enumerate`, `reversed` et `sorted` peuvent prendre tout type d'itérable comme argument et donc en particulier toute chaîne de caractères :

```python
s = "Coucou"
print(list(enumerate(s))) # Affiche "[(0, 'C'), (1, 'o'), (2, 'u'), (3, 'c'), (4, 'o'), (5, 'u')]".
print(list(reversed(s))) # Affiche "['u', 'o', 'c', 'u', 'o', 'C']".
print(sorted(s)) # Affiche "['C', 'c', 'o', 'o', 'u', 'u']".
```

Exercice 15 (Sandwich, ⋆)
Écrire une fonction prenant en argument deux listes `l1` *et* `l2` *et renvoyant la liste composée des éléments de* `l1`*, puis de* `l2` *puis encore de* `l1`*.*

Contrat :
l1, l2 = [1, 2], [8, 9, 0] → *retour : [1, 2, 8, 9, 0, 1, 2]*

□

Exercice 16 (Lettres à répétition, ⋆)

1. *Écrire une fonction prenant en argument un entier* `n` *et une chaîne de caractères* `s`*, et renvoyant la liste des caractères de* `s` *répétée* `n` *fois.*

   Contrat :
   n, s = 3, "az" → *retour : ["a", "z", "a", "z", "a", "z"]*
   n, s = 2, "c" → *retour : ["c", "c"]*
2. *Proposer une autre manière d'écrire cette fonction.*

□

Exercice 17 (Taille fixée, ⋆⋆)
Écrire une fonction prenant en argument un entier `n` *et une liste* `l`*, et retournant une liste de taille* `n` *remplie par les éléments de* `l` *à partir du premier et éventuellement complétée par des 0.*

Contrat :
n, l = 3, [8, 2, 3, 9, 9] → *retour : [8, 2, 3]*
n, l = 7, [8, 2, 3, 9, 9] → *retour : [8, 2, 3, 9, 9, 0, 0]*

□

Exercice 18 (Trier en fonction de la longueur, ⋆⋆)
Écrire une fonction prenant en argument une liste de chaînes de caractères `l` *et renvoyant la liste des éléments*

de `l` *triés en fonction de leur longueur par ordre croissant.* □

Exercice 19 (Trier en fonction de la longueur, ⋆⋆⋆)
Étant donnée une liste d'entiers `l`*, donner trois manières différentes pour obtenir la liste des éléments de* `l` *triés dans l'ordre décroissant.* □

Opérations non fonctionnelles [Cours]

— Contrairement aux chaînes de caractères ou au *n*-uplets, les listes sont *mutables*, c.-à-d. altérables : leurs éléments et le nombre de ces éléments sont modifiables.

— Soit une liste `l`, une position `i` définie dans `l` et `x` une valeur, l'instruction d'assignation `l[i] = x` remplace l'élément d'indice `i` de `l` par `x`. Par exemple :

```
l = [9, 8, 2, 8]
l[1] = 0

print(l) # Affiche "[9, 0, 2, 8]".
```

— L'opérateur `*` permet par exemple d'initialiser une liste avant de la remplir avec les valeurs voulues, grâce à des assignations, lorsque l'on connaît à l'avance le nombre d'éléments. Par exemple :

```
l = [0] * 10
l[3] = 2
l[7] = 1

print(l) # Affiche "[0, 0, 0, 2, 0, 0, 0, 1, 0, 0]".
```

— La méthode `append` permet de rajouter un élément à la fin d'une liste, augmentant ainsi sa longueur d'une unité. `append` ne retourne rien mais modifie la liste elle-même. Par exemple :

```
l = []
print(f"len(l)={len(l)}; l={l}") # Affiche "len(l)=0; l=[]".

l.append(2)
print(f"len(l)={len(l)}; l={l}") # Affiche "len(l)=1; l=[2]".

l.append(-7)
print(f"len(l)={len(l)}; l={l}") # Affiche "len(l)=2; l=[2,
    -7]".
```

