# 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"}

□