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]`

□

Initiation à la programmation

SL25Y031

Cours/TD – Chapitre 6

Université Paris Cité

Objectifs :

- Savoir détecter les erreurs dans du code et le débugguer.
- Savoir implémenter un certain nombre d'opérations classiques sur les listes.

1 Implémentation d'opérations classiques sur les listes : debugguage

Exercice 1 (Présence dans une liste, $\star$)
On cherche à implémenter une fonction `is_present` *prenant en argument une valeur quelconque* `x` *et une liste* `l`*, et renvoyant* `True` *si l'un des éléments de* `l` *vaut* `x` *et* `False` *sinon. Pour chacune des propositions suivantes,*

- *expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes) ;*
- *donner un exemple de valeurs de* `x` *et* `l` *pour lesquelles la fonction ne se comporte pas comme souhaité (indiquer le résultat erroné) ;*
- *effectuer une correction minimale de cette proposition.*

1.

```
# x: any; l: list
def is_present(x, l):
  for y in l:
    if(y == x):
      return True
    else:
      return False

  return False
```

2.

```
# x: any; l: list
def is_present(x, l):
  presence = True
  for y in l:
    if(y == x):
      presence = True
    else:
      presence = False

  return presence
```

3.
```python
# x: any; l: list
def is_present(x, l):
  presence = False
  i = 0
  while((i < len(l)) and presence):
    if(l[i] == x):
      presence = True

  return presence
```

4.
```python
# x: any; l: list
def is_present(x, l):
  presence = False
  i = 0
  while((i < len(l)) and (not presence)):
    presence = (presence and (l[i] == x))

  return presence
```

□

Exercice 2 (Première position dans une liste, ⋆)

On cherche à implémenter une fonction `first_pos` *prenant en argument une valeur quelconque* `x` *et une liste* `l`*, et renvoyant le premier indice dans* `l` *d'un élément valant* `x` *si un tel indice existe et* `None` *sinon. Pour chacune des propositions suivantes,*

- *expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes) ;*
- *donner un exemple de valeurs de* `x` *et* `l` *pour lesquelles la fonction ne se comporte pas comme souhaité (indiquer le résultat eronné) ;*
- *effectuer une correction minimale de cette proposition.*

1.
```python
# x: any; l: list
def first_pos(x, l):
  for i in range(len(l)):
    if(x == l[i]):
      return i
    else:
      return None

  return None
```

2.
```python
# x: any; l: list
def first_pos(x, l):
  p = 0
  for i in range(len(l)):
    if(x == l[i]):
      p = i
    else:
      p = None

  return p
```

3.
```python
# x: any; l: list
def first_pos(x, l):
```

```
  p = None
  i = 0
  while(i < len(l)):
    if(x == l[i]):
      p = i
      i += 1

  return p
```

4.

```
# x: any; l: list
def first_pos(x, l):
  p = None
  i = 0
  while((i < len(l)) and (p != None)):
    if(x == l[i]):
      p = i
    i += 1

  return p
```

□

Exercice 3 (Compter dans une liste, ⋆)

On cherche à implémenter une fonction `count` *prenant en argument une valeur quelconque* `x` *et une liste* `l`*, et renvoyant le nombre d'occurrences de* `x` *dans* `l`*. Pour chacune des propositions suivantes,*

- *expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes) ;*
- *donner un exemple de valeurs de* `x` *et* `l` *pour lesquelles la fonction ne se comporte pas comme souhaité (indiquer le résultat eronné) ;*
- *effectuer une correction minimale de cette proposition.*

1.

```
# x: any; l: list
def count(x, l):
  k = 0
  for y in l:
    if(x == y):
      k += 1
      return k

  return k
```

2.

```
# x: any; l: list
def count(x, l):
  k = len(l)
  for y in l:
    if(x == y):
      k += 1
    else :
      k -= 1

  return k
```

□

Exercice 4 (Construire la liste des positions, ⋆)
On cherche à implémenter une fonction `pos` *prenant en argument une valeur quelconque* `x` *et une liste* `l`*, et renvoyant la liste des indices dans* `l` *où* `x` *apparaît.*

Contrat :

`x, l=2, [3, 4, 5, 8]`	→	*retourne :* `[]`
`x, l=3, [3, 3, 4, 3, 4, 5, 6, 4, 3]`	→	*retourne :* `[0, 1, 3, 8]`
`x, l=3, [3, 3, 4, 3, 4, 5, 6, 4, 3]`	→	*retourne :* `[2, 4, 7]`

Pour chacune des propositions suivantes,

- *expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes) ;*
- *donner un exemple de valeurs de* `x` *et* `l` *pour lesquelles la fonction ne se comporte pas comme souhaité (indiquer le résultat erroné) ;*
- *effectuer une correction minimale de cette proposition.*

1.
```python
# x: any; l: list
def pos(x, l):
  res = []
  for i in range(len(l)):
    if(l[i] == x):
      res = [i]

  return res
```

2.
```python
# x: any; l: list
def pos(x, l):
  return [i for i in range(l) if(x == i)]
```

3.
```python
# x: any; l: list
def pos(x, l):
  return [l[i] for (i, y) in enumerate(l) if(x == y)]
```

□

Exercice 5 (Échanger deux éléments dans une liste, ⋆)
On cherche à implémenter une fonction `swap` *prenant en argument deux entiers* `i` *et* `j` *et une liste* `l`*, et échangeant les éléments aux indices* `i` *et* `j` *dans* `l`*. Considérer la proposition suivante.*

- *Expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes).*
- *Donner un exemple de valeurs de* `i`*,* `j` *et* `l` *pour lesquelles la fonction ne se comporte pas comme souhaité (indiquer le résultat éronné).*
- *Effectuer une correction minimale de cette proposition.*

```python
# i, j: int; l: list
def swap(i, j, l):
  l[i] = l[j]
  l[j] = l[i]
```

□

Exercice 6 (Inverser une liste, ⋆⋆)
On cherche à implémenter une fonction `my_reverse` *prenant en argument une liste* `l` *et inversant* `l` *comme le ferait* `l.reverse`*. Par exemple, si* `l=['a', 'b', 'c', 'd']`*, juste après l'exécution de* `my_reverse(l)`*,* `l` *vaudra* `['d', 'c', 'b', 'a']`*. Considérer la proposition suivante.*

- *Expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes).*
- *Donner un exemple de valeur de* `l` *pour laquelle la fonction ne se comporte pas comme souhaité (indiquer le résultat éronné).*

— *Effectuer une correction minimale de cette proposition.*

```
# l: list
def reverse(l):
  for i in range(len(l)):
    l[i] = l[len(li) - 1 - i]
```

□

Exercice 7 (Rotation avant dans une liste, ⋆⋆⋆)
On cherche à implémenter une fonction `rotate_r` *prenant en argument une liste* `l` *et décalant tous les éléments de* `l` *vers l'indice supérieur (ou au début de la liste pour le dernier élément). Par exemple, si* `l=['a', 'b', 'c', 'd']`*, juste après l'exécution de* `rotate_r(l)`*,* `l` *vaudra* `['d', 'a', 'b', 'c']`*. Considérer la proposition suivante.*

— *Expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes).*
— *Donner un exemple de valeur de* `l` *pour laquelle la fonction ne se comporte pas comme souhaité (indiquer le résultat éronné).*
— *Effectuer une correction minimale de cette proposition.*

```
# l: list
def rotate_r(l):
  if(len(l) < 1): return # Equivalent à "return None".

  tmp = l[-1]
  for i in range(len(l), 0, -1): l[i] = l[i-1]
  l[0] = tmp
```

□

Exercice 8 (Rotation arrière dans une liste, ⋆⋆⋆)
On cherche à implémenter une fonction `rotate_l` *prenant en argument une liste* `l` *et décalant tous les éléments de* `l` *vers l'indice inférieur (ou à la fin de la liste pour le premier élément). Par exemple, si* `l=['a', 'b', 'c', 'd']`*, juste après l'exécution de* `rotate_l(l)`*,* `l` *vaudra* `['b', 'c', 'd', 'a']`*. Considérer la proposition suivante.*

— *Expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes).*
— *Donner un exemple de valeur de* `l` *pour laquelle la fonction ne se comporte pas comme souhaité (indiquer le résultat éronné).*
— *Effectuer une correction minimale de cette proposition.*

```
# l: list
def rotate_l(l):
  if(len(l) < 1): return # Equivalent à "return None".

  tmp = l[0]
  for i, x in enumerate(l): l[i-1] = x
  l[-1] = tmp
```

□

2 À faire chez soi

Exercice 9 (Addition, ⋆⋆⋆)
Dans cet exercice, les nombres entiers sont représentés par des listes de chiffres. Plus précisément, un entier est représenté par une liste de valeurs de type `int` *toutes entre 0 et 9 (inclus) et se lisant de la droite vers la gauche, c.-à-d. que le premier élément de la liste est le nombre d'unités, le second élément est le nombre de dizaines, le troisième élément est le nombre de centaines, etc. Par exemple,* 843 *est représenté par* `[3, 4, 8]` *et* 29 *par* `[9, 2]`.

Écrire une fonction `add` *prenant en argument deux listes* `l1` *et* `l2`*, et retournant la liste représentant la somme des deux nombres représentés par* `l1` *et* `l2`.