— La méthode `extend` permet d'étendre une liste avec tous les éléments d'une seconde liste. Par exemple :

```
l = [0, 1, 2, 3]

l.extend([4, 5, 6])
print(l) # Affiche "[0, 1, 2, 3, 4, 5, 6]".

l.extend([])
print(l) # Affiche "[0, 1, 2, 3, 4, 5, 6]".
```

— L'argument de la méthode `extend` peut être n'importe quel itérable de taille finie. Par exemple :

```
l = []

l.extend(range(3))
print(l) # Affiche "[0, 1, 2]".
```

— Attention à ne pas confondre `extend` et `append`. Bien que l'utilisation de la méthode `extend` avec pour argument une valeur qui n'est pas un itérable génère une erreur, il est à l'inverse tout à fait

possible de donner un itérable comme argument à `append` ; simplement, c'est l'itérable lui-même qui sera ajouté en tant qu'élément (un seul, quelle que soit sa taille) à la liste.

```
l = [0, 1, 2]

l.extend(3) # Erreur

l.append([3, 4])
print(l) # Affiche "[0, 1, 2, [3, 4]]".
print(l == [0, 1, 2, 3, 4]) # Affiche "False".
```

— La méthode `reverse` inverse la liste sur laquelle elle est appelée. Par exemple :

```
l = [-2, 9, 0, 0, 25, 3]

l.reverse()
print(l) # Affiche "[3, 25, 0, 0, 9, -2]".
```

— La méthode `sort` trie la liste sur laquelle elle est appelée. Cette méthode accepte les mêmes arguments spéciaux `reverse` et `key` que la fonction `sorted`. Par exemple :

```
l = ["zèbre", "aliment", "aluminium", "université", "renard"]

l.sort()
print(l) # Affiche "['aliment', 'aluminium', 'renard', '
    université', 'zèbre']".

l.sort(reverse=True, key=get_last_char)
print(l) # Affiche "['université', 'aliment', 'aluminium', 'zè
    bre', 'renard']".
```

— Les méthodes `append`, `extend`, `sort` et `reverse` ne sont pas fonctionnelles. Toutes modifient la liste sur laquelle elles sont appelées et retournent toujours `None`.

— Il est important de noter que si l'on donne en argument à une fonction une valeur mutable (ex : une liste) et que l'on modifie cette valeur (ex : avec une opération non fonctionnelle) dans le corps de la fonction, alors la modification persiste après la fin de l'exécution de la fonction :

```
#l: list[int]
def f(l):
  l.reverse() # Opération non fonctionnelle.

l1 = [0, 1, 2, 3]
f(l1)
print(l1) # Affiche "[3, 2, 1, 0]".
```

Durant l'exécution de l'exemple précédant, la variable locale `l` (argument de `f`) ne désigne pas une copie de la liste désignée par `l1` mais cette liste directement. C'est pourquoi l'appel à la méthode `reverse` dans le corps de `f` a un effet observable même après la fin de l'exécution de `f` (ce que l'on appelle un « effet de bord ») : il n'y a dans cet exemple qu'une seule liste en mémoire, désignée à la fois par `l1` et `l`, et qui est inversée dans le corps de la fonction `f`.

Exercice 20 (Inversion d'éléments, ⋆⋆)

Considérer une liste `l` quelconque.

1. *Écrire une suite d'instructions remplaçant le troisième élément `l` par la valeur `1` (si un tel élément existe ; rien ne doit se produire sinon, en particulier, pas d'erreur).*
2. *Écrire une suite d'instructions inversant le premier et le second élément de `l`.*
3. *Écrire une suite d'instructions inversant le premier et le dernier élément de `l`.*

□

Exercice 21 (Écriture, ⋆)
Après exécution de la suite d'instructions suivante, que vaut `l` *?*

```
l = [2, 3, 4, 5, 6, 7]
for i in range(len(l)):
  l[i] = i
```

□

Exercice 22 (Définition par intension et construction itérative, ⋆⋆)

1. *Écrire une suite d'instructions construisant la liste* `[(i**2) for i in range(7)]` *sans utiliser de définitions par intension mais à l'aide notamment d'une boucle for et de la méthode* `append`.
2. *Même question pour la liste* `[(i**2) for i in range(7) if((i % 3) == 0)]`.

