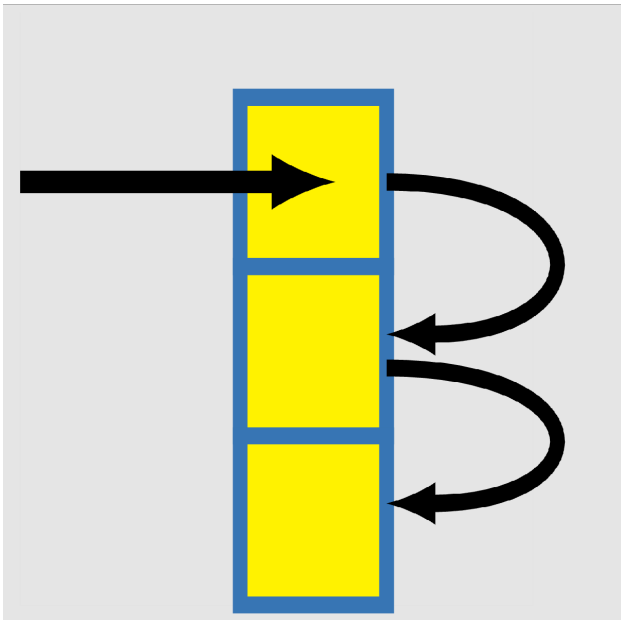
Pascal ORTIZ



Structures de données

Version du 11 novembre 2023

Table des matières

Chapitre I

Dictionnaires

1 L'essentiel

Les dictionnaires : présentation

Supposons qu'on ait une liste d'individus et qu'on veuille stocker leur âge. On souhaiterait disposer d'une structure de données qui à partir du nom d'un individu, donnerait son âge. On souhaiterait en outre que l'accès à l'âge soit « instantané » ou tout du moins rapide, un peu comme l'accès à un élément d'une liste est immédiat à partir de son indice. Un telle structure de données existe et s'appelle un *dictionnaire* ou encore un *mapping*. Le langage Python dispose nativement de dictionnaires.

Voici un exemple de création et d'utilisation de dictionnaire en Python :

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
2
3 print(d["thierry"])
4 print(d["jeanne"])
```

```
5 40
6 18
```

— Ligne 1 : `d` est le dictionnaire ; il associe des prénoms à un âge.

— Lignes 2 et 3 : on accède en lecture à l'âge de `thierry` puis à l'âge de `jeanne`

Un dictionnaire répertorie des items. Un item est une paire formée d'une clé et de la valeur correspondante de cette clé. Dans l'exemple ci-dessus, la paire `("thierry", 40)` est un item, `"thierry"` est une clé du dictionnaire `d` et `40` est la valeur de la clé `"thierry"`. Le dictionnaire ci-dessus contient 5 clés. Ici, les clés sont du type chaîne et les valeurs sont de type entier.

Non seulement les dictionnaires sont un type de données (au-delà de tout langage de programmation) mais, en Python, ils définissent un type, le type `dict` :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }

print(type(d))
```

```
<class 'dict'>
```

Lien vers la documentation officielle des dictionnaires : Mapping Types: dict.

Situations d'utilisation d'un dictionnaire

Les dictionnaires sont très utiles et très courants d'emploi. Voici, en vrac, quelques situations où un dictionnaire peut s'avérer utile :

— ville ⟶ température
— nom ⟶ âge
— nom ⟶ taille
— nom ⟶ (âge, taille, téléphone)
— polynôme (polynôme creux plus précisément) : exposant ⟶ coefficient
— dictionnaire français-anglais
— nom ⟶ surnom
— valeur dans une liste ⟶ fréquence
— pour coder les exceptions du pluriel : "bijou" ⟶ "bijoux"
— codage en décalage dit en rot13 : "A" ⟶ "N", "B" ⟶ "O", etc
— émulation d'une structure des langages C ou C++

mais en réalité, les situations sont **innombrables**.

Dictionnaire littéral

On peut créer un dictionnaire en Python en plaçant les items entre accolades, séparés par des virgules. Chaque item est entré en plaçant successivement :

— la clé,
— le séparateur deux-points
— la valeur de la clé.

Exemple :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
```

On crée ainsi un dictionnaire qui est un objet Python et auquel on peut se référer en utilisant une variable, ici `d`. Un dictionnaire créé de cette façon dans le code-source est appelé un *dictionnaire littéral*.

Dans un dictionnaire, une clé ou une valeur étant des objets Python, elles peuvent être définies par des expressions utilisant des variables :

```
1 p = "paul"
2 age = 25
3
4 d = {p: age, "jean" : 70, "thierry" : 40, "léa" : age - 10, "jeanne" : 18 }
5
6 print(d[p])
7 print(d["léa"])
```

```
8 25
9 15
```

— Ligne 4 : la clé et la valeur du premier item est sont référencés par des variables définies précédemment.
— Ligne 4 : la valeur de la clé `"léa"` est celle d'une expression dépendant de la variable `age`.

Voir plus bas ce qu'il se passe si deux clés sont identiques dans le dictionnaire littéral.

Facilité d'édition d'un dictionnaire littéral

Un dictionnaire littéral peut être édité avec certaines facilités de formatage afin d'en améliorer la lisibilité dans le code-source.
Par exemple, soit le dictionnaire :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
```

Il est équivalent de le saisir dans le code-source Python de la manière suivante :

```
d = {
    "paul"      : 25,
    "jean"      : 70,
    "thierry"   : 40,
    "léa"       : 15,
    "jeanne"    : 18
    }
```

Les sauts de lignes entre les accolades ont permis d'aligner les clés entre elles ainsi que les valeurs correspondantes : le dictionnaire est plus lisible.

Affichage d'un dictionnaire

Si `d` est un dictionnaire, `print(d)` affiche `d` sous forme de dictionnaire littéral :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }

print(d)
```

```
{'paul': 25, 'jeanne': 18, 'thierry': 40, 'jean': 70, 'léa': 15}
```

Le dictionnaire est affiché sous forme littérale. Depuis Python 3.7, l'ordre d'affichage des items correspond toujours à l'ordre d'insertion des items dans le dictionnaire.

Unicité de chaque clé

Dans un dictionnaire, chaque clé est unique, autrement dit, deux items distincts ont des clés distinctes.
Dans un dictionnaire littéral, si on place plusieurs items ayant même clé, c'est le dernier item qui est pris en compte, les précédents items ayant ce nom sont ignorés :

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15,
2     "thierry" : 45, "jeanne" : 18 }
```

```
3 print(d["thierry"])
```

```
4 45
```

— Lignes 1 et 2 : la clé `"thierry"` est présente deux fois.

— Lignes 2 et 4 : c'est la deuxième occurrence, celle qui porte la valeur 45, qui est prise en compte.

Longueur d'un dictionnaire

Un dictionnaire est un conteneur de clés et il contient donc un certain nombre d'éléments. Ce nombre est obtenu en utilisant la fonction `len` :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40,
     "léa" : 15, "thierry" : 45, "jeanne" : 18 }

print(len(d))
```

```
5
```

— le nombre d'éléments du dictionnaire. On observera que les deux entrées `"thierry" : 40` et `"thierry" : 45` ne sont comptées qu'une seule fois.

La longueur d'un dictionnaire est aussi son nombre de clés distinctes.

Dictionnaire vide

Un dictionnaire peut être vide :

```
d = {}
print(len(d))
```

```
0
```

Un dictionnaire vide trouve son intérêt, il peut être le conteneur de base d'un dictionnaire qui va grossir.

Lecture d'une valeur d'un dictionnaire

Un dictionnaire permet d'accéder à partir d'une de ses clés à la valeur de l'item. Pour lire cette valeur, on utilise l'indexation par la clé, avec une paire de crochets, comme pour une liste :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }

print(d["thierry"])
print(d["jeanne"])
```

```
40
18
```

Un dictionnaire est un peu comme une liste mais indexée par des clés et non par des entiers.

Clé absente

Si on tente d'accéder à une clé qui ne figure pas dans le dictionnaire, on déclenche une erreur :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }

print(d["pascal"])
```

```
KeyError: 'pascal'
```

— Erreur sur la clé (`KeyError`) : la clé `"pascal"` n'existe pas dans le dictionnaire `d`.

Appartenance à un dictionnaire

Le point suivant est un des éléments qui font tout l'intérêt des dictionnaires. Il est possible de tester la présence dans un dictionnaire d'une clé avec l'opérateur d'appartenance `in` :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }

print("pascal" in d)
```

```
False
```

L'opérateur `not in` permet de tester la non-appartenance :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }

print("pascal" not in d)
```

```
True
```

Le type des clés

Les clés d'un dictionnaire sont fréquemment des entiers ou des chaînes de caractères. Toute clé d'un dictionnaire est un objet Python, et donc, toute clé a un type [1].

Clés hachables

Toutefois, certains types ne peuvent représenter des clés de dictionnaire. Par exemple, une clé ne peut être de type liste :

cle_invalide.py

```
d = {"paul" : 25, ["thierry", "pierre"]: 40}
print(d)
```

```
TypeError: unhashable type: 'list'
```

Une clé de dictionnaire doit avoir un type dit « hachable ». C'est le cas pour les types entier, chaîne de caractères et tuple.

1. Les clés d'un même dictionnaire ne sont pas forcément toutes du même type même si c'est assez fréquent en pratique.

Ainsi, on peut transformer l'exemple `cle_invalide.py` pour obtenir un dictionnaire valide en utilisant un tuple plutôt qu'une liste :

```
d = {"paul" : 25, ("thierry", "pierre"): 40}
print(d)
```

```
{'paul': 25, ('thierry', 'pierre'): 40}
```

— `d` possède deux clés : une chaîne et un tuple de deux chaînes.

100% **hachable**

Tout objet de type tuple, néanmoins, n'est pas forcément hachable. Si le tuple contient, directement ou indirectement, un objet de type non-hachable, le tuple sera non hachable :

```
t = (42, [421, 100])
d = {5 : 10, t: 10}
```

```
TypeError: unhashable type: 'list'
```

— `t` est un tuple mais un des membres de ce tuple est une liste qui est un type non hachable

— Il en résulte que `t` ne peut figurer comme clé d'un dictionnaire.

En revanche, il n'y a aucune restriction sur les types des valeurs des clés.

Modifier un dictionnaire

Étant donné un dictionnaire `d`, il est possible :

— de modifier la valeur d'une clé déjà présente ;

— d'ajouter des items au dictionnaire.

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
2
3 print(d["paul"])
4 d["paul"]=26
5 print(d["paul"])
6
7 d["pascal"]=50
8 print(d)
```

```
9 25
10 26
11 {'paul': 26, 'jean': 70, 'thierry': 40, 'léa': 15, 'jeanne': 18, 'pascal': 50}
```

— Ligne 4 : on écrase la valeur de la clé "paul"

— Ligne 7 : on ajoute l'item `"pascal : 50`.

On observera que c'est la même syntaxe [2] qui est utilisée pour à la fois

— modifier la valeur d'une clé

2. la syntaxe avec l'opérateur d'indexation `[ ]`.

— placer une nouvelle clé.

Les modifications ci-dessus traduisent qu'un dictionnaire est, comme une liste, un objet **mutable**.

Parcours de dictionnaire

Un dictionnaire est un conteneur. Il est possible de parcourir ce conteneur pour examiner les clés ainsi que les valeurs correspondantes.

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18}
2 for k in d:
3     print(k, "->", d[k])
```

```
4 paul -> 25
5 jean -> 70
6 thierry -> 40
7 léa -> 15
8 jeanne -> 18
```

Le parcours du dictionnaire s'effectue à l'aide d'une boucle `for`. Pour parcourir le dictionnaire, on parcourt les clés (cf. ligne 2), ce qui permet d'accéder à la valeur associée à la clé.

L'ordre de parcours du dictionnaire est celui de l'ordre d'insertion des clés dans le dictionnaire.

Dictionnaire en compréhension

Soit par exemple à créer un dictionnaire dont les items aient pour clé chacun des 8 chiffres romains

I, V, X, L, C, D, M

en sorte que la valeur de chaque item soit la valeur numérique correspondant au chiffre romain, par exemple, la valeur de l'item **X** est 10.

Cette question est commodément résolue en utilisant la syntaxe des dictionnaires en compréhension :

```
1 romains="IVXLCDM"
2 valeurs=[1, 5, 10, 50, 100, 500, 1000]
3
4 d = {romains[i] : valeurs[i] for i in range(len(romains))}
5 print(d)
```

```
6 {'C': 100, 'D': 500, 'I': 1, 'M': 1000, 'L': 50, 'V': 5, 'X': 10}
```

— Ligne 4 : la syntaxe du membre de droite est celle d'un dictionnaire en compréhension. L'indice i parcourt tous les indices possibles des éléments de la chaîne romains (les chiffres romains). Ici, `romains[i]` est la clé de l'item (un chiffre romain) et la valeur correspondante de l'item est `valeurs[i]` qui est l'entier qui correspond à la valeur représentée par le chiffre romain, par exemple "X" représente 10.

— Ligne 4 : pour simplifier l'exposé, ce n'est pas la façon considérée comme pythonique de parcourir simultanément 2 itérables qui a été utilisée.

La syntaxe du dictionnaire en compréhension est calquée sur celle des listes en compréhension. En particulier, il est possible d'utiliser une clause `if`.

Garnir un dictionnaire à partir d'un dictionnaire vide

Note : les situations décrites ci-après sont souvent mieux traitées par utilisation d'un dictionnaire en compréhension ou d'autres méthodes de création de dictionnaire.

On peut créer un dictionnaire en remplissant item par item un dictionnaire au départ vide :

```
d={}
d["paul"] = 25
d["jean"] = 70
d["thierry"] = 40
d["léa"] =15
d["thierry"] = 45
d["jeanne"] = 18
print(d)
```

```
{'paul': 25, 'jean': 70, 'thierry': 45, 'léa': 15, 'jeanne': 18}
```

mais, présenté comme ci-dessus, cela a peu d'intérêt.

Usage d'une boucle

En pratique, les items sont souvent entrées dans le dictionnaire à l'aide d'une boucle. Par exemple, soit la liste des cinq voyelles `"A"`, `"E"`, `"I"`, `"O"`, `"U"`, `"Y"` et soit le dictionnaire dont les items sont les cinq voyelles et la valeur de l'item, le rang de la voyelle (`"A"` est la voyelle n°1, `"E"` la voyelle n°2, etc) :

```
voyelles = "AEIOUY"
d={}
i=1
for v in voyelles:
    d[v] = i
    i = i + 1
print(d)
```

```
{'A': 1, 'E': 2, 'I': 3, 'O': 4, 'U': 5, 'Y': 6}
```

L'exemple ci-dessus serait avantageusement traité par un dictionnaire en compréhension :

```
d = {v:i for i, v in enumerate(voyelles, 1)}
print(d)
```

```
{'A': 1, 'E': 2, 'I': 3, 'O': 4, 'U': 5, 'Y': 6}
```

Une autre possibilité est d'utiliser `dict` :

```
d = dict(zip(voyelles, range(1, len(voyelles)+1)))
print(d)
```

```
{'A': 1, 'E': 2, 'I': 3, 'O': 4, 'U': 5, 'Y': 6}
```

Création de dictionnaire avec `dict`

Il existe d'autres procédés de création de dictionnaire que la construction d'un dictionnaire littéral [3]. On peut construire un dictionnaire avec le type `dict`. Le processus est expliqué à l'exemple suivant :

```
1 mes_items = [["paul", 25], ["jeanne", 18],
2              ["thierry", 45], ["jean", 70], ["léa", 15]]
3 d = dict(mes_items)
4 print(d)
```

```
5 {'paul': 25, 'jeanne': 18, 'thierry': 45, 'jean': 70, 'léa': 15}
```

— Lignes 1-2 : pour créer un dictionnaire formé de clés et des valeurs correspondantes, on écrit chaque item comme une liste de deux éléments (clé et valeur) et on place tous ces items dans une liste.

— Ligne 3 : on appelle ensuite le constructeur `dict` sur cette liste. L'appel renvoie alors le dictionnaire souhaité.

`dict` est souvent appelé un *contructeur* puisqu'il permet de construire des objets d'une certaine catégorie.

Supprimer une clé d'un dictionnaire

Un dictionnaire étant mutable une clé d'un dictionnaire peut être supprimée : il suffit d'utiliser l'instruction `del` appliquée à la clé à supprimer :

```
1 d = {"paris" : 30, "madrid" : 35, "oslo" : 20}
2 print(d)
3
4 del d["oslo"]
5 print(d)
```

```
6 {'paris': 30, 'madrid': 35, 'oslo': 20}
7 {'paris': 30, 'madrid': 35}
```

— Ligne 4 : supression de la clé `"oslo"` du dictionnaire `d`.

— Ligne 7 : le dictionnaire `d` a bien perdu une clé.

`del` supprime une référence

En réalité, l'instruction `del` ne supprime la clé **que** du dictionnaire, autrement dit, `del` ne supprime que la référence que le dictionnaire contient vers l'item. La clé et la valeur peuvent très bien encore exister en mémoire :

3. Heureusement, sinon on ne pourrait pas créer, par exemple, de dictionnaire ayant un très grand nombre de clés.