Par exemple, si les entrées sont les listes `l1=[3, 4, 8]` *et* `l2 = [9, 2]`*, la valeur à retourner est la liste* `[2, 7, 8]`*, qui représente le nombre* 872*. Le premier élément,* `2`*, s'obtient en additionnant* `3` *et* `9` *et en notant la retenue. Le second élément,* `7`*, s'obtient en additionnant* `4`*,* `2` *et la retenue précédente. Le troisième élément,* `8`*, s'obtient directement à partir du* `8`*, le nombre de centaines du premier nombre. Le calcul s'arrête là car il n'y a plus de chiffre à additionner ni de retenue.*

Contrat :

`l1, l2 = [5, 3, 4, 2], [6, 3, 4]`	→	*retour :* `[1, 7, 8, 2]`
`l1, l2 = [2, 1], [3, 1]`	→	*retour :* `[5, 2]`
`l1, l2 = [1], [9, 9]`	→	*retour :* `[0, 0, 1]`

□

Initiation à la programmation

SL25Y031

Cours/TD – Chapitre 7

Université Paris Cité

Objectifs :

- Définir des ensembles.
- Combiner des ensembles.
- Modifier des ensembles.
- Déclarer des dictionnaires en extension et en intension/compréhension.
- Parcourir les clefs et/ou les valeurs d'un dictionnaire.
- Modifier des dictionnaires.

1 Les ensembles

Le type `set` [Cours]

- Le type `set` est le type des structures de données appelées « ensemble ». Un ensemble est une structure mutable représentant une collection *non ordonnée* de valeurs *distinctes* appelées « éléments ».
- Les éléments d'un ensemble ne peuvent pas être de n'importe quel type. Parmi les types natifs, les types non mutables (ex : `int`, `str`, `tuple`) peuvent être utilisés pour les éléments d'un ensemble, mais pas les types mutables (ex : `list`, `set`).
- Pour accéder à un ensemble contenant les éléments `x1`, `x2`, ..., `xn`, il est possible d'utiliser l'expression suivante : `set([x1, x2, ..., xn])`. Par exemple :

```
s = set(["Sabine", "Fred", "Jamy"])
```

- Plus généralement, si `it` est un itérable de taille finie, l'expression `set(it)` vaut un ensemble dont les éléments sont les éléments de `it`.
- L'ensemble vide s'écrit `set()` ou, de manière équivalente, `set([])`.
- Pour accéder à un ensemble contenant les n éléments `x1`, `x2`, ..., `xn`, si $n \neq 0$, il est aussi possible d'utiliser l'expression suivante : `{x1, x2, ..., xn}`. Par exemple :

```
s = {"Sabine", "Fred", "Jamy"}
```

- Attention, l'expression `{}` ne vaut pas l'ensemble vide (mais le dictionnaire vide, ce que nous étudierons plus bas).
- Par définition, les éléments d'un ensemble sont tous distincts et donc, si une valeur apparaît plusieurs fois dans l'itérable utilisé pour définir un ensemble, elle n'apparaîtra quand même qu'une seule fois dans l'ensemble. Par exemple :

```
s1 = set(["Sabine", "Fred", "Jamy"])
s2 = set(["Sabine", "Fred", "Jamy", "Sabine"])
print(s2) # Affiche "{'Fred', 'Sabine', 'Jamy'}".
print(s1 == s2) # Affiche "True".
```

- Par définition, un ensemble est non ordonné. L'éventuel ordre des valeurs dans l'itérable utilisé pour définir un ensemble n'a aucune importance.

```python
s1 = set(["Sabine", "Fred", "Jamy"])
s2 = set(["Jamy", "Sabine", "Fred"])
print(s1 == s2) # Affiche "True".
```

— La fonction `print` appelée sur un ensemble affiche ses éléments dans un ordre arbitraire (c.-à-d. pouvant varier d'une exécution à l'autre).

— Le nombre d'éléments d'un ensemble est sa *longueur*. Comme pour une chaîne de caractères ou une liste, on peut accéder à la longueur d'un ensemble avec la fonction `len`. Par exemple, si `s=set(["Sabine", "Fred", "Jamy"])`, `len(s)` vaut 3.

— Le mot-clef `in` permet de construire des expressions booléennes dénotant si un élément est présent dans un ensemble. Par exemple :

```python
s = set(["Sabine", "Fred", "Jamy"])
print("Fred" in s) # Affiche "True".
print("Frédéric" in s) # Affiche "False".
```

— PYTHON ne contraint pas les éléments d'un même ensemble à être tous du même type. Par exemple, `set(["salut", True, (1, 2)])` est une expression valide qui désigne un certain ensemble de longueur 3. Cependant, il est une très bonne pratique d'éviter de manipuler de tels ensembles hétérogènes lorsque cela est possible, c.-à-d. de n'utiliser que des ensembles dont les éléments sont tous d'un même type.

— Un ensemble est un itérable, dont l'ordre d'itération est arbitraire. Par exemple, l'exécution de la suite d'instructions suivantes affichera, dans un ordre arbitraire, les chaînes `"Sabine"`, `"Fred"` et `"Jamy"` sur une ligne chacune.

```python
s = set(["Sabine", "Fred", "Jamy"])
for x in s:
  print(x)
```

Exercice 1 (Premier exercice sur les ensembles, ⋆)
Soit `s={0, 32, -5, 32, 0, 1}`.

1. *Que vaut* `len(s)` *?*
2. *Que vaut* `(0 in s)` *?*
3. *Que vaut* `("32" in s)` *?*

□

Exercice 2 (Longueur d'un ensemble, ⋆)
Soit `it` *un itérable quelconque de taille finie* `n`*, et* `s=set(seq)`.

1. *Si les éléments de* `it` *sont tous distincts, que vaut* `len(s)` *?*
2. *De manière générale, que peut-on dire de* `len(s)` *?*

□

Exercice 3 (Sont-ce des ensembles ?, ⋆⋆)
Pour chacune des expressions suivantes, dire si elle s'évalue à un ensemble et si oui, en indiquer la longueur et les éléments.

1. `set(range(-2, 10, 3))`
2. `set("Hello world!")`
3. `set([[1, 2], [1, 2]])`
4. `set([(1, 2), (1, 2)])`
5. `set({"Sabine", "Fred", "Jamy"})`
6. `set([{1}, {{1}, 2}])`

□

Exercice 4 (Sous-ensemble, ⋆⋆)
Écrire une fonction `subset` *prenant en argument deux ensembles* `s1` *et* `s2`*, et retournant* `True` *ssi* `s1` *est un* sous-ensemble *de* `s2` *(c.-à-d. si tous les éléments de* `s1` *sont des éléments de* `s2`*), et* `False` *sinon. La fonction doit être construite autour d'une boucle for.*

Contrat :

`s1, s2 = {9, 8, 2}, {1, 2, 9, 7, 8}`	→	*retour :* `True`
`s1, s2 = {9, 8, 2}, {2, 9, 8}`	→	*retour :* `True`
`s1, s2 = set(), {2, 9, 8}`	→	*retour :* `True`
`s1, s2 = {9, 8, 4}, {1, 2, 9, 7, 8}`	→	*retour :* `False`

□

Exercice 5 (Appartenances, ⋆⋆)

1. *Écrire une fonction* `parthood` *prenant en argument un ensemble* `s` *et une liste* `l`*, et retournant une liste de booléens de même longueur que* `l` *et dont l'élément d'indice* `i` *est* `True` *ssi l'élément d'indice* `i` *de* `l` *est contenu dans* `s`*, et* `False` *sinon. La fonction doit être construite autour d'une boucle for.*

   Contrat :

`s, l = {9, 8, 2}, [0, 2, 2, 10]`	→	*retour :* `[False, True, True, False]`

2. *Même question, mais en construisant la liste retournée avec une expression en intension.*

□

Opérations fonctionnelles [Cours]

— Si `s1` et `s2` sont deux ensembles, `s1.union(s2)` vaut l'ensemble contenant à la fois les éléments de `s1` et les éléments de `s2` (et uniquement ceux-là). Par exemple :

```
s = {0, 1, 2, 3}.union({0, 2, 4, 6, 8})
print(s) # Affiche "{0, 1, 2, 3, 4, 6, 8}".
```

— Si `s1` et `s2` sont deux ensembles, `s1.intersection(s2)` vaut l'ensemble contenant les valeurs qui sont à la fois éléments de `s1` et de `s2`. Par exemple :

```
s = {0, 1, 2, 3}.intersection({0, 2, 4, 6, 8})
print(s) # Affiche "{0, 2}".
```

— Si `s1` et `s2` sont deux ensembles, `s1.difference(s2)` vaut l'ensemble contenant les éléments de `s1` qui ne sont pas dans `s2`. Par exemple :

```
s = {0, 1, 2, 3}.difference({0, 2, 4, 6, 8})
print(s) # Affiche "{1, 3}".
```

— Toutes les opérations vues ici sont *fonctionnelles*, c.-à-d. qu'elles génèrent de nouveaux ensembles sans modifier les ensembles initiaux. Par exemple, dans la suite d'instructions suivante, l'ensemble `s1` n'est pas modifié après son initialisation :

```
s1 = {0, 1, 2, 3}
s2 = s1.union({0, 2, 4, 6, 8})
print(s1) # Affiche "{0, 1, 2, 3}".
```

Exercice 6 (Union, ⋆)
Soit `s1=set(range(4))` *et* `s2={2, 3, 5, 7}`,

1. `s1.union(s2)`
2. `s1.union(s1)`
3. `s2.union(s1)`
4. `s2.union(s2)`

□

Exercice 7 (Intersection, ⋆)
Soit `s1=set(range(4))` *et* `s2={2, 3, 5, 7}`,

1. `s1.intersection(s2)`
2. `s1.intersection(s1)`
3. `s2.intersection(s1)`
4. `s2.intersection(s2)`

□

Exercice 8 (Différence, ⋆)
Soit `s1=set(range(4))` *et* `s2={2, 3, 5, 7}`,

1. `s1.difference(s2)`
2. `s1.difference(s1)`
3. `s2.difference(s1)`
4. `s2.difference(s2)`

□

Opérations non fonctionnelles [Cours]

— Tout comme les listes, les ensembles sont *mutables*, c.-à-d. altérables : il est possible de leur ajouter et de leur supprimer des éléments.