□

Exercice 23 (extend et range, ⋆)
Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?

```
l = []

l.extend(range(3))
print(l)

l.extend(range(0))
print(l)

l.extend(range(1))
print(l)

l.extend(range(2))
print(l)
```

□

Exercice 24 (Doubler chaque élément, ⋆⋆)
Écrire une fonction prenant en argument une liste `l` *et renvoyant la liste composée de chaque élément de* `l` *répété deux fois de suite.*

Contrat :
l = [8, 9, 0] → *retour : [8, 8, 9, 9, 0, 0]*

□

Exercice 25 (Intensification, ⋆⋆)
Écrire une fonction prenant en argument une liste `l` *de chaînes de caractères et renvoyant la liste composée de chaque élément de* `l` *mais en répétant deux fois de suite toute occurrence de* `"très"`.

Contrat :
l = ["Un", "très", "grand", "arbre", "."] → *retour : ["Un", "très", "très", "grand", "arbre", "."]*

□

Exercice 26 (Opérations fonctionnelles et non fonctionnelles, ⋆⋆)

1. *Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?*

```
# l: list
# x: any
def f1(l, x):
  return l.append(x)

l = [-1, 0, 1, 2]
print(f1(l, 3))
print(l)
```

2. *Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?*

```
# l: list
# x: any
def f2(l, x):
  return l + [x]

l = [-1, 0, 1, 2]
print(f2(l, 3))
print(l)
```

3. *Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?*

```
# l: list
# x: any
def f3(l, x):
  l.append(x)
  return l

l = [-1, 0, 1, 2]
print(f3(l, 3))
print(l)
```

□

Copie ou non [Cours]

— Nous avons déjà mentionné plus haut la fonction `list`, qui sert essentiellement à convertir les itérables de taille finie en listes. Si on lui passe en argument une liste `l`, elle renvoie une *copie* de `l` :

```
l1 = [-1, 1, -2, 2]
l2 = list(l)
l1.extend([-3, 3])
print(l1,) # Affiche "[-1, 1, -2, 2, -3, 3]".
print(l2) # Affiche "[-1, 1, -2, 2]".
```

— Dans un certain nombre de cas, les mêmes opérations effectuées avec `+`, `append` ou `extend` peuvent être effectuées avec une autre de ces fonctions. Par exemple, les valeurs de `l1` et `l2` sont les mêmes après exécution de n'importe laquelle des trois suites d'instructions suivantes :

```
l1 = [0, 1, 2]
l2 = [3, 4]
l1 = l1 + l2
print(l1, l2) # Affiche "[0, 1, 2, 3, 4] [3, 4]".
```

```
l1 = [0, 1, 2]
l2 = [3, 4]
l1.extend(l2)
print(l1, l2) # Affiche "[0, 1, 2, 3, 4] [3, 4]".
```

```
l1 = [0, 1, 2]
l2 = [3, 4]
for x in l2: l1.append(x)
print(l1, l2) # Affiche "[0, 1, 2, 3, 4] [3, 4]".
```

— Ces différentes opérations fonctionnent cependant de manière radicalement différentes. (Les descriptions qui suivent ignorent un certain nombre de détails techniques qui sont cependant sans impact sur les conclusions.)
  — `l1 = l1 + l2` : cette instruction (i) commande l'allocation en mémoire d'une nouvelle liste de taille `(len(l1) + len(l2))`, (ii) y copie le contenu de `l1` à partir de la position `0` puis (iii) le contenu de `l2` à partir de la position `len(l1)`, et enfin (iv) assigne cette nouvelle liste au nom de variable `l1`. Le *coût* de cette instruction (le temps requis pour son exécution) est donc grosso modo proportionnel à la somme des longueurs de `l1` et `l2`.
  — `l1.extend(l2)` : cette instruction se contente d'étendre la liste `l1` de `len(l2)` cases et d'y copier le contenu de `l2`. Le coût de cette instruction est donc proportionnel à la longueur de `l2`.
  — `for x in l2: l1.append(x)` : pour chaque élément de `l2`, cette instruction étend `l1` d'une case et y copie cet élément. Le coût de cette instruction est donc proportionnel à la longueur de `l2`.
— La différence de coût entre une concaténation (`+`) et une extension (`append`, `expend`) de liste peut être problématique, notamment si l'opération est répétée, comme lorsque l'on utilise une liste comme accumulateur. Il y a par exemple un facteur ≈ 500 entre les coûts des deux suites d'instructions suivantes :

```
l = []
for i in range(1000):
  l += [(3 * i) + 1] # Complexité : 1, puis 2, 3, ..., 1000.
```

```
l = []
for i in range(1000):
  l.append((3 * i) + 1) # Complexité : 1, puis 1, 1, ..., 1.
```