```
1 oslo = "oslo"
2 temp_oslo = 20
3
4 d = {"paris" : 30, "madrid" : 35, oslo : temp_oslo}
5 print(d)
6
7 del d[oslo]
8 print(d)
9 print(oslo, temp_oslo)
```

```
10 {'paris': 30, 'madrid': 35, 'oslo': 20}
11 {'paris': 30, 'madrid': 35}
12 oslo 20
```

— Ligne 4 : la clé `"oslo"` est référencée via la variable `oslo` ; de même, la valeur de la clé dans le dictionnaire est référencée par la variable `temp_oslo`.

— Lignes 7 et 11 : la clé `"oslo"` de `d` est supprimée.

— Lignes 9 et 12 : les objets que référençaient la clé et la valeur de la clé subsistent.

Clé absente

Si on essaye de supprimer un item d'un dictionnaire `d` par une clé qui ne figure pas dans `d`, une erreur se produit :

```
1 d = {"paris" : 30, "madrid" : 35, "oslo" : 20}
2 del d["new-york"]
```

```
3 KeyError: 'new-york'
```

— Ligne 3 : `d` ne contient pas de clé `"new-york"`.

Plusieurs clés

Il est possible de supprimer plusieurs clés en une seule instruction `del` :

```
d = {"paris" : 30, "madrid" : 35, "oslo" : 20}
del d["paris"], d["madrid"]
print(d)
```

```
{'oslo': 20}
```

Fusionner deux dictionnaires

Étant donné deux dictionnaires `d1` et `d2`, la méthode `update` permet de fusionner `d1` et `d2` en `d1` tout en laissant intact `d2` :

```
d1 = {"paris" : 30, "madrid" : 35, "oslo" : 20}
d2 = {"new-york" : 28, "madrid" : 40}

print(d1)
print(d2)
print()
```

```
d1.update(d2)
print(d1)
print(d2)
```

```
{'paris': 30, 'madrid': 35, 'oslo': 20}
{'new-york': 28, 'madrid': 40}

{'paris': 30, 'madrid': 40, 'oslo': 20, 'new-york': 28}
{'new-york': 28, 'madrid': 40}
```

D'une façon générale, si d1 et d2 sont deux dictionnaires alors `d1.update(d2)` **étend** le dictionnaire d1 de tous les items de d2. Le dictionnaire d1 est donc en général modifié par la méthode update. Plus précisément

- si k est une clé présente dans d1 et d2, la valeur de k sera remplacée dans d1 par la valeur de k dans d2
- si k est une clé de d2 non présente dans d1, alors k deviendra une nouvelle clé de d1 et la valeur de k sera la valeur de k dans d2.

Attention, l'appel `d1.update(d2)` modifie d1 et renvoie `None` et non pas le dictionnaire d1 mis-à-jour.

Opérateur de fusion de dictionnaires

Étant donné deux dictionnaires d1 et d2, alors l'opération `d1 | d2` crée un dictionnaire qui est la fusion des deux dictionnaires d1 et d2. Cette fonctionnalité est disponible depuis la version 3.9. de Python. Voici un exemple d'utilisation :

```
d1 = {"paul" : 25, "jean" : 70, "thierry" : 40}
d2 = {"léa" : 15, "jean" : 50, "adam" : 1}
print(d1 | d2)
```

```
{'paul': 25, 'jean': 50, 'thierry': 40, 'léa': 15, 'adam': 1}
```

`d1 | d2` possède les items de d1 et de d2. En cas de conflit de clés, comme dans l'exemple avec la clé `"jean"`, c'est l'item du 2e dictionnaire qui est choisi, en sorte que les dictionnaires `d1 | d2` et `d2 | d1` ne sont pas forcément égaux. Noter que cette opération de fusion crée un **nouveau** dictionnaire qui ne modifie en rien les dictionnaires d1 ou d2 et complètement indépendant d'eux :

```
d1 = {"paul" : 25, "jean" : 70, "thierry" : 40}
d2 = {"léa" : 15, "jean" : 50, "adam" : 1}

d = d1 | d2
print("d =", d)

d1["paul"] = 100
d["adam"] = 2
d["thierry"] = 100
```

```
print("d1 =", d1)
print("d2 =", d2)
print("d =", d)
```

```
d = {'paul': 25, 'jean': 50, 'thierry': 40, 'léa': 15, 'adam': 1}
d1 = {'paul': 100, 'jean': 70, 'thierry': 40}
d2 = {'léa': 15, 'jean': 50, 'adam': 1}
d = {'paul': 25, 'jean': 50, 'thierry': 100, 'léa': 15, 'adam': 2}
```

On notera que modifier `d1` ou `d2` ne modifie pas `d` et qu'inversement, modifier `d` laisse intacts `d1` et `d2`.

Il existe une version mutatrice de l'opération avec l'instruction `|=` :

```
d1 = {"paul" : 25, "jean" : 70, "thierry" : 40}
d2 = {"léa" : 15, "jean" : 50, "adam" : 1}
d1 |= d2
print(d1)
```

```
{'paul': 25, 'jean': 50, 'thierry': 40, 'léa': 15, 'adam': 1}
```

En réalité, l'objet du membre de droite peut être un itérable de couples :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40}
L = [["léa", 15], ["jean", 50], ["adam", 1]]
d |= L
print(d)
```

```
{'paul': 25, 'jean': 50, 'thierry': 40, 'léa': 15, 'adam': 1}
```

2 Compléments

Ordre d'insertion des items d'un dictionnaire

Depuis la version 3.7 de python, un dictionnaire possède un ordre implicite qui est l'ordre d'insertion des items dans le dictionnaire. Cet ordre est celui d'apparition des items quand on affiche le dictionnaire. Par exemple, le code suivant :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40 }
print(d)

d["laure"]=25
print(d)

d["laure"]=30
d["hugo"]=49
print(d)
```

affiche

```
{'paul': 25, 'jean': 70, 'thierry': 40}
{'paul': 25, 'jean': 70, 'thierry': 40, 'laure': 25}
{'paul': 25, 'jean': 70, 'thierry': 40, 'laure': 30, 'hugo': 49}
```

ce qui montre que le premier élément entré dans d est de clé "paul", l'avant-dernier est de clé "laure" et le dernier est de clé "hugo".

Si un élément d'un dictionnaire est retiré et si ultérieurement, un item de même clé est replacé dans le dictionnaire, est-ce que le premier ordre d'insertion compte ? L'exemple qui suit illustre le comportement

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40 }
2 print(d)
3
4 del d["paul"]
5
6 d["laure"]=25
7 print(d)
8
9 d["paul"]=26
10 d["hugo"]=49
11 print(d)
```

```
12 {'paul': 25, 'jean': 70, 'thierry': 40}
13 {'jean': 70, 'thierry': 40, 'laure': 25}
14 {'jean': 70, 'thierry': 40, 'laure': 25, 'paul': 26, 'hugo': 49}
```

— Ligne 4 : la clé "paul" est retirée du dictionnaire.

— Ligne 9 : un item de clé "paul" est réintroduit.

— Ligne 14 : l'ordre du nouvel item est postérieur à celui de tous les éléments présents l'ordre de sa réinsertion.

Autrement dit, lorsqu'une clé est supprimée, son ordre lors de la suppression n'a aucune influence sur son ordre après réinsertion.

Enfin, si une valeur est modifiée, cela ne modifie pas son ordre dans le dictionnaire :

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40 }
2 print(d)
3
4 d["laure"]=25
5 d["paul"]=26
6 print(d)
```

```
7 {'paul': 25, 'jean': 70, 'thierry': 40}
8 {'paul': 26, 'jean': 70, 'thierry': 40, 'laure': 25}
```

— Ligne 5 : l'item de clé "paul" est modifié

— Ligne 8 : son ordre dans le dictionnaire n'a pas changé pour autant.

Comment se voit l'ordre d'un dictionnaire ?

L'ordre d'un dictionnaire est l'ordre de traitement des items quand on parcourt le dictionnaire avec une boucle for. C'est aussi l'ordre de placement des clés quand on convertit le dictionnaire

en liste. Et c'est aussi l'ordre d'affichage :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40 }
print(d)

print("----------------")

for k in d:
    print(k, d[k])

print("----------------")

print(list(d))
```

```
{'paul': 25, 'jean': 70, 'thierry': 40}
----------------
paul 25
jean 70
thierry 40
----------------
['paul', 'jean', 'thierry']
```

Copier un dictionnaire

Il y a deux façons équivalentes de procéder à une copie d'un dictionnaire `d` :

— on utilise la méthode `copy` de `d`

— on appelle le constructeur `dict` sur `d`.

La méthode copy

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
2
3 dd = d.copy()
4 print(dd)
5
6 dd["jeanne"] = 20
7 dd["pascal"] = 42
8 del dd["paul"]
9 print()
10 print(d)
11 print(dd)
```

```
12 {'paul': 25, 'jean': 70, 'thierry': 40, 'léa': 15, 'jeanne': 18}
13
14 {'paul': 25, 'jean': 70, 'thierry': 40, 'léa': 15, 'jeanne': 18}
15 {'jean': 70, 'thierry': 40, 'léa': 15, 'jeanne': 20, 'pascal': 42}
```

— Ligne 3 : `dd` est une copie de `d`.

— Lignes 6-8 : si on modifie ou on ajoute une clé, ces modifications n'affectent pas le dictionnaire originel `d` (c'est en cela que c'est une copie).

Le constructeur dict

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
2
3 dd = dict(d)
4 print(dd)
5
6 dd["jeanne"] = 20
7 dd["pascal"] = 42
8 del dd["paul"]
9 print()
10 print(d)
11 print(dd)
```

```
12 {'paul': 25, 'jean': 70, 'thierry': 40, 'léa': 15, 'jeanne': 18}
13
14 {'paul': 25, 'jean': 70, 'thierry': 40, 'léa': 15, 'jeanne': 18}
15 {'jean': 70, 'thierry': 40, 'léa': 15, 'jeanne': 20, 'pascal': 42}
```

— Ligne 3 : `dd` est une copie de `d`.

— Ligne 6-8 : si on modifie ou on ajoute une clé de `dd`, ces modifications n'affectent pas `d`.

Copie superficielle de dictionnaire

Toutefois, dans les deux cas (méthode `copy` ou constructeur `dict`), il se peut qu'une modification dans une copie affecte l'original.

Voici un exemple avec `dict` :

```
1 d = {"paul" : [25, 30, 35], "jean" : 70,
2       "thierry" : 40, "léa" : 15, "jeanne" : 18 }
3
4 dd = dict(d)
5
6 dd["paul"][0] = 42
7
8 print(d)
9 print()
10 print(dd)
```

```
11 {'paul': [42, 30, 35], 'jean': 70, 'thierry': 40, 'léa': 15, 'jeanne': 18}
12
13 {'paul': [42, 30, 35], 'jean': 70, 'thierry': 40, 'léa': 15, 'jeanne': 18}
```

— Ligne 6 : une valeur correspondant à une clé de la copie est modifiée

— Lignes 8 et 11 : cette modification affecte aussi l'original.

Plus précisément, si `d` est le dictionnaire original et si `dd` est une copie obtenue

— soit par `dd = d.copy()`

— soit par `dd = dict(d)`

alors si `k` est une clé de `d` et si `d[k]` référence un objet mutable tel qu'une liste `L`, toute modification du contenu de `L` via `dd[k]` modifiera l'original `d[k]`.

Il s'agit d'une copie superficielle suivant le même principe que la copie superficielle de liste.

Méthode `pop` de dictionnaire

La méthode pop d'un dictionnaire permet d'en supprimer une clé donnée :

```
d = {"paris" : 30, "madrid" : 35, "oslo" : 20}
print(d)

d.pop("oslo")
print(d)
```

```
{'paris': 30, 'madrid': 35, 'oslo': 20}
{'paris': 30, 'madrid': 35}
```

L'intérêt de cette méthode est qu'un appel à pop retourne la valeur de la clé avant suppression :

```
d = {"paris" : 30, "madrid" : 35, "oslo" : 20}
print(d)
k="oslo"
print(f"Il faisait {d.pop(k)}° à {k}")
print(d)
```

```
{'paris': 30, 'madrid': 35, 'oslo': 20}
Il faisait 20° à oslo
{'paris': 30, 'madrid': 35}
```

Un autre avantage de pop est qu'il est possible de donner une valeur de retour par défaut si la clé donnée n'est pas dans le dictionnaire :

```
d = {"paris" : 30, "madrid" : 35, "oslo" : 20}

k="Albi"
print(f"Il faisait {d.pop(k, '40')}° à {k}")
```

```
Il faisait 40° à Albi
```

Supprimer un item avec `popitem`

On peut supprimer d'un dictionnaire le dernier item inséré en appelant la méthode `popitem`. Dans certaines situations, ces suppressions sont répétées. Dans le code ci-dessous, on vide un dictionnaire dans l'ordre inverse où les clés sont insérées (les dernières insérées sont les premières sorties) :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40 }
d["laure"]=25
```

```
d["laure"]=30
d["hugo"]=49


while d:
    item=d.popitem()
    print(item)

print(d)
```

```
('hugo', 49)
('laure', 30)
('thierry', 40)
('jean', 70)
('paul', 25)
{}
```

Un appel à `popitem` sur un dictionnaire vide lève une exception de type `KeyError`.

Vider un dictionnaire

On peut effacer le contenu d'un dictionnaire `d` via la méthode `clear` :

```
d = {"paris" : 30, "madrid" : 35, "oslo" : 20}
print(d)

d.clear()
print(d)
```

```
{'paris': 30, 'madrid': 35, 'oslo': 20}
{}
```

L'intérêt de l'instruction `d.clear()` au lieu de l'instruction `d = {}` qui semble équivalente est que l'on modifie le dictionnaire initial au lieu d'en créer un nouveau si bien qu'on garde une référence vers le dictionnaire originel. Comparer les deux codes ci-dessous :

```
1 d = {"paris" : 30, "madrid" : 35, "oslo" : 20}
2 print(d)
3 dd = d
4
5 d.clear()
6 print(dd)
```

```
7 {'paris': 30, 'madrid': 35, 'oslo': 20}
8 {}
```

— Lignes 1 et 2 : `d` et `dd` référencent le même objet dictionnaire

— Ligne 5 : le seul et unique dictionnaire est vidé

— Ligne 6 : `dd` référence le dictionnaire vidé.