— La méthode `add` permet de rajouter un élément à un ensemble s'il n'y est pas déjà. Par exemple :

```
s = set()

s.add(1)
print(s) # Affiche "{1}".

s.add(2)
print(s) # Affiche "{1, 2}".

s.add(1)
print(s) # Affiche "{1, 2}".
```

— La méthode `remove` permet de supprimer un élément à un ensemble qui le contient. Notons que `remove` lève une exception (≈ le programme plante) si son argument n'est pas un élément de l'ensemble sur lequel elle est appelée.

```
s = {1, 2, 3, 5, 7}

s.remove(1)
print(s) # Affiche "{2, 3, 5, 7}".

s.remove(1) # KeyError
```

— Une autre manière de supprimer un élément à un ensemble est d'utiliser la méthode `discard`. L'intérêt de cette méthode est qu'elle ne lève pas d'exception si son argument n'est pas un élément de l'ensemble sur lequel elle est appelée.

```
s = {1, 2, 3, 5, 7}

s.discard(1)
print(s) # Affiche "{2, 3, 5, 7}".

s.discard(1)
print(s) # Affiche "{2, 3, 5, 7}".
```

— La méthode `update`, appelée sur un ensemble `s1` avec comme argument un ensemble `s2`, rajoute à `s1` les éléments contenus dans `s2`. Par exemple :

```
s = {0, 1, 2, 3}
s.update({0, 2, 4, 6, 8})
print(s) # Affiche "{0, 1, 2, 3, 4, 6, 8}".
```

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

```
s = set()

s.update(range(3))
print(s) # Affiche "{0, 1, 2}".

s.update(range(-3, 5, 2))
print(s) # Affiche "{0, 1, 2, 3, 5, -3, -1}".
```

— Les méthodes `add`, `remove`, `discard` et `update` ne sont pas fonctionnelles. Toutes modifient l'ensemble sur laquelle elles sont appelées et retournent toujours `None`.

Exercice 9 (Caractères numériques, ⋆⋆)
Écrire une fonction `digits` *prenant en argument une chaîne de caractères* `s` *et retournant l'ensemble des caractères numériques (c.-à-d. des chiffres) apparaissant dans* `s`*. Il est possible d'utiliser la méthode* `isdigit` *qui, lorsque appelée (sans argument) sur une chaîne de caractères, retourne* `True` *si cette chaîne n'est pas vide et n'est constituée que de caractères numériques, et* `False` *sinon.*

Contrat :
`s = "Il est 13h42."` → *retour :* `{"1", "2", "3", "4"}`
`s = "Hello world!"` → *retour :* `set()`

□

Exercice 10 (Union et mise-à-jour, ⋆)
Sans utiliser la méthode `union`*, proposer une instruction équivalente à l'instruction suivante.*

```
s1 = s1.union(s2)
```

□

2 Les dictionnaires

Le type `dict` [Cours]

— Le type `dict` est le type des structures de données appelées « tableaux associatifs » ou « dictionnaires ». Un dictionnaire est une structure mutable associant une *valeur* à un nombre fini de *clefs* (une valeur par clef).
— Les valeurs d'un dictionnaire peuvent être de n'importe quel type (ex : `int`, `str`, `tuple`, `list`, `set`, `dict`), ce qui n'est pas le cas pour les clefs. Parmi les types natifs, les types non mutables (ex : `int`, `str`, `tuple`) peuvent être utilisés pour les clefs d'un dictionnaire, mais pas les types mutables (ex : `list`, `set`, `dict`).
— Pour accéder à un dictionnaire associant les valeurs `v1`, `v2`, ..., `vn` aux clefs `k1`, `k2`, ..., `kn` respectivement, il est possible d'utiliser l'expression suivante, dite « en extension » : `{k1: v1, k2: v2, ..., kn: vn}`. Par exemple :

```
d = {"Lyon": 0, "Manchester": 1, "Firenze": 2, "Marseille": 0,
     "Edinburgh": 1, "Napoli": 2}
```

— Le dictionaire vide s'écrit `{}` ou, de manière équivalente, `dict()`.

— Si la même clef apparaît plusieurs fois dans une définition de dictionnaire par extension, seule l'association de sa dernière occurrence sera prise en compte :

```
d = {"Lyon": 0, "Manchester": 1, "Lyon": 2}
print(d == {'Lyon': 2, 'Manchester': 1}) # Affiche "True".
```

— À part cela, l'ordre des couples clef-valeur dans une définition en extension n'est pas pertinent. La raison d'être d'un dictionnaire est simplement d'enregistrer des associations clef-valeur.

— La fonction `print` appelée sur un dictionnaire affiche les associations clef-valeur dans un ordre arbitraire.

— L'on accède à la valeur associée à la clef `k` d'un dictionnaire `d` avec `d[k]`. Par exemple :

```
d = {"Lyon": 0, "Manchester": 1, "Firenze": 2, "Marseille": 0,
    "Edinburgh": 1, "Napoli": 2}
print(d["Edinburgh"]) # Affiche "1".
```

— Si l'on cherche à accéder dans un dictionnaire à la valeur associée à une clef non définie, une erreur (`KeyError`) sera produite à l'exécution.

— La méthode `get`, appelée avec un argument `k`, retourne la valeur associée à la clef `k` si celle-ci est définie et `None` sinon :

```
d = {"Lyon": 0, "Manchester": 1, "Firenze": 2, "Marseille": 0,
    "Edinburgh": 1, "Napoli": 2}
print(d.get("Edinburgh")) # Affiche "1".
print(d.get("Strasbourg")) # Affiche "None".
```

— La méthode `get` accepte optionnellement un second argument, retourné à la place de `None` si le premier argument n'est pas une clef définie :

```
d = {"Lyon": 0, "Manchester": 1, "Firenze": 2, "Marseille": 0,
    "Edinburgh": 1, "Napoli": 2}
print(d.get("Strasbourg")) # Affiche "None".
print(d.get("Strasbourg", -1)) # Affiche "-1".
```

— Le nombre d'associations d'un dictionnaire est sa *longueur*. Comme pour une chaîne de caractères ou une liste, on peut accéder à la longueur d'un dictionnaire avec la fonction `len`. Par exemple, si `d={8: "pair", 5: "impair", -4: "pair", 0: "pair"}`, `len(d)` vaut 4.

— Le mot-clef `in` permet de construire des expressions booléennes dénotant si une clef est présente (en tant que clef) dans un dictionnaire. Par exemple :

```
d = {"Lyon": 0, "Manchester": 1, "Firenze": 2, "Marseille": 0,
    "Edinburgh": 1, "Napoli": 2}
print("Edinburgh" in d) # Affiche "True".
print("London" in d) # Affiche "False".
print(2 in d) # Affiche "False".
```

— Python ne contraint ni les clefs ni les valeurs d'un même dictionnaire à être tous du même type. Par exemple, `{"salut": True, 2: "hier", True: [1,2]}` est une expression valide qui désigne un certain dictionnaire de longueur 3. Cependant, il est une très bonne pratique d'éviter de manipuler de tels dictionnaires hétérogènes lorsque cela est possible, c.-à-d. de n'utiliser que des dictionnaires dont les clefs, d'une part, et les valeurs, d'une autre, sont toutes d'un même type.