— Il est courant de construire des listes par acculumation ; il faut dans ces cas-là utiliser `append` ou `extend` et non `+`.
— Si l'on cherche à *créer* une nouvelle liste, on utilisera l'opérateur `+`. Si l'on cherche à *modifier* une liste, on utilisera les méthodes `append` ou `extend`.
— Les chaînes de caractères n'étant pas mutables, il n'y a pas d'équivalent des méthodes `append` ou `extend` pour le type `str`. Le coût de la concaténation sur les chaînes de caractères étant similaire à celui sur les listes, si l'on souhaite construire une chaîne de caractères par accumulation, le mieux est de créer par accumulation une liste de chaînes de caractères puis de les concaténer « en bloc » avec la méthode `join`. Par exemple :

```
l = []
for i in range(100):
  l.append(f"coucou {i}")
s = "".join(l) # Complexité : somme des longueurs des chaînes
   de `l`.
```

— La méthode `join` a pour argument la liste de chaînes de caractères à joindre et s'appelle sur une autre chaîne de caractères, insérée entre les premières dans le résultat. Par exemple :

```python
l = list("abcde") # ['a', 'b', 'c', 'd', 'e']
print("--".join(l)) # Affiche "a--b--c--d--e".
print(" ".join(l)) # Affiche "a b c d e".
print("".join(l)) # Affiche "abcde".
```

Exercice 27 (Concaténer les éléments d'une liste, ⋆⋆)

1. *En utilisant* `join`*, écrire une fonction prenant en argument une liste* `names` *de chaînes de caractères, et renvoyant la chaîne obtenue par la concaténation des valeurs de* `names`*, séparées par « , » (un point et un espace).*

   Contrat :
   names = *["Sabine", "Fred", "Jamy"]* → *retour : "Sabine, Fred, Jamy"*
   names = *[]* → *retour : ""*

2. *Même question mais sans utiliser* `join`*.*
3. *Parmi les deux fonctions proposées, laquelle est préférable en termes de complexité ?*

□

Exercice 28 (Concaténer les éléments d'une liste différement, ⋆⋆⋆)

1. *Sans utiliser* `join`*, écrire une fonction prenant en argument une liste* `names` *de chaînes de caractères, et renvoyant la chaîne obtenue par la concaténation des valeurs de* `names`*, séparées par « , » (un point et un espace) pour les* `(len(names) - 1)` *premières valeurs et par « et » pour les 2 dernières.*

   Contrat :
   names = *["Sabine", "Fred", "Jamy"]* → *retour : "Sabine, Fred et Jamy"*
   names = *["Sabine"]* → *retour : "Sabine"*
   names = *[]* → *retour : ""*

2. *Même question mais en utilisant* `join`*.*

□

4 À faire chez soi

Exercice 29 (Multiplier chaque élément, ⋆⋆)
Écrire une fonction prenant en argument un entier `n` *et une liste* `l`*, et renvoyant la liste composée de chaque élément de* `l` *répété* `n` *fois de suite.*

Contrat :
n, l = *3, [8, 9, 0, 1]* → *retour : [8, 8, 8, 9, 9, 9, 0, 0, 0, 1, 1, 1]*
n, l = *0, [8, 9, 0, 1]* → *retour : []*

□

Exercice 30 (Somme, ⋆)
Écrire une fonction `sum_list` *prenant en argument une liste de valeurs numériques, et renvoyant la somme de ces valeurs (et en particulier 0 si la liste est vide). La fonction doit être construite autour d'une boucle for.* □