```
1 d = {"paris" : 30, "madrid" : 35, "oslo" : 20}
2 print(d)
3 dd = d
4
5 d = {}
6 print(dd)
```

```
7 {'paris': 30, 'madrid': 35, 'oslo': 20}
8 {'paris': 30, 'madrid': 35, 'oslo': 20}
```

— Lignes 1 et 2 : d et dd référencent le même objet dictionnaire

— Ligne 5 : un nouveau dictionnaire (vide) est créé

— Ligne 6 : dd référence toujours le dictionnaire initial qui lui n'a pas été vidé.

Outils de création d'items

Pour créer des listes d'items que l'on passe ensuite au constructeur `dict`, plusieurs techniques sont couramment employées dont les deux suivantes :

— les listes en compréhension,

— la fonction built-in `zip`.

Soit par exemple à créer un dictionnaire dont les clés soient les 8 chiffres romains :

"I", "V", "X", "L", "C", "D", "M"

et dont la valeur de chaque item soit la valeur numérique correspondant au chiffre romain, par exemple, la valeur de la clé "X" est l'entier 10.

Fonction zip

On peut créer notre dictionnaire en utilisant la fonction `zip` :

```
romains="IVXLCDM"
valeurs=[1, 5, 10, 50, 100, 500, 1000]

d = dict(zip(romains, valeurs))

print(d)
```

```
{'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}
```

Liste en compréhension

On peut créer notre dictionnaire en utilisant une liste en compréhension :

```
romains="IVXLCDM"
valeurs=[1, 5, 10, 50, 100, 500, 1000]

d=dict([[romains[i],valeurs[i]] for i in range(len(romains))])
```

```
print(d)
```

```
{'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}
```

Cette façon de procéder en utilisant indice pour accéder aux objets des listes n'est pas considérée comme pythonique.

Création, fusion de dictionnaires par décompactage

L'exemple qui suit va permettre de mieux comprendre la possibilité de création de dictionnaire par décompactage de dictionnaires déjà créés :

```
1 d1 = {"madrid" : 35, "oslo" : 20}
2 d2 = {"new-york" : 28, "madrid" : 40}
3
4 d = {"kiev":5, **d1, "odessa":12, **d2}
5
6 print(d)
```

```
7 {'kiev': 5, 'madrid': 40, 'oslo': 20, 'odessa': 12, 'new-york': 28}
```

Depuis la version 3.5 de Python, il est possible de construire un dictionnaire en plaçant entre accolades et séparés par des virgules des items clé-valeur ainsi que des références à des dictionnaires à décompacter avec la syntaxe `**`. Dans l'exemple ci-dessus, ligne 4, lors de la définiton de `d`, on lit `d1` qui est un dictionnaire et `**d1` qui est du sucre syntaxique pour écrire à l'intérieur des accolades de `d` la liste des items de `d1` (c'est comme si on retirait les accolades de `d1`). C'est pareil pour `d2`. Noter qu'on peut incorporer dans `d` des items hors dictionnaire, c'est le cas de par exemple de l'item `"odessa":12`.

En cas de conflit de clés, par exemple la clé `madrid` est présente dans `d1` et `d2`, c'est la dernière clé inséré dans le dictionnaire qui prime. Dans l'exemple ci-dessus, c'est l'item `"madrid" : 40` qui sera retenu dans `d` car cet item est dans `d2` qui est placé après (à droite de) `d1`.

Un cas particulier de ce qui précède est la fusion de deux dictionnaires `d1` et `d2` par `d = {**d1, **d2}`.

Parcours d'un dictionnaire suivant les items, suivant les valeurs

Parcours suivant les items

On peut parcourir les items d'un dictionnaire avec la méthode `items` :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
for k, v in d.items():
    print(k, "->", v)
```

```
paul -> 25
jean -> 70
thierry -> 40
léa -> 15
jeanne -> 18
```

La méthode `dict.items` retourne un conteneur :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
dd=d.items()

print(len(dd))
```

```
5
```

Parcours suivant les valeurs

On peut parcourir les valeurs d'un dictionnaire avec la méthode values :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
for v in d.values():
    print(v)
```

```
25
70
40
15
18
```

La méthode `dict.values` retourne un conteneur qui contient des références vers les objets qui sont les valeurs du dictionnaire.

L'intérêt de la méthode `values` est souvent limité car dans un dictionnaire, les valeurs seules, sans les clés, sont peu utilisables.

Listes des clés, des items, des valeurs d'un dictionnaire

On dispose d'un dictionnaire et on veut récupérer :

— une liste de clés

— une liste des items

— une liste des valeurs

Liste des clés

Pour récupérer une liste des clés, on utilise le constructeur `list` sur le dictionnaire :

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
2 print(list(d))
```

```
3 ['paul', 'jean', 'thierry', 'léa', 'jeanne']
```

— Ligne 2 : la liste des clés. Elles sont données dans le même ordre que l'ordre dans le dictionnaire littéral.

Liste des items

Pour récupérer une liste des items, il suffit d'appeler la méthode `items` et de convertir le conteneur en liste :

methode_items.py

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
2 print(list(d.items()))
```

```
3 [('paul', 25), ('jean', 70), ('thierry', 40), ('léa', 15), ('jeanne', 18)]
```

— Ligne 2 : `d.items()` est un conteneur des items de `d` et `list` permet de récupérer sous forme de liste les éléments sur lesquels on itère, ici les items.

Les items sont fournis sous forme de tuple (clé, valeur) et non pas de liste.

L'usage de la méthode `items` est à peu près équivalent à générer les items en parcourant le dictionnaire par ses clés :

items_cles.py

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
items = [(k, d[k]) for k in d]
print(items)
```

```
[('paul', 25), ('jean', 70), ('thierry', 40), ('léa', 15), ('jeanne', 18)]
```

Liste des valeurs

Pour récupérer une liste des valeurs, il suffit d'appeler la méthode `values` et de convertir le conteneur en liste :

methode_values.py

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
print(list(d.values()))
```

```
[25, 70, 40, 15, 18]
```

L'usage de la méthode `values` est à peu près équivalent à placer les valeurs dans une liste en compréhension en parcourant le dictionnaire par ses clés :

valeurs_cles.py

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
valeurs = [d[k] for k in d]
print(valeurs)
```

```
[25, 70, 40, 15, 18]
```

Noter qu'entre `valeurs_cles.py` et `methode_values.py` l'ordre des valeurs est le même.

Appeler `dict` avec des arguments nommés

La syntaxe de `dict` est typiquement illustrée par l'exemple suivant :

```
1 mes_items = [["paul", 25], ["jeanne", 18],
2              ["thierry", 45], ["jean", 70], ["léa", 15]]
3 d = dict(mes_items)
4 print(d)
```

```
5 {'paul': 25, 'jeanne': 18, 'thierry': 45, 'jean': 70, 'léa': 15}
```

— ligne 3 : `dict` prend en argument une liste (lignes 1-2) formée de listes ayant chacune deux éléments.

Lorsque les clés sont des chaînes de caractères ayant une certaine forme (qui sera précisée ci-dessous), une autre syntaxe est possible :

```
1 d= dict(paul = 25,  jeanne = 18,  thierry = 45,  jean = 70,  lea = 15)
2 print(d)
3
4 print(d["jeanne"])
```

```
5 {'paul': 25, 'jeanne': 18, 'thierry': 45, 'jean': 70, 'lea': 15}
6 18
```

Détaillons cette syntaxe. Lorsque les clés d'un dictionnaire à construire sont toutes des chaînes de caractères pouvant représenter des noms de variables valides (ici, `"paul"`, `"jeanne"`, etc), il suffit de donner en argument à `dict` (ligne 1) un argument nommé de la forme `key = value` où `key` est le contenu de la chaîne de caractères représentant une clé et `value` est la valeur correspondant à cette chaîne (par exemple `jean=70`).

Dans l'exemple ci-dessus, l'appel à `dict` est référencé par `d`. La chaîne `"jeanne"` est une clé de `d` car l'identificateur `jeanne` est un argument nommé de l'appel de dict. La valeur de la clé `"jeanne"` est fournie par la valeur affectée à l'argument nommé, ici 18.

Attention, ce procédé ne fonctionne que si les clés du dictionnaire à construire sont des chaînes de caractères ayant la syntaxe d'un identificateur. Ainsi, un dictionnaire contenant une clé valant 42 ne pourra être créé de la sorte.

Noter que l'ordre d'apparition des arguments nommés définit l'ordre d'insertion de ces éléments du dictionnaire.

Dictionnaire créé avec la méthode `dict.fromkeys`

`dict` est une classe, représentant le type dictionnaire. Cette classe possède une méthode qui permet de créer un dictionnaire à partir des deux éléments suivants :

— une liste de clés

— une même et unique valeur à donner à chacune des clés.

Cette méthode s'appelle `fromkeys`.

Voici un exemple d'utilisation :

```
1 keys = ["paul", "jeanne", "thierry", "jean", "léa"]
2 d = dict.fromkeys(keys, 42)
3
4 print(d)
```

```
5 {'paul': 42, 'jeanne': 42, 'thierry': 42, 'jean': 42, 'léa': 42}
```

— Ligne 1 : les clés du futur dictionnaire

— Lignes 2 et 5 : chaque clé recevra pour valeur 42 ; noter que la méthode est appelée en suffixant la classe `dict`.

Dans le jargon de la POO Python, cette méthode est dite *méthode de classe*.

Absence de valeur

Si aucune valeur pour les clés n'est donnée, la valeur par défaut `None` est attribuée à chaque clé :

```
1 keys = ["paul", "jeanne", "thierry", "jean", "léa"]
2 d = dict.fromkeys(keys)
3
4 print(d)
```

```
5 {'paul': None, 'jeanne': None, 'thierry': None, 'jean': None, 'léa': None}
```

— Ligne 2 : la méthode `dict.fromkeys` est appelée sans valeur à attribuer à chaque clé
— Ligne 5 : chaque clé reçoit pour valeur `None`.

Itérable en argument de `fromkeys`

Au lieu d'une liste, la méthode `fromkeys` peut prendre n'importe quel itérable, par exemple une chaîne :

```
d=dict.fromkeys("aeiouy")
print(d)
```

```
{'a': None, 'e': None, 'i': None, 'o': None, 'u': None, 'y': None}
```

Syntaxes variées de l'argument itérable de `dict`

Le type `dict` peut être utilisé comme une fonction. L'idée est simple : pour créer un dictionnaire, `dict` reçoit un itérable générant des éléments ayant DEUX composantes :

— une composante pour la clé
— une composante pour la valeur de la clé.

En pratique, pour créer un dictionnaire par cette méthode, de nombreuses variations de syntaxe sont possibles puisqu'il existe de nombreuses façons de créer un itérable générant deux éléments.

Exemple typique

Par exemple, voici deux façons équivalentes de définir un même dictionnaire :

```
1 mes_items = [["paul", 25],  ["thierry", 45], ["jean", 70], ["léa", 15]]
2 d = dict(mes_items)
3 print(d)
4
5 mes_items = ("paul", 25), ("thierry", 45), ["jean", 70], ["léa", 15]
6 d = dict(mes_items)
7 print(d)
```

```
8 {'paul': 25, 'thierry': 45, 'jean': 70, 'léa': 15}
9 {'paul': 25, 'thierry': 45, 'jean': 70, 'léa': 15}
```

— Ligne 1 : `mes_items` est une liste de listes de deux éléments.
— Ligne 5 : `mes_items` est un tuple de listes ou de tuples ayant chacun deux éléments.

dict sans argument

Si dict est appelé sans prendre d'argument, dict renvoie un dictionnaire vide :

```
d = dict()
print(len(d))
```

```
0
```

Accès au premier élément d'un dictionnaire

Si un dictionnaire n'est pas vide, et qu'on ne connaît pas les clés du dictionnaire comment accéder à un élément du dictionnaire ? Pour y parvenir de façon économique et simple, il suffit de créer un itérateur sur le dictionnaire et d'itérer juste une seule fois :

```
d = {"léa" : 15, "paul" : 25, "jean" : 70, "thierry" : 40, "jeanne" : 18 }

if d:
    it = iter(d)
    print(next(it))
```

```
léa
```

L'item obtenu est le premier item inséré dans le dictionnaire.

La méthode get

Si on essaie d'accéder à une valeur d'un dictionnaire d avec une clé qui n'existe pas, on obtient un message d'erreur :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }

print(d["pascal"])
```

```
KeyError: 'pascal'
```

Pour remédier à ce problème, on peut envisager de s'assurer avec un test de la présence de la clé avant de l'utiliser. En fait, on peut parvenir au même résultat avec la méthode get mais sans prendre le risque de lever une exception :

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
2
3 v = d.get("pascal")
4 print(v)
```

```
5 None
```

— Ligne 3 : si la chaîne "pascal" est une clé du dictionnaire, v référencera la valeur associée à "pascal" dans le dictionnaire, sinon, v référencera None.

Retour de la méthode `get`

Par défaut, si `d` est dictionnaire, l'appel `d.get(my_key)` renvoie `None` si `my_key` n'est pas dans `d`. Pour que la méthode `get` puisse renvoyer une autre valeur `v` que `None`, il suffit de placer `v` comme deuxième argument :

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18 }
2
3 v = d.get("pascal", 42)
4 print(v)
```

```
5 42
```

— Ligne 3 : la valeur par défaut, ici 42, est placé comme deuxième argument optionnel de l'appel à `get`.

— Lignes 3 et 5 : la clé `"pascal"` est absente de `d`, donc la méthode `get` renvoie `42`.

Exemple d'utilisation de la méthode `get`

Voici un cas d'usage de la méthode `get`. On dispose d'une liste `L` de nombres entiers et on souhaite créer un dictionnaire `d` tel que

— chaque clé `k` soit une des valeurs distinctes de `L`

— la valeur de la clé `k` soit le nombre d'occurrences de `k` dans `L`.

Par exemple, si

```
L = [65, 9, 65, 9, 31, 31, 32, 65, 32, 31, 9, 31]
```

alors

```
d =  {32: 2, 65: 3, 31: 4, 9: 3}
```

car, par exemple, 31 est présent 4 fois dans `L`.

On peut coder le problème comme suit :

```
1 def count(L):
2     d={}
3     for x in L:
4         d[x] =d.get(x, 0)+1
5     return d
6
7 L=[65, 31, 9, 32, 81]
8 print(count(L))
```

```
9 {65: 1, 31: 1, 9: 1, 32: 1, 81: 1}
```

— Ligne 2 : au départ, `d` est vide.

— Lignes 3-4 : lors du parcours de `L`, si une clé `x` n'est pas présente dans `d` alors `get(x, 0)` renvoie `0` et donc `d[x]` vaut `1` ce qui est bien le nombre d'occurrence de `x` à ce moment du parcours de `d`. Si ultérieurement, `x` apparaît à nouveau dans `L` alors `d.get(x, 0)+1` vaudra `d[x] + 1` ce qui est bien le nombre d'occurrences de `x` dans `d` à ce moment-là.

Ambiguïté possible de la méthode `get`

Observons d'abord l'exemple suivant :

```
d1 = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15}
d2 = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "pascal" : None}

print(d1.get("pascal"), d2.get("pascal"))
```

```
None None
```

— `d1` ne contient pas la clé `"pascal"` et `get` renvoie donc `None`

— `d2` contient la clé `"pascal"` associée à la valeur `None` qui est donc renvoyée par `get`.

La méthode `get` d'un dictionnaire ne donne pas une information complète s'il retourne `None`. En effet, si `get` renvoie `None` alors :

— ou bien la clé n'est pas dans le dictionnaire,

— ou bien la clé est dans le dictionnaire, associée à la valeur `None`.

Tuple comme clé d'un dictionnaire

Dans un dictionnaire littéral, si une clé est un tuple littéral, il faut placer des parenthèses autour du tuple. Ainsi, le code suivant est invalide :

```
d={1, "A": 42}
```

Il faut écrire

```
d={(1, "A"): 42}
print(d)
```

```
{(1, 'A'): 42}
```

Pour accéder à un élément du dictionnaire avec des crochets, la présence de parenthèses n'est pas obligatoire :

```
d={(1, "A"): 42}
print(d[1, "A"])
```

```
42
```

Attention qui si on fait appel à la méthode pop, il faudra placer des parenthèses, sinon

```
1 d={}
2 d[1, "A"] = 42
3
4 d.pop(1, "A")
5
6 print(d)
```

```
7 {(1, 'A'): 42}
```

le code n'aura pas l'action souhaité et ne renverra aucune erreur :

— ligne 4 : comme 1 n'est pas clé du dictionnaire d, aucune clé n'est retirée, aucune exception n'est levée (c'est le principe de la méthode pop) et l'expression renverra la valeur "A" (sans l'utiliser).