— Dans un dictionnaire, une seule valeur peut être associée à une clef. Si l'on souhaite intuitivement associer plusieurs valeurs d'un certain type à une même clef, il faut alors utiliser un dictionnaire dont les valeurs sont des n-uplets/listes/ensembles. Par exemple :

```
d = {"France": {"Paris", "Lyon", "Marseille"}, "United Kingdom
    ": {"London", "Birmingham", "Glasgow"}}
```

Exercice 11 (Vérification de clefs, ⋆)
Écrire une fonction `check_keys` *prenant en argument un dictionnaire* `dico` *et une liste* `keys`*, et renvoyant* `True` *si tous les éléments de* `keys` *sont des clefs de* `dico` *et* `False` *sinon.* □

Exercice 12 (Un traducteur, ⋆)
Le but de cet exercice est l'écriture d'un traducteur automatique très primitif. Pour ce traducteur, une phrase est représentée par une liste de tokens*, qui sont des* `str` *représentant des mots ou signes de ponctuation. Ce traducteur traduit une phrase « mot à mot », c.-à-d. en construisant une liste où chaque token de la phrase source est remplacé par son équivalent dans la langue cible d'après un dictionnaire tel que celui-ci :*

```
fr2en = {"langage": "language", "un": "a", "est": "is", "merveilleux
    ": "wonderful"}
```

Quand un token de la phrase source n'est pas trouvé dans le dictionnaire, il n'est pas traduit et est inséré tel quel dans la phrase en sortie. Ce traducteur doit être implémenté sous forme d'une fonction `translate` *prenant en argument un dictionnaire* `src2tgt` *et une phrase* `src_sentence` *(une liste de chaîne de caractères), et retournant le résultat de la traduction.*

> ***Contrat :***
> `src2tgt`, `src` = `fr2en`, `["Python", "est", "merveilleux", "."]` → *retour :* `["Python", "is", "wonderful", "."]`

□

Exercice 13 (Recherche dans des dictionnaires, ⋆⋆)
On suppose donnés les dictionnaires suivants, l'un associant des noms à des prénoms de personnages (de la série The Big Bang Theory) et l'autre associant des noms d'acteur · rice · s à des noms de personnages :

```
names = {"Leonard": "Hofstadter", "Amy": "Fowler", "Sheldon": "
    Cooper", "Bernadette": "Rostenkow"}
actors = {"Fowler": "Bialik", "Cooper": "Parsons", "Rostenkow": "
    Rauch"}
```

1. *Écrire une fonction* `actor_from_character` *qui prend un prénom* `first_name` *en argument, et qui retourne le nom de l'acteur · rice qui joue ce personnage.*

   > ***Contrat :***
   > `first_name` = `"Sheldon"` → *retour :* `"Parsons"`

2. *Améliorer la fonction pour qu'elle retourne la chaîne* `"[character name unknown]"` *dans le cas où* `first_name` *n'est pas connu en tant que prénom de personnage, et retourne* `"[actor name unknown]"` *dans le cas où le nom de famille du personnage est connu mais pas le nom de l'acteur · rice qui l'interprète.*

   > ***Contrat :***
   > `first_name` = `"Penny"` → *retour :* `"[character name unknown]"`
   > `first_name` = `"Leonard"` → *retour :* `"[actor name unknown]"`

□

Création en intension/compréhension [Cours]

— Il est possible de définir un dictionnaire en utilisant l'expression suivante, dite « en intension » (ou « en compréhension »), à partir d'un itérable de taille finie `s`, d'un nom de variable `x` et de deux expressions `key` et `value` : `{key: value for x in s}`. Dans l'exemple suivant, l'itérable `s` est `range(6)`, l'expression `key` est `i`, l'expression `value` est `(2*i)` et le nom de variable `x` est `i` :

```
d = {i: (2*i) for i in range(6)}
print(d) # Affiche "{0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10}".
```

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

```
d1 = {i: (2*i) for i in range(6) if((i % 3) == 0)}
print(d1) # Affiche "{0: 0, 3: 6}".

d2 = {i: (2*i) for i in range(6) if((i % 3) != 0)}
print(d2) # Affiche "{1: 2, 2: 4, 4: 8, 5: 10}".
```

— Une autre manière courante de créer un dictionnaire est d'appeler la fonction `dict` sur un itérable de taille finie de paires. Chaque élément de la paire est alors interprété comme une association clef-valeur, enregistrée dans le dictionnaire ainsi créé. Par exemple :

```
l = [("Marie", "fr"), ("John", "en"), ("Otto", "de")]
name2language = dict(l)
print(name2language) # Affiche "{'Marie': 'fr', 'John': 'en',
    'Otto': 'de'}".
```

— Pour ces différentes manières de créer un dictionnaire aussi, si la même clef est rencontrée plusieurs fois, seule l'association correspondant à sa dernière occurrence sera conservée.

Exercice 14 (Création d'un dictionnaire par intension, ⋆)

1. *Écrire une fonction* `f` *prenant en argument une liste* `l` *dont on supposera tous les éléments uniques et renvoyant le dictionnaire associant à chaque élément de* `l` *sa position dans* `l`.

   Contrat :
   `l = ["Sabine", "Fred", "Jamy"]` → *retour :* `{"Sabine": 0, "Fred": 1, "Jamy": 2}`
2. *Que vaut* `f(["Sabine", "Fred", "Sabine", "Jamy"])`, *où* `f` *est la fonction proposée pour la question précédente ?*
3. *Plus généralement, si l'on ne se restreint pas à des listes dont tous les éléments sont uniques, dans* `f(l)`, *quelle est la valeur associée à chaque élément de* `l` *?*

□

3 Opérations sur les dictionnaires

Parcours de dictionnaires [Cours]

— La méthode `keys` permet d'accéder à un itérable contenant les clefs d'un dictionnaire. Par exemple, l'exécution de la suite d'instructions suivantes affichera, dans un ordre arbitraire, les chaînes `"Paris"`, `"London"` et `"Berlin"` sur une ligne chacune.

```
d = {"Paris": "France", "London": "United Kingdom", "Berlin":
    "Deutschland"}
for k in d.keys():
  print(k)
```

— La méthode `values` permet d'accéder à un itérable contenant les valeurs d'un dictionnaire. Par exemple, l'exécution de la suite d'instructions suivantes affichera, dans un ordre arbitraire, les chaînes `"France"`, `"United Kingdom"` et `"Deutschland"` sur une ligne chacune.

```
d = {"Paris": "France", "London": "United Kingdom", "Berlin":
    "Deutschland"}
for v in d.values():
  print(v)
```

— En PYTHON, un dictionnaire est un itérable dont les éléments sont ses clefs. Par exemple, la suite d'instructions suivante est équivalente à celle mentionnée plus haut en introduction de la méthode `keys`.

```
d = {"Paris": "France", "London": "United Kingdom", "Berlin":
    "Deutschland"}
for k in d:
  print(k)
```

— Pour éviter de faire des erreurs et faciliter la lecture du code, je vous recommande de ne jamais itérer directement sur un dictionnaire mais d'utiliser plutôt la méthode `keys`. Je vous recommande aussi de choisir des noms de variables appropriés pour les compteurs de vos boucles, comme dans les exemples précédents.

— La méthode `items` permet d'accéder à un itérable contenant les paires clefs-valeurs d'un dictionnaire. Ainsi, les deux suites d'instructions suivantes sont équivalentes :

```
d = {"Paris": "France", "London": "United Kingdom", "Berlin":
    "Deutschland"}
for (k, v) in d.items()
  print(f"La ville nommée '{k}' est la capitale du pays nommé
    '{v}'.")
```

```
d = {"Paris": "France", "London": "United Kingdom", "Berlin":
    "Deutschland"}
for k in d.keys()
  print(f"La ville nommée '{k}' est la capitale du pays nommé
    '{d[k]}'.")
```

Exercice 15 (Clefs disjointes, ⋆⋆)
Écrire une fonction `disjoint_keys` *prenant en argument deux dictionnaires* `d1` *et* `d2`*, et retournant* `True` *ssi les clefs des* `d1` *et* `d2` *sont disjointes (c.-à-d. si les deux dictionnaires n'ont aucune clef en commun), et* `False` *sinon.*

Contrat :
`d1, d2 = {"Li": 3, "H": 1, "He": 2}, {"Ne": 20.2, "F": 9.0, "O": 16.0}` → *retour :* `True`
`d1, d2 = {"H": 1, "He": 2}, {"Ne": 20.2, "He": 4.0, "Ar": 40.0}` → *retour :* `False`

□

Exercice 16 (Inversion d'un dictionnaire, ⋆⋆)
Écrire une fonction `inverse` *prenant en argument un dictionnaire* `d` *et retournant le dictionnaire dont les associations clef-valeur sont les inverses des associations clef-valeur de* `d`.

Contrat :
`d = {"Sabine": "Quindou", "Frédéric": "Courant", "Jamy": "Gourmaud"}` → *retour :* `{"Quindou": "Sabine", "Courant": "Frédéric", "Gourmaud": "Jamy"}`

□

Modification de dictionnaires [Cours]

— Tout comme les listes ou les ensembles, les dictionnaires sont *mutables*, c.-à-d. altérables : leurs associations et le nombre de ces associations sont modifiables.

— Soit un dictionnaire `d`, une clef `k` définie ou non dans `d` et `v` une valeur, l'instruction d'assignation `d[k] = v` associe la valeur `v` à la clef `k`, écrasant le cas échéant l'association précédente. Par exemple :

```
d = {"France": "Paris", "United Kingdom": "London", "Brasil":
    "Rio de Janeiro"}

d["Italia"] = "Roma"
print(d) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London', 'Brasil': 'Rio de Janeiro', 'Italia': 'Roma'}".

d["Brasil"] = "Brasilia" # En 1960.
print(d) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London', 'Brasil': 'Brasilia', 'Italia': Roma}".
```

La méthode `update`, appelée sur un dictionnaire `d1` avec comme argument un dictionnaire `d2`, rajoute à `d1` les associations contenues dans `d2`, écrasant les éventuelles associations conflictuelles initiales. Par exemple :

```
d1 = {"France": "Paris", "United Kingdom": "London", "Brasil":
     "Rio de Janeiro"}
d2 = {"Italia": "Roma", "Brasil": "Brasilia"}

d1.update(d2)
print(d1) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London', 'Brasil': 'Brasilia', 'Italia': 'Roma'}".
```

— Il est possible d'effacer une association de clef `k` dans un dictionnaire `d` avec l'instruction `del d[k]`. Notons que l'exécution de cette instruction lève une exception si `k` n'est pas une clef définie dans `d`. Par exemple :

```
d = {"France": "Paris", "United Kingdom": "London", "Brasil":
    "Rio de Janeiro"}

del d["Brasil"]
print(d) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London'}".

del d["Brasil"] # KeyError
```

— Une autre manière d'effacer une association de clef `k` dans un dictionnaire `d` est d'utiliser l'instruction `d.pop(k)`. Si `k` est effectiment une clef de `k`, l'expression `d.pop(k)` est évaluée à la valeur qui lui est associée avant l'effacement de l'association ; sinon, une exception (`KeyError`) est levée.