Exercice 31 (Moyenne, ⋆)
Écrire fonction `mean_list` *prenant en argument une liste de valeurs numériques, et renvoyant la moyenne de ces valeurs ou* `None` *si la liste est vide.* □

Exercice 32 (Grelottement, ⋆)
Écrire une fonction prenant en argument un entier `n` *et affichant les* `n` *premiers « grelottements » : « brhh », « brrhh », « brrrhh », etc.* □

Exercice 33 (Première occurrence, ⋆⋆)
Écrire une fonction `first_occ` *prenant en argument une liste d'entiers* `l` *et un entier* `n`*, et renvoyant l'indice de la première occurrence de* `n` *dans* `l` *ou -1 s'il n'existe aucune telle occurrence.* □

Exercice 34 (Dernière occurrence, ⋆⋆)
Écrire une fonction `last_occ` *prenant en argument une liste d'entiers* `l` *et un entier* `n`*, et renvoyant l'indice de la dernière occurrence de* `n` *dans* `l` *ou -1 s'il n'existe aucune telle occurrence.* □

Exercice 35 (Suite dans une liste, ⋆⋆)
Écrire une fonction `progression` *prenant en argument trois entiers* `a`, `b` *et* `n`*, et renvoyant la liste* `[a, (a + b), (a + 2*b), (a + 3*b), ..., (a + (n-1)*b)]`. □

Exercice 36 (Énumération, ⋆⋆)
Écrire une fonction prenant en argument un itérable de taille finie `s` *et renvoyant la liste associant à chaque élément de* `s` *sa position dans* `s`*. La fonction* `enumerate` *ne doit pas être utilisée.*

Contrat :
s = [8, 9, 0] → *retour : [(0, 8), (1, 9), (2, 0)]*
s = "Hello" → *retour : [(0, "H"), (1, "e"), (2, "l"), (3, "l"), (4, "o")]*

□

Exercice 37 (Entrelacement, ⋆⋆)
Écrire une fonction `interlace` *prenant en argument deux listes* `l1` *et* `l2` *supposées de même longueur, et renvoyant une liste de longueur double qui contient les valeurs des deux listes de façon entrelacée, c.-à-d.* `[l1[0], l2[0], l1[1], l2[1], ..., l1[len(l1)], l2[len(l1)]]`.

Contrat :
l1, l2 = [0, 1, 6], [2, 4, 7] → *retour : [0, 2, 1, 4, 6, 7]*

□

Exercice 38 (Plagiat, ⋆⋆)

1. *Écrire une fonction* `plagiarism` *prenant en argument deux listes* `l1` *et* `l2`*, et renvoyant une paire d'indices* `(i, j)` *telle que* `l1[i] == l2[j]` *si une telle paire existe et* `None` *sinon.*
2. *Écrire une fonction* `auto_plagiarism` *prenant en argument une liste* `l` *et renvoyant une paire d'indices* `(i, j)` *telle que* `(l[i] == l[j]) and (i < j)` *si une telle paire existe et* `None` *sinon.*

□

Exercice 39 (Fonctions et liste de chaînes de caractères, ⋆⋆)

1. *Que fait la fonction* `func_ab` *définie de la manière suivante ?*

```
# n: int
def func_ab(n):
  l = []
  s = "ab"
  for _ in range(n):
    l.append(s)
    s = s + "ab"

  return l
```

2. *Simplifier cette fonction à l'aide d'une expression par intension.*
3. *Comparer la complexité de ces deux implémentations en fonction de la valeur de l'argument* `n`.

□

Exercice 40 (Comptage, ⋆⋆)
Écrire une fonction `comptage` *prenant en argument une liste d'entiers* `l` *et un nombre entier* `n`*, et renvoyant la liste d'entiers dont l'élément d'indice* `i` *(entre 0 et* `n` *inclus) est le nombre d'occurrences de l'entier* `i` *dans* `l`.

Contrat :
`l, n = [0, 1, 2, 2, 0, 0], 4` → *retour :* `[3, 1, 2, 0, 0]`
`l, n = [0, 1, 2, 2, 0, 0], 1` → *retour :* `[3, 1]`

□