La classe Counter pour lister les doublons

Le module standard collections dispose d'une structure de données appelée **Counter** et qui enregistre le nombre d'occurrences d'éléments d'un itérable. Voici un exemple :

```
from collections import Counter

L = [65, 9, 65, 9, 31, 31, 32, 42, 65, 32, 31, 50, 9, 31]

cnt = Counter(L)
print(cnt)
```

```
Counter({31: 4, 65: 3, 9: 3, 32: 2, 42: 1, 50: 1})
```

On observe que cnt représente un dictionnaire dont les clés sont les éléments distincts de l'itérable L et dont les valeurs correspondantes sont simplement le nombre d'occurrences dans l'itérable. Ainsi, on observe bien que dans L, l'élément 31 apparaît 4 fois.

Une instance de la classe Counter, comme cnt ci-dessus, est un dictionnaire, et on peut donc utiliser les méthodes d'un dictionnaire, ce qui permet de facilement lister les doublons d'un itérable :

```
from collections import Counter

L = [65, 9, 65, 9, 31, 31, 32, 42, 65, 32, 31, 50, 9, 31]

cnt = Counter(L)
distrib=cnt.items()
doublons = [z for (z, k) in distrib if k >1]

print(doublons)
```

```
[65, 9, 31, 32]
```

Comme pour un dictionnaire, les éléments de l'itérable doivent être immutables.

Vues d'un dictionnaire

Conteneur adapté aux « composants » d'un dictionnaire

On appellera *composant* d'un dictionnaire :

— soit les clés,

— soit les valeurs des clés

— soit les items qui sont des paires clé et valeur.

La liste n'est pas forcément le conteneur pertinent pour placer les composants d'un dictionnaire.

Un dictionnaire possède justement des méthodes qui renvoient des conteneurs adaptés. Ces méthodes sont les suivantes :

— `keys` pour les clés
— `values` pour les valeurs
— `items` pour les items

Ces méthodes renvoient un objet appelé *vue* (*view*). Les vues permettent d'utiliser les mêmes opérations que les listes (l'appartenance, la longueur, l'itération) :

```
d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15, "jeanne" : 18}

v = d.values()
print(sum(v))
print(42 in v)

print(("léa", 15) in d.items())

print("pascal" in d.keys())
```

```
168
False
True
False
```

L'intérêt essentiel est que cela évite de créer des listes inutiles : par exemple, `list(d.values())` ne permet pas de faire plus d'opérations pertinentes que ne le permet `d.values()`.

Vues dynamiques sur un dictionnaire

La méthode `keys` d'un dictionnaire `d` est une vue *dynamique* sur les clés de `d`, autrement dit, si les clés de `d` sont modifiées par ajout ou suppression, `d.keys()` mettra à jour les modifications sur les clés (suppression ou insertion) :

```
1 d = {"Paul" : [25, 30, 35], "Jean" : 70, "Thierry" : 40, "Léa" : 15}
2
3 k = d.keys()
4 print(len(k))
5
6 d["Anouar"] = 42
7 del d["Jean"], d["Léa"]
8 print(len(k))
```

```
9 4
10 3
```

— Ligne 3 : `k` est une vue dynamique sur les clés de `d`
— Ligne 1 : `d` a présentement 4 clés
— Ligne 6 : on ajoute une clé à `d`
— Ligne 7 : on retire à `d` deux clés
— Ligne 8 : `k` a bien mis à jour les clés puisque il répertorie $4 + 1 - 2 = 3$ clés pour `d`.

Le cas de values et items

Il en est de même des méthodes `values` et `items` :

```
d = {"Paul" : [25, 30, 35], "Jean" : 70, "Thierry" : 40, "Léa" : 15}

vals = d.values()

print(70 in vals)

del d["Jean"]
print(70 in vals)
print()

d["Thierry"] = 42
print(42 in vals)
```

```
True
False

True
```

Synchronisation des vues

Les trois vues (`keys`, `values`, `items`) d'un dictionnaires sont synchronisées, c'est-à-dire que si on parcourt simultanément :

— les clés avec la méthode `keys`,

— les valeurs avec la méthode `values`

— les items par la méthode `items`,

à chaque étape, l'item sera bien constitué de la clé et de la valeurs :

```
d = {"Paul" : [25, 30, 35], "Jean" : 70, "Thierry" : 40, "Léa" : 15}

K = d.keys()
V = d.values()
I = d.items()

print(list(K))
print(list(V))
print(list(I))
```

```
['Paul', 'Jean', 'Thierry', 'Léa']
[[25, 30, 35], 70, 40, 15]
[('Paul', [25, 30, 35]), ('Jean', 70), ('Thierry', 40), ('Léa', 15)]
```

— Par vérification visuelle, on observe qu'à un indice donné `i`, l'item de la troisième liste est formé de la clé en indice `i` de la première liste et de la valeur à l'indice `i` de la deuxième liste.

Les dictionnaires par défaut (`defaultdict`)

La bibliothèque standard de Python définit un type de dictionnaire appelé `defaultdict` qui a toutes les propriétés d'un dictionnaire (un `defaultdict` est un type, qui hérite du type `dict`) et qui en diffère seulement lorsqu'on cherche à accéder à une clé du dictionnaire qui n'existe pas et pour laquelle, justement, le `defautdict` fournit une valeur par défaut (d'où le terme de `defaultdict`). Pour définir un `defaultdict`, il faut l'importer depuis le module `collections` :

```
1 from collections import defaultdict
2
3 def f():
4     return 42
5
6 d = defaultdict(f)
7 d["begonia"] = 7
8 d["rose"] = 4
9 print(d)
10 print(d["prune"])
11 print(d)
```

```
12 defaultdict(<function f at 0x7f22cec441f0>, {'begonia': 7, 'rose': 4})
13 42
14 defaultdict(<function f at 0x7f22cec441f0>, {'begonia': 7, 'rose': 4, 'prune': 42})
```

— Lignes 3-6 : pour définir `defaultdict`, on a besoin d'une fonction qui va fournir les valeurs par défaut aux clés non présentes dans le dictionnaire. Ici la fonction est la fonction `f` constante et qui renvoie 42.

— Ligne 6 : pour l'instant `d` est vide.

— Lignes 7, 9 et 12 : on définit des clés du dictionnaire avec leur valeur, comme on le ferait pour un dictionnaire habituel en sorte que `d` possède désormais deux clés.

— Ligne 10 : on tente d'accéder à une clé de `d` qui n'existe pas, la clé `"prune"`. Cet accès a pour conséquence :

  — d'appeler la fonction `f` et de renvoyer `f()` (sans argument) pour `d["prune"]`

  — de placer `"prune"` dans le dictionnaire avec pour valeur `42`, cf. ligne 14.

La fonction donnée en argument à `defaultdict` s'appelle une `default_factory` (fabrique de valeurs par défaut) et est un attribut du `defaultdict` créé.

En cas de clé absente, la fonction `defaultdict`, disons `f`, ne reçoit aucun argument particulier, autrement dit est appelée sous la forme `f()`.

Outre `default_factory`, on peut donner des arguments nommés à `default_factory` pour créer immédiatement des clés dans le dictionnaire, exactement comme on le ferait avec le constructeur `dict` :

```
from collections import defaultdict

def f():
    return 42

d = defaultdict(f, begonia = 7, rose = 4)
```

```
print(d.default_factory)
print(d)
```

```
<function f at 0x7f74381f41f0>
defaultdict(<function f at 0x7f74381f41f0>, {'begonia': 7, 'rose': 4})
```

Utilisation classique

Un exemple d'utilisation classique d'un `defaultdict` est le suivant : on dispose d'une liste `L` de nombres entiers et on souhaite créer un dictionnaire `d` dont chaque clé `k` soit une des valeurs distinctes de `L` et dont la valeur soit le nombre d'occurrences de `k` dans `L`. Par exemple, si `L = [65, 9, 65, 9, 31, 31, 32, 65, 32, 31, 9, 31]` alors `d = {32: 2, 65: 3, 31: 4, 9: 3}` car, par exemple, `31` est présent 4 fois dans `L`.

```
1  from collections import defaultdict
2
3  def count(L):
4      d=defaultdict(int)
5      for x in L:
6          d[x] +=1
7      return d
8
9  L= [65, 9, 65, 9, 31, 31, 32, 65, 32, 31, 9, 31]
10 print(count(L))
```

```
11 defaultdict(<class 'int'>, {65: 3, 9: 3, 31: 4, 32: 2})
```

— Ligne 4 : il est assez fréquent de donner comme `default_factory` un constructeur built-in comme `list`, `set` ou, comme ici, `int`. L'important ici est que `int()` vaille 0.

— Ligne 6 : l'affectation `d[x] += 1` serait impossible avec un `dict` classique lorsque `x` est une clé absente du dictionnaire `d`. Avec un `defaultdict`, lorsque `x` n'est pas présent dans le dictionnaire, `d[x]` est donné pour la valeur par défaut `int()`, c'est-à-dire `0`, et donc `d[x] = d[x] + 1` donne `d[x] = 1`.

Noter que le code précédent est équivalent à :

```
from collections import defaultdict

def f():
    return 0

d=defaultdict(f)

L= [65, 9, 65, 9, 31, 31, 32, 65, 32, 31, 9, 31]
for x in L:
    d[x] +=1

print(d)
```

```
defaultdict(<function f at 0x7f187dd381f0>, {65: 3, 9: 3, 31: 4, 32: 2})
```

La méthode `setdefault`

La méthode `setdefault` permet de placer un item dans un dictionnaire `d` si la clé proposée par `setdefault` n'est pas déjà présente dans `d`.
Voici un exemple d'utilisation de la méthode `setdefault` :

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15}
2
3 v = d.setdefault("pascal", 42)
4 print(v)
5 print(d)
6 print()
7
8 v = d.setdefault("jeanne", 20)
9 print(v)
10 print(d)
```

```
11 42
12 {'paul': 25, 'jean': 70, 'thierry': 40, 'léa': 15, 'pascal': 42}
13
14 20
15 {'paul': 25, 'jean': 70, 'thierry': 40, 'léa': 15, 'pascal': 42, 'jeanne': 20}
```

— Ligne 3 : la chaîne `"pascal"` ne figure pas dans `d` donc elle est ajoutée et `setdefault` renvoie la valeur donnée à cette clé, qui est celle présente en argument à `setdefault`.
— Ligne 8 : la chaîne `"jeanne"` figure dans `d` donc l'item n'est pas modifié et `setdefault` renvoie la valeur de la clé dans `d`.

Autrement dit, lorsque la méthode `setdefault` appliquée à un dictionnaire `d` prend deux arguments `k` et `v`, elle crée l'item `(k, v)` si la clé `k` est absente de `d`. Un appel à `setdefault` renvoie la valeur de la clé présente dans le dictionnaire.

Absence de valeur

Si la méthode `setdefault` appliquée à un dictionnaire `d` ne prend qu'un seul argument et pas de valeur alors, si `k` n'est pas une clé de `d`, alors `setdefault` ajoute `k` aux clés du dictionnaire et associe `None` pour valeur de cette clé :

```
1 d = {"paul" : 25, "jean" : 70, "thierry" : 40, "léa" : 15}
2 d.setdefault("pascal")
3 print(d)
```

```
4 {'paul': 25, 'jean': 70, 'thierry': 40, 'léa': 15, 'pascal': None}
```

— Ligne 2 : `setdefault` ne prend ici qu'un seul argument, ici la chaîne `"pascal"`
— Lignes 3-4 : après appel, `"pascal"` est devenue une clé de `d` et la valeur de la clé est `None`.

La méthode `__missing__`

La méthode `__missing__` permet d'ajouter des valeurs par défaut à des clés absentes :

```
1 class D(dict):
2     def __missing__(self, k):
3         return "rosa"
4
5 d=D()
6 d[5]= "begonia"
7 d[10]= "kiwi"
8 print(d)
9 print(d[42])
10 print(d)
```

```
11 {5: 'begonia', 10: 'kiwi'}
12 rosa
13 {5: 'begonia', 10: 'kiwi'}
```

— Lignes 5-8 : on crée un dictionnaire avec deux clés.

— Ligne 9 : on tente d'accéder à un clé inexistante.

— Ligne 9 : au lieu de lever une exception `KeyError`, la méthode `__missing__` est appelée et le retour de l'appel donne une valeur.

— Lignes 10 et 13 : toutefois, aucune clé `42` n'est rajoutée au dictionnaire `d`.

Comparer deux dictionnaires

L'opérateur `==` permet de comparer deux dictionnaires c'est-à-dire de savoir s'ils ont même contenu.

Deux dictionnaires sont considérés comme égaux s'ils admettent le même ensemble de clés et si les contenus correspondants de chaque clé sont égaux :

```
d = {42: [1, 2, 3], 10:10}
dd = {10:10, 42: [1, 2, 3]}

print(d == dd)
```

```
True
```

— les deux dictionnaires `d` et `dd` sont égaux.

L'ordre d'insertion des clés n'a pas d'importance lorsqu'on compare deux dictionnaires.

Valeurs différentes

Si les clés sont identiques mais que les valeurs ne correspondent pas, les dictionnaires ne sont pas considérés comme égaux :

```
d = {42: [1, 2, 3], 10:10}
dd = {10:1000000, 42: [1, 2, 3]}

print(d == dd)
```

```
False
```

Clés différentes

Et si les clés ne sont pas identiques, les dictionnaires ne sont pas égaux :

```
d = {42: [1, 2, 3], 10:10}
dd = {10000:10, 42: [1, 2, 3]}

print(d == dd)
```

```
False
```

Modifier un dictionnaire pendant son parcours

On NE peut rajouter ou supprimer des items d'un dictionnaire `d` lorsque celui-ci est parcouru par une boucle `for` :

```
d={2020:42, 421:100, 512:1789, 0:42}

for k in d:
    d[k+1]=0
```

```
Traceback (most recent call last):
  File "_.py", line 3, in <module>
    for k in d:
RuntimeError: dictionary changed size during iteration
```

Cela est dû à l'implémentation d'un dictionnaire : ajouter ou retirer des clés à un dictionnaire pourrait obliger à réimplémenter complètement le dictionnaire [4].

Itérateur sur un dictionnaire

Si `d` est un dictionnaire, on obtient un itérateur sur `d` en utilisant la fonction standard `iter` :

```
d = {"léa" : 15, "paul" : 25, "jean" : 70, "thierry" : 40, "jeanne" : 18 }
it = iter(d)
print(list(it))
```

```
['léa', 'paul', 'jean', 'thierry', 'jeanne']
```

`iter(d)` itère sur les clés du dictionnaire `d` et, dans le même ordre, que lorsqu'on parcourt `d` avec une boucle `for` :

```
d = {"léa" : 15, "paul" : 25, "jean" : 70, "thierry" : 40, "jeanne" : 18 }
it = iter(d)
print(list(it))

for k in d:
    print(k, end = " ")
print()
```

4. Brandon Craig Rhodes : The Mighty Dictionary (PyCon 2010)

```
['léa', 'paul', 'jean', 'thierry', 'jeanne']
léa paul jean thierry jeanne
```

`dict` et couple d'items non usuels

Quand on construit un dictionnaire `d = dict(it)`, l'objet `it` est un itérable sur des objets ayant exactement **deux** membres.

Par exemple, soit le fichier-texte à deux lignes suivant :

```
orange
kiwi
```

Voici un exemple peu réaliste mais possible d'items pour un dictionnaire :

```
1 mes_items = ["42", open("my_file.txt"), set([42,2020])]
2
3 d = dict(mes_items)
4 print(d)
```