— L'un des avantages de la méthode `pop` tient au fait qu'elle accepte optionnellement un second argument : l'instruction `d.pop(k, None)` efface dans `d` l'association de clef `k` et retourne la valeur correspondante si elle existe, et retourne `None` (sans erreur) sinon. Par exemple :

```
d = {"France": "Paris", "United Kingdom": "London", "Brasil":
    "Rio de Janeiro"}

d.pop("Brasil", None)
print(d) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London'}".

d.pop("Brasil", None)
print(d) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London'}".
```

Exercice 17 (Définir un dictionaire, ⋆⋆)
Écrire une fonction `associate` *prenant en argument deux listes* `keys` *et* `values`*, et retournant le dictionnaire associant chaque élément de* `values` *à l'élément de même indice dans* `keys` *si ces deux listes sont de même longueur et si les éléments de* `keys` *sont tous distincts, et* `None` *sinon.*

Contrat :
`keys, values` = `["e", "a", "z", "d"], [5, 1, 26, 4]` → *retour :* `{"e": 5, "a": 1, "z": 26, "d": 4}`
`keys, values` = `["e", "a", "z"], [5, 1, 26, 4]` → *retour :* `None`
`keys, values` = `["e", "a", "e", "z"], [5, 1, 26, 4]` → *retour :* `None`

□

Exercice 18 (Compter les occurrences, ⋆⋆)
Écrire une fonction `count` *prenant en argument une liste* `l` *et retournant un dictionnaire associant à chaque élément de* `l` *son nombre d'occurrences dans* `l`*.*

Contrat :
`l` = `["h", "e", "l", "l", "o"]` → *retour :* `{"h": 1, "e": 1, "l": 2, "o": 1}`

□

Exercice 19 (Ensemble des occurrences, ⋆⋆)
Écrire une fonction `positions` *prenant en argument une liste* `l` *et retournant un dictionnaire associant à chaque élément de* `l` *l'ensemble des indices de ses occurrences dans* `l`*.*

Contrat :
`l` = `["h", "e", "l", "l", "o"]` → *retour :* `{"h": {0}, "e": {1}, "l": {2, 3}, "o": {4}}`

□

4 À faire chez soi

Exercice 20 (Faux amis, ⋆⋆)
On dit de deux mots de deux langues différentes qu'ils sont des faux amis *si ces deux mots ont des sens clairement distincts mais s'orthographient de manière tellement similaire que l'on pourrait croire (à tort) qu'il s'agit de traductions l'un de l'autre. C'est le cas, par exemple, de « actually » en anglais et « actuellement » en français ; « actually » se traduit généralement par « en fait » et « actuellement » par « currently ». Un autre exemple est le cas de « eventually » en anglais et « éventuellement » en français, qui se traduisent généralement par « finalement » et « possibly », respectivement.*

Dans cet exercice, on va s'intéresser aux faux amis exacts, *qui sont des mots ayant exactement la même orthographe, comme « coin »/« coin » ou « figure »/« figure ».*

Écrire une fonction `faux_amis` *prenant en argument un dictionnaire* `src2tgt` *dont les paires clef-valeurs consistent en un mot dans une langue source et sa traduction dans une langue cible, et retournant l'ensemble de tous les* faux amis exacts *qui existent dans ce dictionnaire.*

Contrat :
`src2tgt` = `{"avion": "plane", "coin": "corner", "pièce": "coin", "zoo": "zoo"}` → *retour :* `{"coin"}`

□

Exercice 21 (Composition de dictionnaires, ⋆⋆)
Écrire une fonction `compose_dict` *prenant en argument deux dictionnaires* `d1` *et* `d2`*, et retournant le dictionnaire contenant les associations clef-valeurs* `k: v` *telles qu'il existe* `x` *tel que* `k: x` *est une association de* `d1` *et* `x: v` *est une association de* `d2`*.*

Contrat :
Soit les trois dictionnaires suivants,

```
fr2en = {"maison": "house", "rue": "road", "lac": "lake"}
en2de = {"house": "Haus", "road": "Strasse", "tower": "Turm"}
fr2de = {"maison": "Haus", "rue": "Strasse"}
```

`d1, d2` = `fr2en, en2de` → *retour :* `fr2de`

□

Exercice 22 (Associations clef-clef', ⋆⋆⋆)
Imaginons que l'on souhaite non plus enregistrer de simples associations clef-valeur comme dans un dictionnaire, où chaque clef est associée à exactement une valeur et où plusieurs clefs peuvent être associées à la même valeur, mais des associations « clef-clef' », dans le sens où si la clef `k_l` *est associée à la clef'* `k_r`*,* `k_l` *n'est associée à aucune autre clef' que* `k_r` *et aucune autre clef que* `k_l` *n'est associée à la clef'* `k_r`*.*

Pour représenter une telle structure de donnée, nous allons utiliser deux dictionnaires `d1` *et* `d2` *tels qu'une association clef-clef'* `k_l: k_r` *soit représentée par une association* `k_l: k_r` *dans* `d1` *ainsi qu'une association* `k_r: k_l` *dans* `d2`*.*

Écrire une fonction `change` *prenant en argument deux tels dictionnaires* `d1` *et* `d2`*, une clef* `k_l` *et une clef'* `k_r`*, et modifiant* `d1` *et* `d2` *de manière à enregistrer l'association* `k_l: k_r` *en écrasant le cas échéant toute association conflictuelle.*

Contrat :
`d1, d2, k_l, k_r` = `{}, {}, "a", 1` → `d1, d2` = `{"a": 1}, {1: "a"}`
`d1, d2, k_l, k_r` = `{"a": 1}, {1: "a"}, "b", 2` → `d1, d2` = `{"a": 1, "b": 2}, {1: "a", 2: "b"}`
`d1, d2, k_l, k_r` = `{"a": 1, "b": 2}, {1: "a", 2: "b"}, "c", 2` → `d1, d2` = `{"a": 1, "c": 2}, {1: "a", 2: "c"}`

d1, d2, k_l, k_r = {"a": 1, "c": 2}, {1: "a", 2: "c"}, "a", 2 → d1, d2 = {"a": 2}, {2: "a"}

□

Initiation à la programmation

SL25Y031

Cours/TD – Chapitre 8

Université Paris Cité

Objectifs :

- Tester l'existence d'un fichier ou dossier.
- Lister le contenu d'un dossier.
- Lire un fichier texte.
- Écrire dans un fichier texte.

1 Système de fichiers

Le système de fichiers [Cours]

- Les données enregistrées sur un support physique (ex : un disque dur, une clef usb) sont généralement représentées sous forme d'une structure constituée de *dossiers* (ou « répertoires ») et de *fichiers*.
- Un fichier peut contenir tout type de données alors que les dossiers ne servent a priori qu'à l'organisation des fichiers. Un dossier peut être vu comme une liste (éventuellement vide) de fichiers et de dossiers formant son *contenu*. Excepté un dossier particulier, la *racine du système de fichiers*, tout dossier et tout fichier est *contenu* dans un unique dossier, son *dossier parent*.
- Les *descendants* d'un dossier sont (par récurrence) :
  - tout fichier ou dossier qu'il contient ;
  - tout descendant d'un dossier qu'il contient ;
- La structure obtenue est *arborée* :
  - elle ne contient pas de boucle, dans le sens où un dossier n'est jamais l'un de ses propres descendants ;
  - il existe un dossier (la racine), dont tous les autres éléments descendent.
- Tout dossier et tout fichier possède un *nom*, représenté en Python par une chaîne de caractères (ex : `"documents"`, `"exam.pdf"`). Le nom de la racine correspond généralement à la chaîne vide et tous les autres éléments du système de fichiers ont des noms non vides. Les éléments contenus dans un même dossier doivent avoir des noms tous distincts.
- Il est courant pour un processus interagissant avec le système de fichiers (ex : un terminal, un navigateur de fichiers) de donner (temporairement ou non) un statut particulier à un dossier, alors appelé « répertoire de travail actuel ». Lorsque Python exécute du code, un répertoire de travail actuel est défini. Le choix initial de ce dossier dépend de la manière dont le code a été exécuté.

Chemins [Cours]

- Un *chemin* est une chaîne de caractères non vide désignant (ou « pointant vers ») un élément existant ou fictif dans un système de fichiers. Un chemin désignant un fichier existant peut par exemple servir à lire ce fichier. Un chemin désignant un élement fictif peut par exemple servir à créer un élement à cet emplacement.
- Pour interpréter un chemin, il est nécessaire de définir trois valeurs de type `str` (qui peuvent varier d'un système à un autre) :
  - le *séparateur* (`"/"` dans les exemples qui suivent) ;

  - l'*abréviation de répertoire courant* (`"."` dans les exemples qui suivent) ;
  - l'*abréviation de répertoire parent* (`".."` dans les exemples qui suivent).