```
5 {'4': '2', 'orange\n': 'kiwi\n', 42: 2020}
```

— Ligne 1 : la chaîne `"42"` est un itérable à deux membres

— Ligne 1 : `open("my_file.txt")` est un itérable sur deux lignes de texte. La première ligne devient une clé de `d`, la deuxième la valeur correspondante.

— Ligne 1 : `set([42,2020])` est bien un itérable à deux éléments mais on ne sait pas à l'avance comment lequel des deux éléments sera le premier itérable et donc la clé. Donc ce type d'item est à éviter.

Signature de `dict` et arguments nommés

L'usage d'arguments nommés avec le constructeur `dict` :

```
d= dict(paul = 25,  jeanne = 18,  thierry = 45,  jean = 70,  léa = 15)
print(d)

print(d["jeanne"])
```

```
{'jean': 70, 'thierry': 45, 'léa': 15, 'paul': 25, 'jeanne': 18}
18
```

n'a rien de spécifique aux dictionnaires ; elle est simplement conforme à la signature du type `dict` :

```
dict(it, **kwargs)
```

Les arguments nommés donnés à `dict` comme sont `paul = 25` dans l'exemple ci-dessus, sont compressés en le dictionnaire `kwargs`.

Les dictionnaires de type `OrderedDict`

La classe `OrderedDict` est une classe dérivée de la class `dict` qui permet de disposer, dès la version 3.1 de Python, de dictionnaires préservant l'ordre d'insertion des items :

```
from collections import OrderedDict

d = OrderedDict([('paul', 25), ('jean', 70), ('thierry', 40)])
print(d)
d["laure"]=25
print(d)
```

```
OrderedDict([('paul', 25), ('jean', 70), ('thierry', 40)])
OrderedDict([('paul', 25), ('jean', 70), ('thierry', 40), ('laure', 25)])
```

Depuis la version 3.7 de Python, les dictionnaires de type `dict` possèdent cette propriété de respect de l'ordre d'insertion, ce qui retire de l'intérêt à la classe `OrderedDict`. Toutefois, et suivant les arguments de Greg Gandenberger, ces dictionnaires gardent encore un intérêt pour les deux raisons suivantes :

— compatibilité avec du code antérieur à la version 3.7 de Python ;

— déclaration que l'ordre d'insertion des clés du dictionnaire est nécessaire au bon fonctionnement du programme. Et en effet, dans de nombreux programmes utilisant des dictionnaires, on n'a pas besoin de tenir compte de l'ordre d'insertion des clés. Utiliser un `OrderedDict` est justement une façon de déclarer que cet ordre a une importance.

Noter que la comparaison de dictionnaires de type `OrderedDict` est plus stricte que celle des `dict` :

```
from collections import OrderedDict

L = [('paul', 25), ('jean', 70)]
M = [('jean', 70), ('paul', 25)]


print(dict(L) == dict(M))
print(OrderedDict(L) == OrderedDict(M))
```

```
True
False
```

Chapitre II

Ensembles

1 Généralités sur les ensembles

La structure d'ensemble

Un ensemble est une structure de données de type conteneur et dont *les éléments sont deux à deux distincts*. Python dispose nativement d'une structure appelée `set` qui implémente une structure d'ensemble. Voici un exemple :

```
1 s = {65, 31, 12, 81, 81, 65, 81, 12}
2 print(s)
```

```
3 {81, 31, 12, 65}
```

— Ligne 1 : `s` est un ensemble au sens du langage Python.

— Ligne 3 : on observe que `s` a bien ses éléments deux à deux distincts.

Les ensembles en Python définissent un type, appelé *set*[1].

Un ensemble ressemble à une liste sauf que

— ses éléments ne sont jamais répétés,

— il n'existe pas d'ordre prédéfini de parcours de l'ensemble, tandis que pour une liste, l'ordre par défaut est donné par indices croissants.

A la différence des dictionnaires, les ensembles en Python ne préservent pas l'ordre d'insertion des éléments dans l'ensemble, voir cette discussion.

Ensemble littéral

Le moyen le plus immédiat de définir un ensemble est de lister tous ses éléments en les plaçant entre des accolades et séparés par des virgules, comme on le fait en mathématiques. Une telle expression définit ce qu'on appelle un *ensemble littéral* :

1. En réalité, il existe deux catégories d'ensembles en Python : `set` et `frozenset`. Ce dernier semble moins utilisé. La différence entre ces deux types est que `set` est mutable alors que `frozenset` est immutable.

```
1 s = {65, 31, 12, 81, 81, 65, 81, 12}
2 print(s)
```

```
3 {81, 31, 12, 65}
```

— Ligne 1 : par définition, le membre de droite de l'affectation est un *ensemble littéral*. Ses éléments figurent entre les accolades et, ici, sont des entiers.

Comme un ensemble ne contient que des éléments différents, si dans la liste entre accolades on place des éléments de même valeur, seul un exemplaire de cet élément est répertorié dans l'ensemble :

```
1 s = {65, 31, 12, 81, 81, 65, 81, 12}
2 print(s)
```

```
3 {81, 31, 12, 65}
```

— Ligne 1 : des éléments identiques apparaissent dans la liste des éléments de l'ensemble `s`.

— Ligne 3 : on lit la liste des éléments distincts. Chaque élément qui était répété est pris en compte une seule fois.

Nombre d'éléments d'un ensemble

Un ensemble est un conteneur et il possède donc un certain nombre d'éléments. Le nombre d'éléments d'un ensemble `s` est obtenu par appel à la fonction built-in `len` :

```
1 s = {65, 31, 12, 81, 81, 65, 81, 12}
2 print(s)
3 n = len(s)
4 print(n)
```

```
5 {81, 31, 12, 65}
6 4
```

— Lignes 2 et 6 : le nombre d'éléments de l'ensemble `s`. On observera qu'un élément présent plusieurs fois dans la liste initiale n'est compté qu'une seule fois.

Affichage d'un ensemble

Si `s` est un ensemble non vide, l'affichage produit par `print(s)` est l'expression d'un ensemble sous forme d'ensemble littéral :

```
1 s = {65, 31, 12, 81, 81, 65, 81, 12}
2 print(s)
```

```
3 {81, 31, 12, 65}
```

Les éléments de l'ensemble apparaissent une fois et une seule, il n'y a jamais de doublons dans l'affichage. L'ordre d'affichage est non spécifié.

Création d'ensembles avec `set`

Il existe d'autres procédés de création d'ensemble que la syntaxe d'ensemble littéral [2]. On peut construire un ensemble avec le type `set`. Le processus est expliqué à l'exemple suivant :

```
1 L = [65, 31, 12, 81, 81, 65, 81, 12]
2 s = set(L)
3 print(s)
```

```
4 {65, 12, 81, 31}
```

— Ligne 1 : pour créer un ensemble formé à partir de certains éléments, on peut placer tous ses éléments dans une liste.

— Ligne 2 : on appelle ensuite le constructeur `set` sur cette liste. L'appel renvoie alors l'ensemble associé. Les éléments de la liste présents plusieurs fois ne sont pris en compte qu'une seule fois.

`set` est appelé un *constructeur* puisqu'il permet de construire des objets d'une certaine catégorie, ici un ensemble.

Le constructeur `set` permet de construire un ensemble à partir d'une liste. Les éléments distincts de la liste seront les éléments de l'ensemble.

Ensemble vide

Un ensemble peut être vide :

```
1 s = set()
2 print(s)
3 print(len(s))
```

```
4 set()
5 0
```

— Lignes 1 et 4 : observer la notation spécifique de l'ensemble vide

Un ensemble vide trouve son intérêt :

— on cherche à comparer un ensemble donné à l'ensemble vide

— l'ensemble vide peut être le conteneur de base d'un ensemble qui va grossir

Attention, la notation `{}` ne doit pas être utilisée pour désigner l'ensemble vide [3].

Test d'appartenance à un ensemble

La possibilité de tester l'appartenance d'un objet à un ensemble est une propriété fondamentale d'un ensemble et qui en fait tout l'intérêt. On peut tester avec l'opérateur `in` l'appartenance d'un objet donné à un ensemble donné :

2. Heureusement, sinon on ne pourrait pas créer, par exemple, d'ensemble ayant un très grand nombre d'éléments.

3. L'expression `{}` désigne un dictionnaire vide.

```
s = {65, 31, 12, 81, 81, 65, 81, 12}

print(42 in s)
print(42 not in s)
```

```
False
True
```

Un ensemble est une structure beaucoup plus efficace qu'une liste pour savoir si un objet appartient à un groupe d'éléments. Par exemple, si des objets sont rangés dans une liste L de longueur n, la recherche de l'appartenance d'un élément x à L peut nécessiter jusqu'à n comparaisons. En revanche, si les objets sont placés dans un ensemble, la nature de la structure de données qu'est un ensemble rend cette recherche efficace [4].

Type des éléments d'un ensemble

Les éléments d'un ensemble ne sont pas nécessairement de même type :

```
s = {42, "orange", "rose", 42, 2038}
print(s)
```

```
{'rose', 42, 'orange', 2038}
```

La documentation Python, ne mentionne aucune recommandation sur la nature hétérogène ou homogène des éléments d'un ensemble.

La nature hachable des éléments d'un ensemble

Les éléments d'un ensemble ne peuvent être quelconques : ils doivent appartenir à la catégorie des objets dit *hachables*. C'est le langage qui définit les objets qui sont hachables ou non. Par exemple, une liste n'est pas hachable et donc, un ensemble ne peut contenir aucune liste parmi ses éléments :

```
s = {[42, 81, 10], 100}
print(s)
```

```
TypeError: unhashable type: 'list'
```

Les objets suivants sont toujours hachables :

— les entiers,
— les chaînes de caractères,
— les tuples s'ils sont formés d'éléments eux-mêmes hachables.

Un élément de type `set` n'est jamais hachable. On ne peut donc considérer un ensemble d'ensembles ni même un ensemble de listes.

Utiliser des tuples

Un tuple étant hachable (au moins en première approximation), on peut transformer l'exemple ci-dessus pour obtenir un ensemble valide en utilisant un tuple plutôt qu'une liste :

4. En O(1).

```
1 s = {(42, 81, 10), 100}
2 print(s)
```

```
3 {100, (42, 81, 10)}
```

— Ligne 3 : s possède deux éléments : un entier et un tuple de trois entiers.

Ensemble élément d'un ensemble

Un ensemble de type `set` n'étant pas hachable, il ne peut être élément d'un autre ensemble :

```
1 a = set([4, 6])
2 b = set([5, 8])
3 c = set([1, 3])
4 s = set([a, b, c])
```

```
5 TypeError: unhashable type: 'set'
```

Donc pas d'ensemble des parties d'un ensemble en Python comme cela existe en mathématiques ...

Un ensemble est mutable

Un ensemble de type `set`, de même qu'une liste, est un type mutable : un ensemble peut être modifié par complétion ou suppression d'éléments [5] :

```
1 a = set([4, 2, 1])
2 print(a)
3
4 a.remove(1)
5 print(a)
6
7 a.add(100)
8 print(a)
```

```
9 {1, 2, 4}
10 {2, 4}
11 {100, 2, 4}
```

— Ligne 1 : l'ensemble a va être modifié

— Ligne 4 : un ensemble possède une méthode permettant de retirer un élément ; ici, on retire l'élément 1 à l'ensemble a

— Ligne 7 : un ensemble possède une méthode permettant d'ajouter un élément ; ici, on adjoint l'élément 100 à l'ensemble a.

Parcours d'un ensemble

Un ensemble est un itérable et en particulier, on peut parcourir un ensemble avec une boucle `for` :

5. En particulier, un ensemble dispose d'un certain nombre de méthodes dites *mutatrices* qui modifient un ensemble donné ; par exemple, la méthode `add` est une méthode mutatrice.

```
1 s = {65, 31, 12, 81, 81, 65, 81, 12}
2 for x in s:
3     print(x)
```

```
4 81
5 31
6 12
7 65
```

À la différence d'une liste, un ensemble n'est pas une séquence qui posséderait un ordre canonique de parcours. L'ordre de parcours d'un ensemble est non spécifié par le langage.

Ensembles en compréhension

Il existe un analogue des listes en compréhension pour les ensembles : les ensembles en compréhension.

```
1 A={10 * i for i in range(1,11)}
2 print(A)
```

```
3 {100, 70, 40, 10, 80, 50, 20, 90, 60, 30}
```

— Ligne 1 : `A` est défini par un ensemble en compréhension.

La syntaxe de l'ensemble en compréhension est calquée sur celle des listes en compréhension. En particulier, il est possible d'utiliser une clause `if` :

```
1 A={10 * i for i in range(1,11) if i % 3 == 0}
2 print(A)
```

```
3 {90, 60, 30}
```

Valeur booléenne d'un ensemble

On sait qu'une liste n'ayant aucun élément a une valeur booléenne `False`. De même, l'ensemble vide a une valeur booléenne `False` :

```
1 empty = set()
2 print(bool(empty))
3
4 non_empty = set([42, 2020])
5 print(bool(non_empty))
```

```
6 False
7 True
```

En revanche, un ensemble non vide a une valeur booléenne `True`.

Ensembles construits sur un itérable

On peut construire un ensemble à partir d'une liste :

```
1 L = [65, 31, 12, 81, 81, 65, 81, 12]
2 s = set(L)
```

mais en fait à partir de n'importe quel itérable comme une chaîne :

```
1 s = "BANANA"
2 S = set(s)
3 print(S)
```

```
4 {'A', 'N', 'B'}
```

— Lignes 2 et 4 : S est l'ensemble dont les éléments sont les lettres distinctes de la chaîne s : la chaîne s est de longueur 6 mais l'ensemble S est de longueur 3.

Les objets de type `range(n)` sont aussi des itérables et permettent donc de créer des ensembles :

```
1 r = range(10)
2 S = set(r)
3 print(S)
```

```
4 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

2 Doublons

Présence de doublons dans une liste

Comme un ensemble est une structure de données dont les éléments ne sont jamais répétés, un ensemble donne le moyen d'extraire les éléments distincts d'une liste. Un ensemble permet aussi de savoir si une liste L admet des répétitions (ce qu'on appelle des « doublons ») :

```
1 L = [65, 31, 12, 81, 81, 65, 81, 12]
2 print(L)
3 # L est-elle sans doublons ?
4 print(len(L) == len(set(L)))
5 print()
6
7 L = [65, 31, 12, 81]
8 print(L)
9 # L est-elle sans doublons ?
10 print(len(L) == len(set(L)))
```

```
11 [65, 31, 12, 81, 81, 65, 81, 12]
12 False
13
14 [65, 31, 12, 81]
15 True
```

— Ligne 1 : une liste qui contient des doublons, par exemple 81 (qui apparaît trois fois).

— Lignes 4 et 12 : ce test montre que L contient des doublons.

— Lignes 7 et 15 : L'ensemble défini par L est sans doublon.

Ainsi, on vient d'observer la **technique du test de doublon** : pour savoir si une liste `L` de longueur `n` contient des doublons il suffit de construire l'ensemble associé à `L` et examiner si cet ensemble admet également `n` éléments. Si ce n'est pas le cas, c'est que `L` admet des doublons.

Conversion d'un ensemble en liste

A tout ensemble Python est associée une liste Python qui représente la liste des éléments de l'ensemble. Pour générer la liste associée à un ensemble on utilise le constructeur `list` :

```
1 s = {65, 31, 12, 81, 81, 65, 81, 12}
2 print(s)
3 L = list(s)
4 print(L)
5 print(s)
```

```
6 {81, 31, 12, 65}
7 [81, 31, 12, 65]
8 {81, 31, 12, 65}
```

— Lignes 3 et 7 : un ensemble peut être converti en liste avec le constructeur `list` : les éléments de l'ensemble sont rangés dans la liste dans un ordre indéterminé.

— Lignes 2, 5, 6 et 8 : convertir un ensemble en liste crée une liste et ne modifie pas l'ensemble.

Suppression des doublons d'une liste

On dispose d'une liste `L` et on souhaite créer une nouvelle liste `M` à partir des éléments de `L` mais en sorte que la liste `M` soit sans répétition. Par exemple, si

```
L = [65, 31, 12, 81, 81, 65, 81, 12]
```

alors `M = [65, 12, 81, 31]`.

Voici comment on obtient la liste sans doublon :

```
1 L = [65, 31, 12, 81, 81, 65, 81, 12]
2 print(L)
3 M = list(set(L))
4 print(M)
```

```
5 [65, 31, 12, 81, 81, 65, 81, 12]
6 [65, 12, 81, 31]
```

On a converti la liste en ensemble et reconverti le résultat en liste.

3 Méthodes mutatrices

Les méthodes `add` et `update` d'un ensemble

On peut compléter un ensemble par ajout d'un élément :

```
1 s = {65, 31, 12, 81, 81, 65, 81, 12}
2 print(s)
3
4 z = s.add(42)
5 print(s)
6 print(z)
7
8 s.add(42)
9 print(s)
```

```
10 {81, 31, 12, 65}
11 {81, 42, 31, 12, 65}
12 None
13 {81, 42, 31, 12, 65}
```

— Ligne 4 : on peut « ajouter » un élément à un ensemble avec la méthode `add`.
— Lignes 6 et 12 : la méthode `add` ne renvoie rien, elle se contente de modifier l'ensemble.
— Lignes 8-9 : il est possible d'ajouter un élément déjà présent : l'ensemble est alors inchangé.

Si on veut ajouter **plusieurs** éléments à un ensemble `s`, on peut

— répéter l'insertion avec la méthode `s.add` comme plus haut ;
— utiliser la méthode `s.update` pour faire l'opération en une fois, comme expliqué ci-essous.

Si `s` est un ensemble, la méthode `s.update` permet même de compléter l'ensemble `s` par les éléments d'un itérable `t`, cet itérable n'étant pas forcément un ensemble, cela peut être une liste comme dans l'exemple ci-dessous :

```
s = {4, 5, 6}
t = [40, 50, 60, 70]

s.update(t)
print(s)
```

```
{4, 5, 6, 70, 40, 50, 60}
```

Noter que l'appel `s.update(t)` modifie `s` mais ne le renvoie pas (ça renvoie même `None`).

Retirer un élément avec la méthode `discard`

Un appel `s.discard(a)` retire l'élément `a` de l'ensemble `s` si `a` est présent dans `s` et sinon, l'appel est sans effet :

```
1 s = {65, 31, 12, 81, 81, 65, 81, 12}
2
3 z = s.discard(81)
4 print(s)
5 print(z)
6
7 s.discard(81)
8 print(s)
```

```
{31, 12, 65}
None
{31, 12, 65}
```

— Ligne 1 : la méthode `discard` retire un élément s'il est présent dans un ensemble.

— Lignes 5 et 10 : la méthode `discard` ne retourne rien et se contente de modifier l'ensemble.

— Lignes 7 et 11 : il est possible de tenter de retirer un élément non présent dans l'ensemble. L'appel laisse l'ensemble inchangé.

Retirer un élément avec la méthode `remove`

La méthode `remove` permet de retirer un élément donné d'un ensemble donné :

```
s = {65, 31, 12, 81, 81, 65, 81, 12}
print(s)
z = s.remove(81)
print(s)
print(z)
```

```
{81, 31, 12, 65}
{31, 12, 65}
None
```

Si on tente de retirer avec la méthode `remove`, un élément `a` d'un ensemble `s` qui ne contient pas `a`, l'appel déclenche une erreur :

```
s = {65, 31, 12, 81, 81, 65, 81, 12}
print(s)
s.remove(42)
```

```
{81, 31, 12, 65}
Traceback (most recent call last):
  File "_.py", line 3, in <module>
    s.remove(42)
KeyError: 42
```

— 42 n'est pas dans l'ensemble `s`.

La méthode `pop`

La méthode pop retire un élément arbitraire d'un ensemble :

```
s = {65, 31, 12, 81, 81, 65, 81, 12}
z = s.pop()

print(s)
print(z)
```

```
{31, 12, 65}
81
```

— Lignes 2, 4 et 6 : la méthode pop retire un élément arbitraire d'un ensemble
— Lignes 2 et 5 : la méthode pop renvoie l'élément qu'elle retire. La méthode pop est la seule façon d'accéder à un élément d'un ensemble.

La méthode pop retire un élément arbitraire et inconnu au moment de l'appel. En fait, la méthode pop renvoie l'élément qu'elle retire de l'ensemble et l'usage essentiel de cette méthode est plutôt d'extraire un élément, a priori inconnu, d'un ensemble donné.

Méthode pop et ensemble vide

Un appel de la méthode pop sur l'ensemble vide entraîne la levée d'une exception :

```
1 s = {42}
2 s.pop()
3 s.pop()
4 print(s)
```

```
5 KeyError: 'pop from an empty set'
```

— Ligne 2 : on a retiré 42 de l'ensemble s, l'ensemble est donc vide
— Ligne 3 : on tente de retirer un élément d'un ensemble vide : le programme est interrompu (ligne 5)

4 Opérations entre ensembles

Panorama des opérations sur les ensembles

On peut faire les opérations mathématiques usuelles sur les ensembles.

Opérations de comparaison

— *égalité* : deux ensembles sont-ils égaux ?
— *disjonction* : deux ensembles n'ont-ils aucun élément commun ?
— *contenance* : un ensemble est-il contenu dans l'autre ?

La contenance admet deux syntaxes : opérateurs et méthodes.

Opérations binaires sur des ensembles

— l'intersection,
— la réunion
— la différence
— la différence symétrique

Les quatre opérations ci-dessus admettent deux syntaxes utilisant :

— soit un opérateur,
— soit une méthode.

Et en outre, chacune des quatre opérations ci-dessus peut être utilisée dans une version qui modifie un des deux ensembles, et là encore sous deux syntaxes (opérateur et méthode).

	Intersection	Réunion	Différence	Différence symétrique
opérateur	`&`	`|`	`-`	`^`
méthode	intersection	union	difference	`symmetric_difference`
opérateur augmenté	`&=`	`|=`	`-=`	`^=`
méthode mutatrice	`intersection_update`	`update`	`difference_update`	`symmetric_difference_update`

Égalité d'ensembles

On dispose de deux ensembles `A` et `B` et on cherche à savoir si les ensembles `A` et `B` ont mêmes éléments ie si mathématiquement $A = B$.

Pour comparer deux ensembles, on utilise les opérateurs `==` et `!=` :

```
s = {1, 2, 1, 3}
t = {2, 2, 1, 3, 3}
print(s)
print(t)
print(s == t)
print(s != t)
```

```
{1, 2, 3}
{1, 2, 3}
True
False
```

Inclusion et contenance entre ensembles

Le tableau ci-dessous présente :

— les quatre opérateurs de comparaison, large ou stricte, entre ensembles

— les deux méthodes testant l'inclusion ou la contenance large.

Table : Opérateurs et méthodes de comparaison entre ensembles

Comparaison	Notation mathématique	Opérateur Python	Méthode
A est inclus dans B au sens large	$A \subseteq B$	`A <= B`	A.issubset(B)
A contient B au sens large	$A \supseteq B$	`A >= B`	A.issuperset(B)
A est strictement inclus dans B	$A \subsetneq B$	`A < B`	
A contient strictement B	$A \supsetneq B$	`A > B`	

Les appels de méthodes et les opérateurs renvoient des booléens.

Illustration avec les opérateurs

```
A = {1, 2, 1, 3}
B = {0, 2, 2, 1, 3, 3}
print(A)
print(B)

print("A <= B","->", A <= B)
print("A < B","->", A < B)
print("A >= B","->", A >= B)
print("A > B","->", A >= B)
```

```
{1, 2, 3}
{0, 1, 2, 3}
A <= B -> True
A < B -> True
A >= B -> False
A > B -> False
```

Illustration avec les méthodes

```
A = {1, 2, 1, 3}
B = {0, 2, 2, 1, 3, 3}
print(A)
print(B)

print("A.issubset(B)","->", A.issubset(B))
print("A.issuperset(B)","->", A.issuperset(B))
```

```
{1, 2, 3}
{0, 1, 2, 3}
```

```
10 A.issubset(B) -> True
11 A.issuperset(B) -> False
```

Ensembles disjoints

Deux ensembles sont dits *disjoints* s'ils n'admettent aucun élément commun. Par exemple, les ensembles $\{10, 20, 30\}$ et $\{4, 5, 6, 7\}$ sont disjoints.

La méthode `isdisjoint`

Pour tester en Python le caractère disjoint de deux ensembles A et B, on utilise la méthode `isdisjoint` :

```
1 A = {10, 20, 30}
2 B = {4, 5, 6, 7}
3 A.isdisjoint(B)
```

```
4 True
```

Alternative avec l'opérateur d'intersection

Il serait possible aussi d'utiliser l'opérateur d'intersection d'ensembles et de tester s'il renvoie l'ensemble vide :

```
1 A = {10, 20, 30}
2 B = {4, 5, 6, 7}
3 print(A)
4 print(B)
5
6 inter = A & B
7 sontDisjoints = True
8
9 if len(inter) != 0:
10     sontDisjoints = False
11
12 print(sontDisjoints)
```

```
13 {10, 20, 30}
14 {4, 5, 6, 7}
15 True
```

— Ligne 6 : `inter` est l'ensemble vide.

— Lignes 9-10 : que A et B soient disjoints signifie que `inter` est l'ensemble vide et donc `len(inter)` vaut 0.

— Ligne 9 : on aurait pu abréger la ligne en `if inter:`

Réunion et intersection

L'intersection

L'intersection de deux ensembles `s` et `t` est l'ensemble formé des éléments appartenant aux deux ensembles `s` et `t`. Par exemple, l'intersection des ensembles $s = \{1, 2, 3, 4\}$ et $t = \{1, 2, 5, 6\}$ est l'ensemble $\{1, 2\}$.

La réunion

La réunion de deux ensembles `s` et `t` est l'ensemble formé des éléments appartenant à l'un des deux ensembles `s` ou `t`. Par exemple, la réunion des ensembles $s = \{1, 2, 3, 4\}$ et $t = \{1, 2, 5, 6\}$ est l'ensemble $\{1, 2, 3, 4, 5, 6\}$.

Réunion et intersection en Python

Étant donné deux ensembles, pour chaque opération d'intersection ou de réunion, il existe de 4 procédés pour exécuter l'opération :

— soit en utilisant un opérateur
— soit en utilisant une méthode non mutatrice
— soit en utilisant une méthode mutatrice
— soit en utilisant une affectation augmentée

Le tableau ci-dessous résume les différentes possibilités :

	Intersection	Union	Remarques
Notation mathématique	$A \cap B$	$A \cup B$	
Éléments appartenant ...	... à A ET à B	... à A OU à B	
Opérateur Python	`A & B`	`A | B`	Un nouvel ensemble est créé
Méthode non mutatrice	`A.intersection(B)` `B.intersection(A)`	`A.union(B)` `B.union(A)`	Renvoie l'ensemble cherché sans modifier ni `A` ni `B`
Méthode mutatrice	`A.intersection_update(B)` `B.intersection_update(A)`	`A.update(B)` `B.update(A)`	Modifie l'ensemble en attribut
Affectation augmentée	`A &= B` `B &= A`	`A |= B` `B |= A`	Modifie l'ensemble référencé à gauche

Chaque catégorie est illustrée ci-dessous par des exemples.

Opérateurs d'intersection et de réunion

```
1 s = {1, 2, 3, 4}
2 t= {1, 2, 5, 6}
3 print(s)
4 print(t)
5 print()
6
7 print("intersection ->", s & t)
8 print("union ->", s | t)
```

```
9 {1, 2, 3, 4}
10 {1, 2, 5, 6}
11
12 intersection -> {1, 2}
13 union -> {1, 2, 3, 4, 5, 6}
```

En Python, l'intersection de deux ensembles `s` et `t` est `s & t`. Le choix de l'opérateur `&` (qui désigne une abréviation de *et*) rappelle que les éléments de l'intersection de `s` et de `t` sont les éléments appartenant à `s` **et** à `t`. Il est possible de prendre l'intersection `r & s & t` de trois ensembles (ou de plus de trois ensembles).

En Python, la réunion de deux ensembles `s` et `t` est `s | t`. Le choix de l'opérateur `|` (qui, en programmation, désigne souvent un *ou* logique ou bit à bit) rappelle que les éléments de l'intersection de `s` et de `t` sont les éléments appartenant à `s` **ou** à `t`. Il est possible de prendre la réunion `r | s | t` de trois ensembles (ou de plus de trois ensembles).

Méthodes non mutatrices `intersection` et `union`

```
1 s = {1, 2, 3, 4}
2 t= {1, 2, 5, 6}
3 print(s)
4 print(t)
5 print()
6
7 print("intersection ->", s.intersection(t))
8 print("union ->", t.union(s))
9 print(s)
10 print(t)
```

```
11 {1, 2, 3, 4}
12 {1, 2, 5, 6}
13
14 intersection -> {1, 2}
15 union -> {1, 2, 3, 4, 5, 6}
16 {1, 2, 3, 4}
17 {1, 2, 5, 6}
```

L'intersection de deux ensembles `s` et `t` est `s.intersection(t)` ou encore `t.intersection(s)`. De même, `s.union(t)` est la réunion de `s` et de `t`.

La méthode `intersection` appliquée à `s` ne modifie pas `s` (comparer lignes 16 et11). La méthode

`intersection` renvoie l'intersection des deux ensembles.

C'est analogue pour la méthode `union`.

Méthodes mutatrices `intersection_update` et `update`

```
1 s = {1, 2, 3, 4}
2 t= {1, 2, 5, 6}
3 print(s)
4 print(t)
5 print()
6
7 s.intersection_update(t)
8 print("intersection ->", s)
9 print("----------------------------")
10 s = {1, 2, 3, 4}
11 s.update(t)
12 print("union ->", s)
```

```
13 {1, 2, 3, 4}
14 {1, 2, 5, 6}
15
16 intersection -> {1, 2}
17 ----------------------------
18 union -> {1, 2, 3, 4, 5, 6}
```

La méthode `intersection` appliquée à `s` *modifie* le contenu de `s` (comparer lignes 13 et 16). La méthode `intersection` (ligne 7) ne renvoie rien (ou plutôt `None`).

C'est analogue pour la méthode `union`.

Affectation augmentée pour l'intersection et la réunion

L'ensemble qui est à gauche de l'affectation augmentée est modifié :

```
1 s = {1, 2, 3, 4}
2 t= {1, 2, 5, 6}
3 print(s)
4 print(t)
5 print()
6
7 s &= t
8 print("intersection ->",s)
9 s |= t
10 print("union ->", s)
```

```
11 {1, 2, 3, 4}
12 {1, 2, 5, 6}
13
14 intersection -> {1, 2}
15 union -> {1, 2, 5, 6}
```

Différences d'ensembles

Mathématiquement, il existe deux types d'opérations de différences entre deux ensembles A et B :

— la différence simple $A - B$ encore notée $A \setminus B$ en mathématiques
— la différence symétrique notée $A \Delta B$ en mathématiques.

Étant donné deux ensembles, pour chacune des deux opérations de différence, il existe 3 procédés pour exécuter l'opération :

— soit en utilisant un opérateur
— soit en utilisant une méthode modifiant l'un des ensembles
— soit en utilisant une affectation augmentée modifiant l'un des ensembles

Le tableau ci-dessous résume les différentes possibilités :

	Différence simple	Différence symétrique
Opération mathématique	$A - B$ ou $A \setminus B$	$A \Delta B$
Éléments appartenant	à A mais pas à B	soit à A soit à B
Exemple $A = \{1, 2, 5\}$ et $B = \{1, 2, 3, 4\}$	$\{5\}$	$\{5, 3, 4\}$
Opérateur Python	`A - B`	`A ^ B`
Méthode de `set`	`A.difference(B)`	`A.symmetric_difference(B)`
Opérateur augmenté	`A -= B`	`A ^= B`

L'opérateur `^` est parfois appelé `XOR` pour *exclusive or* qui correspond à *soit ... soit ...* c'est-à-dire un *ou* exclusif.