  Pour interpréter certains chemins, dit « relatifs », il est aussi nécessaire d'avoir une notion de répertoire de référence. C'est généralement le répertoire de travail actuel qui va servir de référence pour l'interprétation des chemins relatifs en PYTHON.
- Quelques exemples :
  - le chemin `"/home/guest/documents"` désigne un élément `"documents"` situé dans le dossier `"guest"` situé dans le dossier `"home"` de la racine ;
  - le chemin `"./../../exam.pdf"` désigne l'élément `exam.pdf` situé dans le dossier parent du dossier parent du répertoire de référence ;
  - le chemin `"../../exam.pdf"` désigne exactement le même élément que le chemin précédent ;
  - le chemin `"/"` désigne la racine du système de fichiers.
- Formellement, pour interpréter un chemin, il faut le décomposer en fonction des occurrences du séparateur. Nous reviendrons plus bas sur la méthode `split`, mais `path.split(sep)` est une liste de chaînes de caractères : la liste des chaînes obtenues en découpant `path` au niveau de chaque occurrence de `sep`. Par exemple, `"./../../exam.pdf".split("/")` vaut `[".", "..", "..", "exam.pdf"]`, `"/home/guest/documents".split("/")` vaut `["", "home", "guest", "documents"]` et `"/".split("/")` vaut `["", ""]`.
- On peut définir l'élément désigné par un chemin `path` étant donné le séparateur `sep` par récurrence sur la liste `path.split(sep)` :
  - cas de base :
    - `[""]` désigne la racine,
    - `["."]` désigne le répertoire de référence,
    - `[".."]` désigne le parent du répertoire de référence ou le répertoire de référence s'il s'agit de la racine,
    - pour tout autre chaîne de caractères `s` ne contenant pas `sep`, `[s]` désigne l'élément de nom `s` contenu dans le répertoire de référence ;
  - si la liste `l` désigne un dossier `d`, alors :
    - `l + [""]` désigne encore `d`,
    - `l + ["."]` désigne encore `d`,
    - `l + [".."]` désigne le parent de `d` ou `d` s'il s'agit de la racine,
    - pour tout autre chaîne de caractères `s` ne contenant pas `sep`, `l + [s]` désigne l'élément de nom `s` contenu dans `d`.
- Les chemins commençant directement par le séparateur (ex : `"/home/guest/documents"`) sont *absolus* : leur interprétation ne dépend pas du répertoire de référence. Les autres chemins sont les chemins relatifs (ex : `"../../exam.pdf"`), dont l'interprétation dépend du répertoire de référence.

2 Lecture du système de fichiers

Lecture de l'arborescence de fichiers [COURS]

- Un grand nombre de fonctions et autres valeurs utiles pour la manipulation du système de fichiers sont disponibles en PYTHON dans le module (≈ la bibliothèque) `os`. On peut accéder à ces fonctions et valeurs après avoir exécuté l'instruction suivante :

```
1 import os
```

- Il est possible de connaître à tout moment le chemin absolu du répertoire de travail actuel (sous forme d'une chaîne de caractères) en évaluant l'expression `os.getcwd()` :

```
1 print(os.getcwd()) # Affiche "/home/guest".
```

- Le nom de la fonction `getcwd` est une abréviation de « *get current working directory* ».
- La valeur du séparateur est assignée à la variable `os.sep`. La valeur de l'abréviation de répertoire courant est assignée à la variable `os.curdir`. La valeur de l'abréviation de répertoire parent est

assignée à la variable `os.pardir`.

```
print(os.curdir) # Affiche ".".
print(os.pardir) # Affiche "..".
print(os.sep) # Affiche "/".
```

— Comme la valeur des trois chaînes précédentes peut varier d'un système à l'autre, afin que votre code soit portable (c.-à-d. puisse s'exécuter facilement sur différentes machines), je vous demande de toujours passer par ces trois noms de variable et non vers leurs valeurs directement (ce qui serait un exemple de « codage en dur »).

— En pratique, `os.sep` est surtout utilisée comme chemin absolu vers la racine. Pour construire des chemins à partir d'autres chemins ou noms de fichier/dossier, il est possible d'utiliser la fonction `os.path.join`. Par exemple :

```
path = os.sep # Désigne la racine.
print(path) # Affiche "/".
path = os.path.join(path, "home", "guest")
print(path) # Affiche "/home/guest".
```

```
path = os.curdir # Désigne le répertoire de travail actuel.
print(path) # Affiche ".".
path = os.path.join(path, "documents", os.pardir, "videos", "
    films")
print(path) # Affiche "./documents/../videos/films".
```

— Il est possible de lister le contenu d'un dossier à l'aide de la fonction `os.listdir`. Si `path` est un chemin pointant vers un dossier `d`, alors `os.listdir(path)` vaut la liste des noms des éléments contenus dans `d`. Par exemple, les instructions suivantes, qui sont équivalentes, affichent le contenu du répertoire de travail actuel :

```
for e_name in os.listdir(os.getcwd()): print(e_name)
```

```
for e_name in os.listdir(os.curdir): print(e_name)
```

Notons que `os.listdir` lève une exception ($\approx$ le programme plante) si son argument pointe vers un fichier (plutôt qu'un dossier) ou vers un élément fictif.

— La fonction `os.path.exists` permet de savoir si son argument est un chemin pointant vers un élément existant. La fonction `os.path.isfile`, elle, ne vaut `True` que si le chemin pointe vers un fichier, alors que la fonction `os.path.isdir` ne vaut `True` que si le chemin pointe vers un dossier. Par exemple, si le système de fichiers contient à sa racine un dossier `"home"`, contenant un dossier `"guest"`, contenant un fichier `"chap_6.pdf"` :

```
path = os.path.join(os.sep, "home", "guest", "chap_6.pdf")
print(os.path.exists(path)) # Affiche "True".
print(os.path.isfile(path)) # Affiche "True".
print(os.path.isdir(path)) # Affiche "False".

path = os.path.join(os.sep, "home", "guest")
print(os.path.exists(path)) # Affiche "True".
print(os.path.isfile(path)) # Affiche "False".
print(os.path.isdir(path)) # Affiche "True".
```

Exercice 1 (Afficher les fichiers, $\star$)
Écrire une suite d'instructions affichant les noms des fichiers (et non des dossiers) contenus dans le répertoire de travail courant. □

Exercice 2 (Afficher les dossiers, ⋆)
Écrire une suite d'instructions affichant les noms des dossiers (et non des fichiers) contenus dans le répertoire de travail courant. □

Lecture de fichiers [Cours]

— La fonction `open` permet d'ouvrir un fichier afin d'y lire ou écrire du contenu. Cette fonction renvoie une *interface d'entrée/sortie* (« IO wrapper ») possédant des méthodes de lecture/écriture de contenu. Pour éviter certains effets indésirables, il faut *fermer* toute interface ouverte dès qu'elle n'est plus utile, avec la méthode `close`.

— Pour lire un fichier texte, il est possible de l'ouvrir en fournissant à `open` deux arguments :

1. un chemin pointant vers ce fichier ;
2. la chaîne `"r"` (pour « read »).

Le deuxième argument de `open` est appelé « mode d'ouverture ».

— Notons que `open` lève une exception (≈ le programme plante) son premier argument n'est pas un chemin vers un fichier texte existant.

— Python associe à toute interface d'entrée/sortie une position dans le fichier appelée « position courante ». Lorsque l'on lit ou écrit dans un fichier via une interface, on y lit ou écrit à la position courante. Lorsqu'un fichier est ouvert en mode `r`, la position courante est initialisée au tout début du fichier.

— La méthode `read` appelée sur l'interface ainsi obtenue renvoie, sous forme d'une chaîne de caractères, tout le contenu du fichier *à partir de la position courante*. La position courante est aussi déplacée par `read`, à la fin du fichier. Par exemple, si `filename` représente un chemin pointant vers un fichier texte :

```
f = open(filename, "r") # Ouverture pour lecture. Position
    courante initialisée au début du fichier.

print(f.read(), end="") # Affiche le contenu du fichier. La
    position courante est déplacée à la fin du fichier.

f.close() # Fermeture.
```

— Noter que tout fichier texte non vide *correctement formé* inclut un caractère de fin de ligne `"\n"` final. Il existe donc une distinction entre un fichier de texte vide, dont le contenu est la chaîne vide `""`, et un fichier de texte contenant une unique ligne vide, dont le contenu est `"\n"`.

— La position courante étant déplacée par `read` à la fin du fichier, si l'on appelle deux fois de suite cette méthode sur la même interface, le deuxième appel retourne nécessairement la chaîne vide (`""` ; il s'agit bien du contenu du fichier se situant entre la position du courante, se trouvant alors être la fin du fichier, et la fin du fichier).