Opérateurs de différence : exemple

```
1 A = {1, 2, 5}
2 B = {1, 2, 3, 4}
3 print(A)
4 print(B)
5 print()
6
7 print("A - B ->", A - B)
8 print("A ^ B ->", A ^ B)
```

```
9 {1, 2, 5}
10 {1, 2, 3, 4}
11
```

```
A - B -> {5}
A ^ B -> {3, 4, 5}
```

Méthode pour la différence : exemple

```
A = {1, 2, 5}
B = {1, 2, 3, 4}
print(A)
print(B)
print()

print("A.difference(B) ->", A.difference(B))
print("A.symmetric_difference(B) ->", A.symmetric_difference(B))
print()
print(A)
print(B)
```

```
{1, 2, 5}
{1, 2, 3, 4}

A.difference(B) -> {5}
A.symmetric_difference(B) -> {3, 4, 5}

{1, 2, 5}
{1, 2, 3, 4}
```

Ces méthodes créent de nouveaux ensembles et ne modifient pas les ensembles sur lesquels elles agissent.

Affectation augmentée : exemple

L'ensemble qui est à gauche de l'affectation augmenté est modifié :

```
A = {1, 2, 5}
B = {1, 2, 3, 4}
print(A)
print(B)
print()
A -= B
print("A - B ->", A)
print("----------------------")
A = {1, 2, 5}
B = {1, 2, 3, 4}
print(A)
print(B)
print()
A ^= B
print("A ^ B ->", A)
```

```
{1, 2, 5}
{1, 2, 3, 4}

A - B -> {5}
----------------------
{1, 2, 5}
{1, 2, 3, 4}

A ^ B -> {3, 4, 5}
```

Comparaison des différentes opérations sur des ensembles `set`

Pour chacune des opérations mathématiques suivantes

— l'intersection,
— la réunion,
— la différence
— la différence symétrique

Python propose deux façons de la réaliser avec des ensembles de type `set` [6]. Comme le principe est le même pour chaque opération, seul le cas de la réunion sera présenté.

Création d'un nouvel ensemble

Soient `s` et `t` des ensembles. Alors `s | t` est un *nouvel* ensemble qui contient des références vers les éléments de la réunion. Il en est de même de l'ensemble `s.union(t)`.

L'intérêt est que les références vers les objets de `s` et de `t` sont préservées. La contrepartie est que des références vers des objets de `s` et de `t` sont recopiées :

```
s = {1, 2, 3, 4}
t= {1, 2, 5, 6}


print("s :", s)
print("t :", t)
print()
print("id(s)->", id(s))
print("id(t)->", id(t))
print()
u = s | t

print("réunion :", u)
print()
print("id(s | t) :", id(u))
print()
print("id(s)->", id(s))
print("id(t)->", id(t))

```

6. Le cas de frozenset est légèrement différent.

```
20 print()
21 print("s :", s)
22 print("t :", t)
```

```
23 s : {1, 2, 3, 4}
24 t : {1, 2, 5, 6}
25
26 id(s)-> 3072177852
27 id(t)-> 3071850172
28
29 réunion : {1, 2, 3, 4, 5, 6}
30
31 id(s | t) : 3071851516
32
33 id(s)-> 3072177852
34 id(t)-> 3071850172
35
36 s : {1, 2, 3, 4}
37 t : {1, 2, 5, 6}
```

— Lignes 8 et 9 : les identités de `s` et `t` avant l'opération de réunion.
— Lignes 11-15 : un nouvel objet est créé par le renvoi de `s | t`
— Lignes 15 et 16 : les identités de `s` et `t` après l'opération de réunion : elles sont inchangées, cf. lignes 26 et 27.
— Lignes 36 et 37 : les contenus de `s` et `t` après l'opération de réunion : ils sont inchangés, cf. lignes 23 et 24.

La situation est essentiellement la même pour construire la réunion de `s` et de `t` en utilisant la méthode `s.union` :

```
1 s = {1, 2, 3, 4}
2 t= {1, 2, 5, 6}
3
4
5 print("s :", s)
6 print("t :", t)
7 print()
8 print("id(s)->", id(s))
9 print("id(t)->", id(t))
10 print()
11 u=s.union(t)
12
13 print("réunion :", u)
14 print()
15 print("id(s.union(t)) :", id(u))
16 print()
17 print("id(s)->", id(s))
18 print("id(t)->", id(t))
19
20 print()
```

```
print("s :", s)
print("t :", t)
```

```
s : {1, 2, 3, 4}
t : {1, 2, 5, 6}

id(s)-> 3072751292
id(t)-> 3072423612

réunion : {1, 2, 3, 4, 5, 6}

id(s.union(t)) : 3072424956

id(s)-> 3072751292
id(t)-> 3072423612

s : {1, 2, 3, 4}
t : {1, 2, 5, 6}
```

— On notera que la réunion de `s` et de `t` est obtenue par une méthode appliquée à `s` sans que `s` soit modifié : `s` référence le même objet et le contenu de cet objet est inchangé

Altération d'un des ensembles initiaux

Soient `s` et `t` des ensembles. Alors les instructions :

— `s.update(t)`

— `s |= t`

sont totalement équivalentes du point de vue de la création ou préservation d'objets. Elles conservent l'objet `s` mais en modifient le contenu. Le bilan est le suivant :

— l'avantage est qu'il y a moins d'ajout de références que si un nouvel objet était complètement recréé.

— l'inconvénient est que le contenu initial de `s` est perdu. L'objet `t`, lui, n'est en rien modifié.

```
s = {1, 2, 3, 4}
t= {1, 2, 5, 6}


print("s :", s)
print("t :", t)
print()
print("id(s)->", id(s))
print("id(t)->", id(t))
print()
s.update(t)

print("réunion :", s)

print("id(s)->", id(s))
```

```
16 print("id(t)->", id(t))
17
18 print()
19 print("s :", s)
20 print("t :", t)
```

```
21 s : {1, 2, 3, 4}
22 t : {1, 2, 5, 6}
23
24 id(s)-> 3072874172
25 id(t)-> 3072550588
26
27 réunion : {1, 2, 3, 4, 5, 6}
28 id(s)-> 3072874172
29 id(t)-> 3072550588
30
31 s : {1, 2, 3, 4, 5, 6}
32 t : {1, 2, 5, 6}
```

— Ligne 8 et 9 : les identités de `s` et `t` avant l'opération de réunion.

— Ligne 11 : la construction de la réunion de `s` et `t` : aucun *nouvel* ensemble n'est créé.

— Lignes 28 et 29 : les identités de `s` et `t` après l'opération de réunion : elles sont inchangées, cf. lignes 24 et 25.

— Lignes 36 et 37 : le contenu de `s` a changé : désormais, `s` représente la réunion des ensembles initiaux. Le premier contenu de `s` est perdu. Le contenu de `t`, lui, est inchangé.

Conversion implicite lors de l'appel de certaines méthodes

Si `s` et `t` sont des ensembles, `s.union(t)` renvoie la réunion des deux ensembles. En réalité, une expression `s.union(t)` ne suppose pas que `t` soit de type ensemble mais, en fait, que `t` soit n'importe quel conteneur susceptible d'être converti en un objet de type `set` :

```
1 s = set("ROSES")
2 t= "ORANGES"
3 print(s.union(t))
```

```
4 {'E', 'R', 'S', 'A', 'N', 'O', 'G'}
```

La conversion s'applique aux opérations mathématiques et aux méthodes de `set` données dans le tableau suivant :

opération	méthode non mutatrice	méthode mutatrice
intersection	`intersection`	`intersection_update`
réunion	`union`	`update`
différence	`difference`	`difference_update`
différence symétrique	`symmetric_difference`	`symmetric_difference_update`

5 Divers

Création, réunion d'ensembles par décompactage

L'exemple qui suit va permettre de mieux comprendre la possibilité de création d'ensemble par décompactage de certains itérables déjà créés :

```
1 s1 = {10, 12, 14}
2 s2 = {11, 13, 15}
3 L = [23, 24, 25]
4
5 s= {20, *s1, *range(16, 20), 22, *L, 21, *s2}
6
7 print(s)
```

```
8 {10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

Depuis la version 3.5 de Python, il est possible de construire un ensemble en plaçant entre accolades et séparés par des virgules des éléments de cet ensemble ainsi que des références à des itérables à décompacter avec la syntaxe `*`. Dans l'exemple ci-dessus, ligne 5, on construit un ensemble `s` dont les éléments sont

— les entiers 20, 22 et 21

— les éléments de l'ensemble `s1`

— les éléments de l'itérable `range(16, 20)`, donc les entiers de 16 à 19

— les éléments de la liste `L`

— les éléments de l'ensemble `s2`

La syntaxe `*it` appliquée à un itérable telle qu'une liste, un ensemble, etc. à pour effet de placer dans l'ensemble les éléments de l'itérable. C'est juste du sucre syntaxique.

Un cas particulier de ce qui précède est l'obtention de la réunion de deux ensembles `s1` et `s2` par `s = {*s1, *s2}`.

Faux éléments différents dans un ensemble

Un ensemble est formé d'éléments qui n'ont jamais la *même valeur* c'est-à-dire que si `x` et `y` sont deux éléments distincts de la liste des éléments d'un ensemble `s` alors on a `x !== y`. Par conséquent, si on définit l'ensemble

```
1 s = {1, 42, True, 0, 1.0, False}
```

alors, cet ensemble ne contient pas six éléments mais en fait uniquement **trois**, à savoir 42, 1 et 0.

En effet, en Python, on a ceci :

```
1 >>> 1 == True == 1.0
2 True
3 >>>
```

Ces trois éléments, bien qu'ayant des types différents, ont même valeur. De même, `0` et `False` ont même valeur.

Copie d'ensemble

Pour copier un ensemble, on utilise le constructeur `set` :

```
1 A = {1, 2, 5}
2 B=set(A)
3 print("A =", A)
4 print("B =", B)
5 print("-"*10)
6
7 A.remove(1)
8 print("A =", A)
9 print("B =", B)
10 print("-"*10)
11
12 B.remove(2)
13 print("A =", A)
14 print("B =", B)
15 print("-"*10)
```

```
17 A = {1, 2, 5}
18 B = {1, 2, 5}
19 ----------
20 A = {2, 5}
21 B = {1, 2, 5}
22 ----------
23 A = {2, 5}
24 B = {1, 5}
25 ----------
```

Vider un ensemble de ses éléments

Au cours d'opérations avec un ensemble `s`, on peut avoir besoin de supprimer tous les éléments de `s` tout en gardant une référence à `s` (plutôt que de créer un nouvel ensemble). Pour cela, on utilise la méthode `clear` :

```
1 s = {65, 31, 12, 81, 81, 65, 81, 12}
2 print(s)
3 s.clear()
4 print(s)
```

```
5 {81, 31, 12, 65}
6 set()
```

— Lignes 3 et 6 : après avoir appelé la méthode `clear`, `s` référence l'ensemble vide.

Itérateur sur un ensemble

Si s est un ensemble, on obtient un itérateur sur s en utilisant la fonction standard `iter` :

```
s = {65, 31, 12, 81, 81, 65, 81, 12}
it = iter(s)
print(list(it))
```

```
[81, 31, 12, 65]
```

`iter(d)` itère sur les éléments de l'ensemble dans le même ordre, que lorsqu'on parcourt s avec une boucle `for` :

```
s = {65, 31, 12, 81, 81, 65, 81, 12}
print(list(s))
it = iter(s)

for i in range(len(s)):
    print(next(it))
```

```
[[81, 65, 12, 31]
81
65
12
31
```

Accès à un élément arbitraire d'un ensemble

Si un ensemble n'est pas vide, comment accéder à un élément arbitraire de cet ensemble ? Pour y parvenir de façon économique et simple, il suffit de créer un itérateur sur l'ensemble et d'itérer juste une seule fois :

```
s = {65, 31, 12, 81, 81, 65, 81, 12}

if s:
    it = iter(s)
    print(next(it))
```

```
81
```

La méthode pop

La méthode pop a un effet analogue en apparence puisqu'elle permet d'accéder à un élément arbitraire d'un ensemble ; toutefois la méthode pop retire l'élément de l'ensemble (ce qui parfois n'est pas souhaité) :

```
s = {65, 31, 12, 81, 81, 65, 81, 12}
print(s)
print(s.pop())
print(s)
```

```
{81, 31, 12, 65}
81
{31, 12, 65}
```

Parcourir en modifiant un ensemble

Il n'est pas possible de modifier un ensemble alors que l'ensemble est parcouru par une boucle `for` :

```
S = {2020}
for z in S:
    S.add(42)
```

```
Traceback (most recent call last):
  File "_.py", line 2, in <module>
    for z in S:
RuntimeError: Set changed size during iteration
```

Implémentation des ensembles

Les ensembles que Python utilise permettent d'accéder à un de ses éléments en un temps de `O(1)`, autrement dit instantanément.

Les ensembles de l'implémentation CPython sont en fait des tables de hachage à adressage ouvert, et utilisent un algorithme de Knuth du tome 3 de AOCP, cf. le code source en C.

6 `frozenset`

Les ensembles de type `frozenset`

Les ensembles de type `frozenset` représentent une version immutable des ensembles de type `set`. À la différence de `set`, il n'existe pas d'ensembles littéraux de type `frozenset`. Donc, pour créer un ensemble de type `frozenset`, il faut impérativement utiliser le constructeur `frozenset`, qui est analogue au constructeur `set` :

```
L = [65, 31, 12, 81, 81, 65, 81, 12]
s = frozenset(L)
print(s)
print(len(s))
```

```
frozenset({65, 12, 81, 31})
4
```

Les ensembles de type `frozenset` ont un comportement analogue aux ensembles de type `set`.

Les opérateurs et les méthodes d'union, intersection, différence et différence symétrique sont inchangés. Par exemple :

```
A = frozenset({1, 2, 5})
B = frozenset({1, 2, 3, 4})
print(A - B)
print(A.difference(B))

A -= B
print(A)
```

```
frozenset({5})
frozenset({5})
frozenset({5})
```

Naturellement, comme les instances de `frozenset` sont immutables, aucune méthode mutatrice de la classe `set` ne s'applique à la classe `frozenset` :

```
a = frozenset({1, 2, 5})
a.remove(1)
```

```
Traceback (most recent call last):
  File "_.py", line 2, in <module>
    a.remove(1)
AttributeError: 'frozenset' object has no attribute 'remove'
```

Un ensemble de type `frozenset` est hachable et peut donc être élément d'un ensemble de type `set` ou `frozenset` (alors qu'un élément de type set ne peut être membre d'aucun ensemble) :

```
a = frozenset([4, 6])
b = frozenset([5, 8])
c = frozenset([1, 3])
s = set([a, b, c])
print(s)
```

```
{frozenset({1, 3}), frozenset({8, 5}), frozenset({4, 6})}
```

Comparaison des différentes opérations sur des `frozenset`

Comme pour les ensembles de types `set`, si `s` et `t` sont des frozenset, les instructions `s | t` et `s.union(t)` vont créer un nouvel objet de type frozenset correspondant à la réunion des deux ensembles.

Le comportement entre des ensembles de type frozenset et set va différer pour l'affectation augmentée.

Pour des objets de type `set`, une opération augmentée telle que `s &= t` modifie le contenu de `s` pour qu'il corresponde à la réunion des deux ensembles mais elle ne modifie pas l'identité de objet `s` :

```
s = {1, 2, 3, 4}
t= {1, 2, 5, 6}
print(id(s))
s &= t
print(id(s))
```

```
3072579148
3072579148
```

Pour des objets de type frozenset qui sont immutables, c'est différent :

```
s = frozenset({1, 2, 3, 4})
t= frozenset({1, 2, 5, 6})
print(id(s))
s &= t
print(id(s))
```

```
3072052332
3072370252
```

Après l'affectation augmentée `s &= t`, le nom `s` ne réfère plus vers le même objet, un nouvel objet a été complètement créé ; en particulier, l'objet initial `s` est perdu s'il n'est plus référencé par ailleurs.

Opérations entre des ensembles mixtes `set` et `frozenset`

Il est possible d'effectuer des opérations sur des ensembles de types différents, à la fois `set` et `frozenset`. Ces opérations renvoient alors un objet du type du premier opérande :