— Plutôt que d'avoir à explicitement fermer le gestionnaire d'entrée/sortie avec la méthode `close`, il est possible d'ouvrir le fichier avec une construction en `with`. Par exemple, les deux blocs de code suivants sont équivalents :

```
f = open(filename, "r")

print("[début de la lecture]")
print(f.read(), end="")
print("[fin de la lecture]")

f.close() # Fermeture.
```

```
with open(filename, "r") as f: # f est automatiquement fermé
    après l'exécution du bloc de code introduit.
  print("[début de la lecture]")
  print(f.read(), end="")
  print("[fin de la lecture]")
```

— La méthode `readline` permet de lire un fichier texte ligne par ligne. Si la position courante se trouve être la fin du fichier, `readline` ne fait rien et retourne la chaîne vide. Sinon, `readline` déplace la position courante jusqu'après le prochain caractère de fin de ligne `"\n"` et retourne tout le texte se situant entre les deux. Dans ce cas, `readline` renvoie donc une *ligne de texte*, c.-à-d. une chaîne de caractères se terminant par un caractère de fin de ligne, et ne contenant aucun autre caractère de fin de ligne. (On peut se rappeler que la fonction `input` aussi retourne une ligne de texte, mais provenant d'une saisie clavier et non d'un fichier texte.) Par exemple, si `filename` représente un chemin pointant vers un fichier constitué d'exactement trois lignes de texte :

```
with open(filename, "r") as f:
  print(f.readline(), end="") # Affiche la première ligne du fichier.
  print(f.readline(), end="") # Affiche la seconde ligne du fichier.
  print(f.readline(), end="") # Affiche la troisième ligne du fichier.
  print(f.readline(), end="") # Aucun effet.
  print(f.readline(), end="") # Aucun effet.
```

— La méthode `readlines` fonctionne de manière similaire à `read`, dans le sens où elle lit le fichier de la position courante jusqu'à la fin et y déplace la position courante. Cependant, `readlines` ne retourne pas le contenu lu sous forme d'une unique chaîne de caractères, mais d'une liste de lignes de texte (chacune se terminant par le caractère de fin de ligne). Par exemple, si `filename` représente un chemin pointant vers un fichier constitué d'exactement trois lignes de texte :

```
with open(filename, "r") as f:
  l1 = f.readlines() # Liste de trois chaînes de caractères.
  print(len(l1)) # Affiche "3".
  l2 = f.readlines() # Liste vide.
  print(len(l2)) # Affiche "0".
```

Exercice 3 (Manipulation des méthodes de base, ⋆)
Considérer que `"/home/guest/documents/dialog.txt"` pointe vers un fichier texte contenant exactement les deux lignes suivantes (incluant les tirets) :

```
- Bonjour, comment allez-vous ?
- Très bien, et vous ?
```

Décrire très précisément (c.-à-d. notamment sans oublier les éventuelles lignes vides) l'affichage produit par l'exécution de chacune des suites d'instructions suivantes.

1.
```
import os
with open(os.path.join(os.sep, "home", "guest", "documents", "dialog.txt") as f:
  print(f.read())
  print(f.readline())
  print(f.readlines())
```

2.
```
import os
with open(os.path.join(os.sep, "home", "guest", "documents", "dialog.txt") as f:
  print(f.readline())
  print(f.readlines())
  print(f.read())
```

□

Exercice 4 (Réimplémentation de readlines, ⋆⋆)
Écrire une fonction `readlines` *réimplémentant la méthode du même nom, c.-à-d. prenant en argument* `f`*, le gestionnaire d'entrée/sortie obtenu à l'ouverture d'un fichier texte en lecture, et retournant la liste de toutes les lignes qui restent à y être lues. La fonction ne doit pas utiliser la méthode* `readlines` *mais plutôt* `readline`*.* □

Exercice 5 (Affichage de certaines lignes, ⋆⋆)

1. *Écrire une fonction* `f` *prenant en argument une chaîne de caractères* `filename`*, supposée être un chemin pointant vers un fichier texte (existant), et affichant (sans saut de ligne supplémentaire) chacune des lignes ne commençant pas par* `"#"`*.*
2. *Modifier la fonction afin que rien ne se passe (en particulier, pas d'erreur) si l'argument* `filename` *passé ne pointe pas vers un fichier existant.*

□

Lecture de fichier et chargement en mémoire [Cours]

— Il peut être plus pratique d'utiliser `readlines` que `readline`. Cependant, alors qu'un appel à `readline` ne provoque la lecture et mise en mémoire que d'au plus une ligne de texte, un appel à `readlines` (comme à `read`) provoque la lecture et mise en mémoire de tout le contenu du fichier à partir de la position courante. Dans certaines situations, typiquement parce que ce contenu est trop volumineux ou parce que seule une ligne est utile, il est préférable voire nécessaire d'utiliser `readline`.

— Il est utile d'insister sur le fait que lorsqu'une fonction telle que `readline` lit une ligne vide, celle-ci est matérialisée par la chaîne `"\n"`. Cette chaîne est donc différente de la chaîne vide (`""`), retournée lorsque la position courante est déjà à la fin du fichier.

Exercice 6 (Fichier vide ?, ⋆)
Écrire une fonction `isEmpty` *prenant en argument une chaîne de caractères* `filename`*, supposée être un chemin pointant vers un fichier texte (existant), et retournant* `True` *si ce fichier est vide,* `False` *sinon. Écrire une fonction ne nécessitant pas forcément la lecture de tout le fichier texte.* □

Fonctions utiles au traitement de fichiers texte [Cours]

— La méthode `lstrip` (« left strip ») appelée sans argument sur une chaîne de caractères `s` retourne la chaîne de caractères obtenue en supprimant de `s` tous les caractères d'espacement (ex : espace, tabulation) et de saut de ligne situés *à gauche* du premier caractère autre (c.-à-d. qui n'est ni un caractère d'espacement ni un saut de ligne). La méthode `rstrip` (« right strip ») a un comportement similaire mais supprime les caractères situés *à droite* du dernier caractère autre. Par exemple :

```
s1 = " \t\t  \n   Bonjour. Comment-allez vous ?   \t    \n"
s2 = s1.lstrip() # "Bonjour. Comment-allez vous ?   \t    \n"
s3 = s1.rstrip() # " \t\t  \n   Bonjour. Comment-allez vous ?"
```

— La méthode `strip` peut être vue comme une combinaison de `rstrip` et de `lstrip`. Appelée sans argument sur une chaîne de caractères `s`, elle retourne la chaîne de caractères obtenue en supprimant de `s` tous les caractères d'espacement et de saut de ligne situés à gauche du premier caractère autre et tous ceux situés à droite du dernier caractère autre. Par exemple :

```
s1 = " \t\t  \n   Bonjour. Comment-allez vous ?   \t    \n"
s2 = s1.strip() # "Bonjour. Comment-allez vous ?"
```

— Les méthodes `strip`/`rstrip` sont particulièrement utiles pour éliminer le caractère `"\n"` situé à la fin d'une ligne lue depuis un fichier texte.

— Par défaut, ces trois méthodes suppriment les caractères d'espacement et de saut de ligne, mais il est possible de spécifier l'ensemble des caractères à supprimer en leur passant comme argument une chaîne de caractères composée des caractères à supprimer. Par exemple :

```
s1 = "aabbabaccabcdddbaaab"
s2 = s1.strip("ab") # "ccabcddd"
s3 = s1.strip("abcd") # ""
```

— La méthode `split` appelée sur une chaîne de caractères `s` avec pour argument une chaîne de caractères `sep` retourne la liste de chaînes de caractères obtenue en découpant `s` au niveau de chaque occurrence de `sep`. Par exemple :

```
s = "bloup--blip-bloup--blop"
l = s.split("--") # ["bloup", "blip-bloup", "blop"]
```

```
s = "bloup--blip-bloup--blop"
l = s.split("-") # ["bloup", "", "blip", "bloup", "", "blop"]
```

— Il faut garder en tête que l'argument de `split` est interprété de manière très différente de l'argument de `strip` et ses variantes :
  — `s.split(sep)` découpe `s` au niveau de chaque occurrence (complète) de `sep`.
  — `s.strip(chars)` retourne une chaîne obtenue en supprimant de `s` des occurrences, non pas de `chars`, mais de n'importe quel caractère présent dans `chars`.

— La méthode `split` est la réciproque de `join`, dans le sens où si `s` et `sep` sont deux chaînes de caractères, `sep.join(s.split(sep))` vaut `s`.

— La méthode `split` est utile pour séparer (approximativement) les différents *tokens* (c.-à-d. occurrences de mots) d'une phrase en français. Par exemple :

```
s = "Le chat court après la souris."
l = s.split(" ") # ["Le", "chat", "court", "après", "la", "
    souris."]
```

— La méthode `split` accepte un argument supplémentaire indiquant un nombre maximum de coupes à effectuer. La méthode `split` appelée sur une chaîne de caractères `s` avec pour argument une chaîne de caractères `sep` et un entier `n` retourne la liste de chaînes de caractères obtenue en découpant `s` au niveau des **`n` premières occurrences de `sep`**. Par exemple :

```
s = "Le chat court après la souris."
l = s.split(" ", 3) # ["Le", "chat", "court", "après la souris
    ."]
l = s.split(" ", 0) # ["Le chat court après la souris."]
```

Exercice 7 (Découpage de noms, ⋆⋆)

1. *Écrire une fonction* `split_name` *prenant en argument une chaîne de caractères* `name` *supposée contenant un unique espace, et retournant la paire (un* `tuple`*) composée des deux sous-chaînes de* `name` *obtenues en la découpant au niveau de cet espace.*

   Contrat :
   `name = "George Sand"` → *retour :* `("George", "Sand")`

2. *Modifier la fonction proposée à la question précédente pour qu'elle renvoie* `None` *si* `name` *ne contient pas un unique espace.*

   Contrat :
   `name = "George Sand Michael"` → *retour :* `None`
   `name = "George"` → *retour :* `None`
   `name = "George Sand"` → *retour :* `("George", "Sand")`

3. *Modifier la fonction proposée à la question précédente pour qu'elle ne renvoie plus une paire* `(s1, s2)` *lorsque* `name` *contient un unique espace mais un dictionnaire de clefs* `"prénom"` *et* `"nom"`, `{"prénom": s1, "nom": s2}`.

   Contrat :
   `name = "George Sand"` → *retour :* `{"prénom": "George", "nom": "Sand"}`

□

Exercice 8 (Création de vocabulaire, ⋆⋆)
Écrire une fonction `words_set` *prenant en argument une chaîne de caractères* `filename` *supposée représenter le chemin d'un fichier texte, et retournant l'ensemble des mots apparaissant dans ce fichier. (Supposer que tous les tokens sont séparés par des espaces.)* □

3 Écriture du système de fichiers

Écriture de fichiers [Cours]

— Si l'on appelle la fonction `open` avec comme arguments
  — un chemin pointant vers un fichier fictif mais situé dans dossier existant et
  — le mode `"w"` (« write »),
  alors le fichier pointé par le chemin est créé (avec un contenu vide) et une interface d'entrée/sortie permettant d'y écrire du contenu est retournée. Par exemple, si le répertoire de travail actuel contient un dossier `"documents"` qui lui ne contient pas de fichier `"test.txt"`, alors l'exécution des instructions suivantes crée un tel fichier, entièrement vide :

```
with open(os.path.join(os.curdir, "documents", "test.txt"), "w
    ") as f: # Ouverture pour écriture.
  print("Empty file created.")
```

— Si le chemin passé comme premier argument à `open` avec le mode `"w"` pointe vers un fichier existant, alors ce fichier est *écrasé*, c.-à-d. que son contenu est effacé.
— Dans tous les cas, l'interface d'entrée/sortie retournée par `open` en mode `"w"` correspond donc à un fichier vide ; la position courante est définie au début du fichier, qui se trouve être aussi la fin.
— La méthode `write`, appelée sur une interface ouverte en écriture et avec pour argument une chaîne de caractères `s`, écrit `s` à partir de la position courante et déplace celle-ci jusqu'après le dernier caractère inséré. Par exemple, après exécution des instructions suivantes, le fichier pointé par `filename` contient exactement la ligne de texte `"Hello world!\n"` :

```
with open(filename, "r") as f: # Ouverture pour écriture.
  f.write("Hell")
  f.write("o world!")
  f.write("\n")
```

— Contrairement à `print` qui, par défaut, affiche à l'écran un saut de ligne supplémentaire après son argument, la méthode `write` n'écrit dans un fichier que les sauts de ligne explicitement contenus dans son argument. Par exemple, après exécution des instructions suivantes, le fichier pointé par `filename` n'est pas un fichier texte correctement formé car son contenu ne se termine pas par un caractère de saut de ligne :

```
with open(filename, "r") as f: # Ouverture pour écriture.
  f.write("Hello world!")
```

— Il est possible d'utiliser `open` avec le mode `"a"` (« append »). Ce mode fonctionne comme `"w"` dans le cas où le chemin pointe vers un fichier inexistant, mais n'écrase pas le fichier s'il existe. Dans ce dernier cas, le contenu du fichier n'est pas modifié et la position courante est initialisée à la fin du fichier. Par exemple, si `filename` pointe vers un fichier contenant une ligne `"Hello\n"`, l'exécution des instructions suivantes y ajoutent une seconde ligne `"word!\n"` :

```
with open(filename, "a") as f: # Ouverture pour écriture.
  f.write("world!\n")
```

Pour aller plus loin [Cours]

— Il est possible de créer des répertoires avec les méthodes `os.makedirs` et `os.mkdir`.
— Il est possible de télécharger des fichiers avec la méthode `urllib.request.urlretrieve`.

4 À faire chez soi

Exercice 9 (Parcours d'une branche, ⋆⋆⋆)
Écrire une fonction `listdir_rec` *prenant en argument une chaîne de caractères* `path` *et (i) retournant* `True` *et affichant les chemins de tous les fichiers descendants du dossier pointé par* `path` *si un tel dossier existe, et (ii) retournant* `False` *(sans rien afficher) sinon. La fonction doit être récursive, c.-à-d. que son fonctionnement repose sur le fait qu'un appel à* `listdir_rec` *apparaît dans le corps de* `listdir_rec` *elle-même.* □

Exercice 10 (Premières lignes, ⋆⋆)

1. *Écrire une fonction* `f` *prenant en argument une chaîne de caractères* `filename`*, supposée être un chemin vers un fichier texte, et un entier positif* `n`*, et affichant (sans saut de ligne supplémentaire) les* `n` *premières lignes de ce fichier (ou moins si le fichier en contient moins).*
2. *Modifier la fonction de telle sorte que si l'argument* `n` *passé est strictement négatif, toutes les lignes soient lues.*

□

Exercice 11 (Création de chemins par jointure (I), ⋆⋆)
Dans cet exercice, on cherche à implémenter des fonctions implémentant de manière plus ou moins simplifiée le comportement de `os.path.join`*.*

1. *Écrire une fonction* `myJoin` *prenant en argument une liste de chaînes de caractères* `l` *et retournant la chaîne obtenue en concaténant, séparés par des occurrences de* `os.sep`*, les élements non vides de* `l`*. La fonction ne doit pas utiliser* `os.path.join`*, mais peut utiliser* `join`*.*

   Contrat :
   (En supposant que `os.sep = "/"`*.)*
   `l = ["..", "documents"]` → *retour :* `"../documents"`
   `l = ["a/b", ".", "c"]` → *retour :* `"a/b/./c"`
   `l = ["..", "", "documents", ""]` → *retour :* `"../documents"`
   `l = ["..", "", "documents", "", ""]` → *retour :* `"../documents"`
2. *Récrire* `myJoin` *de manière à ce que si un élément de l'argument* `l` *vaut ou commence par* `os.sep`*, tous les éléments précédents soient ignorés. Veiller à bien généraliser le comportement illustré dans le contrat, impliquant probablement de gérer les éléments égaux à* `os.sep` *différement des autres éléments commençant par* `os.sep`*.*

   Contrat :
   (En plus du contrat de la question précédente et toujours en supposant que `os.sep = "/"`*.)*
   `l = ["/", "home", "guest"]` → *retour :* `"/home/guest"`
   `l = ["/home", "guest"]` → *retour :* `"/home/guest"`
   `l = ["..", "/", "tmp", "/", "home", "guest"]` → *retour :* `"/home/guest"`
   `l = ["..", "/", "tmp", "/home", "guest"]` → *retour :* `"/home/guest"`

□

Exercice 12 (Comptage d'occurrences, ⋆⋆)
Écrire une fonction `words_count` *prenant en argument une chaîne de caractères* `filename` *supposée être un chemin pointant vers un fichier texte, et retournant le dictionnaire associant à chaque mots apparaissant dans ce fichier son nombre d'occurrences. (Supposer que tous les tokens sont séparés par des espaces.)* □

Exercice 13 (Découpage, ⋆⋆⋆)
Écrire une fonction `split` *réimplémentant la méthode du même nom pour les séparateurs constitués d'un unique caractère, c.-à-d. prenant en argument une chaîne de caractères* `s` *et un caractère* `sep`*, et retournant la liste de chaînes de caractères obtenue en découpant* `s` *au niveau de chaque occurrence de* `sep`*. La fonction ne doit pas utiliser la méthode* `split`*. (Bien vérifier sur machine que la fonction proposée satisfait le contrat.)*

Contrat :
`s, sep = "abc.de..fgh."` → *retour :* `["abc", "de", "", "fgh", ""]`
`s, sep = "abc.de..fgh"` → *retour :* `["abc", "de", "", "fgh"]`

□

Exercice 14 (Création de chemins par jointure (II), ⋆⋆⋆)
Écrire une fonction `myJoin` *réimplémentant* `os.path.join`*, à l'unique différence près qu'alors que* `os.path.join` *accepte n'importe nombre d'arguments de type* `str`*,* `myJoin` *doit accepter un unique argument* `l`*, une liste de chaînes de caractères. Attention,* `os.path.join` *a un comportement qui n'est pas facile à décrire ; étudier attentivement le contrat et ne pas hésiter à faire d'autres tests sur machine pour comprendre précisément la valeur retournée par* `os.path.join`*. La fonction ne doit pas utiliser* `os.path.join`*, mais peut utiliser* `join`*. Indice : Construire une liste de chaînes de caractères* `tmp` *et retourner* `"".join(tmp)`*.*

Contrat :
(En supposant que `os.sep = "/"`*.)*

`l = ["..", "documents"]`	→	*retour :* `"../documents"`
`l = ["a/b", ".", "c"]`	→	*retour :* `"a/b/./c"`
`l = ["..", "", "documents", ""]`	→	*retour :* `"../documents/"`
`l = ["..", "", "documents", "", ""]`	→	*retour :* `"../documents/"`
`l = ["/", "home", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/", "home", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/home", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/home/", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/home//", "guest"]`	→	*retour :* `"/home/guest"`

□