```
s = frozenset({1, 2, 3, 4})
t= {1, 2, 5, 6}
print(s | t)
print(type(t | s))
```

```
frozenset({1, 2, 3, 4, 5, 6})
<class 'set'>
```

Il en va de même des méthodes :

```
s = frozenset({1, 2, 3, 4})
t= {1, 2, 5, 6}

print(s.union(t))
print(type(t.union(s)))
```

```
frozenset({1, 2, 3, 4, 5, 6})
<class 'set'>
```

Ainsi que des affectations combinées :

```
s = frozenset({1, 2, 3, 4})
t= {1, 2, 5, 6}
s |= t
print(type(s))

s = frozenset({1, 2, 3, 4})
t |= s
print(type(t))
```

```
9 <class 'frozenset'>
10 <class 'set'>
```

Chapitre III

Les dèques

Rappel sur les dèques

Un dèque est une structure de données de type conteneur et qui s'utilise à la fois comme

— une pile (*stack*)
— une file d'attente (*queue*).

Le terme anglais de *deque* est un acronyme pour *double-ended queue*.
Usuellement, une fois un dèque créé, il s'utilise soit comme une pile soit comme une file mais pas les deux simultanément, bien que ce soit possible. Les opérations usuelles sur un dèque s'effectuent sur les extrémités droite et gauche de la structure de données et consistent en

— la lecture des extrémités
— des adjonctions (souvent traduites par `push`)
— des suppressions (souvent traduites par `pop`)

Ces opérations de lecture, d'adjonction ou de suppression aux extrémités sont censées être efficaces, c'est-à-dire qu'elle ont un coût qui est un $O(1)$. En revanche, une lecture, une adjonction ou une suppression ailleurs qu'en une extrémité a un coût important qui est un $O(n)$ où n est le nombre d'éléments du dèque. De toutes façons, en principe, aucune méthode d'un dèque ne permet d'effectuer des opérations sur des éléments différents des extrémités car un dèque n'est pas prévu pour cela.
Quelques algorithmes peuvent utiliser des dèques, par exemple les parcours d'un graphe en profondeur et en largeur.

Les dèques en Python

La bibliothèque standard Python propose la structure de données nommée `deque` (sans accent et au singulier) via le module `collections`. L'implémentation en CPython de ce module est écrite en C et non en Python, ce qui lui assure une bonne efficacité d'exécution.
Voici un exemple simple d'utilisation d'un dèque avec la bibliothèque standard :

```
1 from collections import deque
2
3 t=[17, 87, 14, 18, 19]
```

```
4
5 # création d'un deque
6 d= deque(t)
7 print("création :", d)
8
9 # suppression à droite
10 d.pop()
11 print("suppression à droite :", d)
12
13 # suppression à gauche
14 d.popleft()
15 print("suppression à gauche :", d)
16
17
18 # insertion à droite
19 d.append(42)
20 print("insertion à droite :", d)
21
22
23 # insertion à gauche
24 d.appendleft(10)
25 print("insertion à gauche :", d)
```

```
26 création : deque([17, 87, 14, 18, 19])
27 suppression à droite : deque([17, 87, 14, 18])
28 suppression à gauche : deque([87, 14, 18])
29 insertion à droite : deque([87, 14, 18, 42])
30 insertion à gauche : deque([10, 87, 14, 18, 42])
```

On trouvera un tableau des performances algorithmiques des opérations sur un dèque dans le wiki Python.

Création d'un dèque

Pour utiliser un dèque, il faut importer la structure de données `deque` depuis le module standard `collections`.

Un dèque est créé avec le constructeur `deque`. On peut donner en argument à `deque` n'importe quel itérable :

```
1 from collections import deque
2
3 d= deque(range(10,15))
4 print(d)
5
6 d= deque("orange")
7 print(d)
```

```
8 deque([10, 11, 12, 13, 14])
9 deque(['o', 'r', 'a', 'n', 'g', 'e'])
```

— Ligne 3 : le dèque est initialement formé de la liste des entiers de 10 à 14
— Ligne 6 : le dèque est initialement formé des lettres du mot `orange`.

Il est aussi possible de créer un dèque vide et de lui adjoindre des éléments par la suite, soit par la droite, soit par la gauche.

Taille d'un dèque

On peut connaître à tout moment le nombre d'élément d'un dèque avec la fonction standard `len` :

```
1 from collections import deque
2
3 L=[17, 87, 14, 18, 19]
4 d= deque(L)
5
6 print(len(d))
```

```
7 5
```

Adjonction et insertion

En dehors de la création, les opérations fondamentales d'un dèque sont :

— l'insertion à droite ou à gauche
— la suppression à droite ou à gauche

Les deux méthodes d'insertion sont : `append` et `appendleft`

Les deux méthodes de suppression sont : `pop` et `popleft`

Les méthodes `append` et `appendleft` prennent un argument qu'elles adjoignent à l'extrémité du dèque. Ces méthodes ne renvoient rien :

```
1 from collections import deque
2
3 t=[17, 87, 14, 18, 19]
4 d= deque(t)
5
6 print(d.append(42))
```

```
7 None
```

Les méthodes `pop` et `popleft` ne prennent aucun argument, retirent un élément à une extrémité du dèque et renvoient l'élément retiré :

```
1 from collections import deque
2
3 t=[17, 87, 14, 18, 19]
4 d= deque(t)
5
6 print(d)
7 print(d.pop(), d.popleft())
```

```
8 print(d)
```

```
9 deque([17, 87, 14, 18, 19])
10 19 17
11 deque([87, 14, 18])
```

Tenter de retirer un élément, d'un côté ou de l'autre, d'un dèque **vide** lève une exception de type `IndexError` :

```
1 from collections import deque
2
3 t=[17]
4 d= deque(t)
5 print(d)
6
7 d.pop()
8 d.pop()
```

```
9 Traceback (most recent call last):
10   File "__.py", line 8, in <module>
11     d.pop()
12 IndexError: pop from an empty deque
```

Extension d'un dèque vide

Un procédé fréquent de remplissage d'un dèque est de partir d'un dèque vide et de lui rajouter soit par la droite, soit par la gauche des éléments :

```
1 from collections import deque
2
3 d= deque()
4 print(len(d))
5
6 d.append(42)
7 d.appendleft(10)
8 print(d)
```

```
9 deque([17, 87, 14, 18, 19])
10 0
11 deque([10, 42])
```

— Ligne 3 : création d'un dèque vide.

— Lignes 6-7 : on remplit le dèque, soit par la droite, soit par la gauche.

Création d'un dèque à partir d'un itérable

La création d'un dèque `d` avec le constructeur `deque` à partir d'un itérable `t`, modifiable ou pas, entraîne la création de références vers les éléments de `t`. En particulier, même si un itérable est modifiable, si un deque `d` est construit à partir d'un itérable modifiable tel qu'une liste `L`, toute modification de `d` de type `pop` ou `append` est sans effet sur `L` :

```
from collections import deque

L=[17, 87, 14, 18, 19]
d= deque(L)

print("L :", L)
print("d :", d)

d.append(42)
print("d :", d)
print("L :", L)
```

```
L : [17, 87, 14, 18, 19]
d : deque([17, 87, 14, 18, 19])
d : deque([17, 87, 14, 18, 19, 42])
L : [17, 87, 14, 18, 19]
```

Méthodes de dèque et méthodes de liste

Pour les deux méthodes qui agissent à droite (append et pop), on notera leur analogie avec les méthodes du même nom agissant sur des listes :

```
from collections import deque

t=[17, 87, 14, 18, 19]
d= deque(t)
print(d)

d.pop()
d.append(42)
print(d)

print(t)
t.pop()
t.append(10)
print(t)
```

```
deque([17, 87, 14, 18, 19])
deque([17, 87, 14, 18, 42])
[17, 87, 14, 18, 19]
[17, 87, 14, 18, 10]
```

Méthode `extend` d'un dèque

De la même façon qu'une liste admet une méthode `extend` qui permet d'étendre une liste par un itérable, un dèque possède des méthodes `extend` (extension par la droite) et `extendleft` (extension par la gauche) :

```
1 from collections import deque
2
3 L=[17, 87, 14, 18, 19]
4 d= deque(L)
5 print(d)
6
7 d.extend([10, 100])
8 d.extendleft([-10, -100])
9
10 print(d)
```

```
11 deque([17, 87, 14, 18, 19])
12 deque([-100, -10, 17, 87, 14, 18, 19, 10, 100])
```

Remplacer un dèque par une liste

Le cas d'une pile

En Python, une liste peut remplacer sans dégradation de performance un dèque si le dèque est utilisé comme une pile. Les méthodes d'adjonction et de suppression portent les mêmes noms (pop et append).

La cas d'une file

On pourrait simuler par une liste `L` un dèque utilisé comme *file d'attente*. La suppression de l'élément tout à fait à gauche de `L` se ferait par `L.pop(0)`. Mais les performances seront mauvaises voire catastrophiques. En effet, l'insertion ou la suppression dans une liste par la gauche entraînent la réindexation de tous les éléments suivants.

Chapitre IV

Les files de priorité

Rappels sur les files de priorité

On dispose d'une collection **dynamique** d'objets comparables (tels que des entiers) : des objets peuvent être ajoutés ou retirés à la collection. On cherche à maintenir de manière efficace la connaissance du plus petit élément de la collection d'objets.

Une simple liste ne permet pas de parvenir à cette tâche : si on ajoute des objets à la collection, on peut certes mettre à jour le minimum mais si on retire le minimum il faut ré-examiner tous les éléments un à un pour déterminer le nouveau minimum.

Une **file de priorité** est justement une structure de données capable de maintenir de manière efficace le plus petit élément de la collection d'objets. Quand il est dit « efficacement », il est attendu que la recherche soit en $O(1)$ et l'extraction soit, au pire, en $O(\log n)$ où n est le nombre d'éléments.

Ce type de structure sert par exemple dans

— l'algorithme de Dijkstra de recherche d'un plus court chemin dans un graphe valué

— l'algorithme de Prim de recherche d'un arbre couvrant de poids minimal.

Ainsi, dans l'algorithme de Dijkstra, le gain peut être vraiment substantiel : pour un graphe de quelques milliers de sommets, l'amélioration peut être d'un facteur 10.

La structure de file de priorité permet aussi de trier une liste de nombres de manière asymptotiquement optimale.

Cette structure de données est connue sous le nom anglais *priority queue*. Le terme de *tas binaire* (*binary heap queue*) est réservé à une implémentation particulière de cette structure de données (sous la forme d'un arbre binaire équilibré).

Pour simplifier l'exposé, on va supposer que les éléments de la file de priorité sont des entiers dont on cherche à connaître le minimum alors que la structure évolue. Une file de priorité peut se représenter par un tas binaire : il s'agit d'un arbre binaire dont les sommets sont les entiers de la structure et en sorte que la racine de tout sous-arbre soit inférieure à tous les éléments du sous-arbre. En particulier, la racine de l'arbre est le plus petit de tous les éléments de la structure (l'élément prioritaire). En outre, les feuilles de l'arbre se répartissent sur deux niveaux consécutifs au plus. Ainsi, le nombre de niveaux d'une file de priorité sur n éléments est d'au plus $\log_2(n)$.

Deux opérations sont associées à une file de priorité :

— l'insertion d'un nouvel élément (opération en général désignée par `push`) ;

— l'extraction du minimum via la suppression de la racine de l'arbre (opération en général désignée par pop) ;

Une fois achevée une opération d'insertion ou de suppression du minimum, la structure doit rester un arbre de priorité ce qui nécessite de réorganiser l'arbre. Ces opérations de réorganisation ne sont pas difficiles à comprendre mais elles ne sont pas triviales et demandent du soin à être programmées. Voir par exemple :

— Sedgewick, Wayne : priority queue
— Wikipedia : opérations sur un tas binaire
— la série de trois vidéos Introduction to a Heap.

Il existe d'autres structures de données implémentant les opérations d'une file de priorité. En comparaison, le tas binaire a de bonnes performances. Toutefois, comme le montre un tableau de performances, un tas de Fibonacci a une meilleure complexité pour l'insertion d'un élément qui est en $O(1)$ au lieu de $O(\log n)$ pour une file de priorité.

Les files de priorité en Python

La bibliothèque standard Python propose la structure de données file de priorité via le module heapq. L'implémentation en CPython de ce module est écrite en C et non en Python, ce qui lui assure une bonne efficacité d'exécution.

Voici un exemple simple d'utilisation d'une file de priorité avec le module heapq :

```
1  from heapq import heappush, heappop
2  from random import randrange
3
4  n=6
5  L=[]
6  for _ in range(n):
7      x=randrange(10,100)
8      heappush(L, x)
9      print(x, L)
10
11 print()
12
13 for _ in range(n):
14     print(heappop(L))
```

```
15 68 [68]
16 19 [19, 68]
17 99 [19, 68, 99]
18 70 [19, 68, 99, 70]
19 66 [19, 66, 99, 70, 68]
20 34 [19, 66, 34, 70, 68, 99]
21
22 19
23 34
24 66
25 68
```

```
26 70
27 99
```

— Ligne 7-8 : on génère 6 entiers aléatoires qu'on place successivement dans une liste L.
— Ligne 8 : la fonction `heappush` insère l'élément et réorganise la structure de données pour qu'elle reste une file de priorité.
— Ligne 15-20 : la représentation sous forme de liste de la file de priorité après chaque insertion.
— lignes 13-14 et 22-27 : on retire à chaque étape la racine de l'arbre binaire et on obtient les éléments dans l'ordre croissant.

Dans le code ci-dessus, les éléments sont entrés dans la file un à un à l'aide de la fonction `heappush`. Il est également possible de transformer une liste en une file de priorité grâce à la fonction `heapify` :

```
1 from heapq import heapify, heappop
2 from random import randrange
3
4 n=6
5 L=[randrange(10,100) for _ in range(n)]
6 print(L)
7
8 heapify(L)
9
10 for _ in range(n):
11     print(heappop(L))
```

```
12 [65, 31, 68, 47, 57, 48]
13 31
14 47
15 48
16 57
17 65
18 68
```

L'instruction de la ligne 8 est évidemment essentielle, c'est elle qui organise la liste en file de priorité. Si cette instruction est omise le classement obtenu lignes 13-18 devient incorrect :

```
1 from heapq import heapify, heappop
2 from random import randrange
3
4 n=6
5 L=[randrange(10,100) for _ in range(n)]
6 print(L)
7
8 for _ in range(n):
9     print(heappop(L))
```

```
10 [57, 42, 27, 35, 56, 29]
11 57
12 27
13 29
```

```
35
42
56
```

Une file de priorité Python accepte n'importe quel type d'objets à condition que Python sache les comparer. Ainsi, une file de priorité pourra accepter des listes de listes mais pas des objets parmi lesquels se trouveraient un entier et une chaîne de caractère :

```
from heapq import heappush, heappop

L=[]
heappush(L, 5)
heappush(L, 'z')
print(L)
```

```
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    heappush(L, 'z')
TypeError: unorderable types: str() < int()
```

Exercice : liste des puissances croissantes

On dit qu'un entier est une *puissance* s'il peut s'écrire de la forme x^n où $x > 1$ est un entier et $n \geq 2$ est un exposant entier.

On donne un rang $N > 0$ et on demande d'écrire un code qui donne la N^e puissance P dans l'ordre strictement croissant. Par exemple, si $N = 8$ alors la liste croissante des N premières puissances est :

```
[4, 8, 9, 16, 25, 27, 32, 36]
```

et donc la 8^e puissance vaut $P = 36$.

On placera dans une file de priorité des tuples de la forme `(x**n, (x, n))`. Vérifier que la millionième puissance est $P = 979848556129$.

Concernant la suite des puissances, on pourra consulter l'article de OEIS ou encore Wikipedia: powerful numbers.

Une implémentation en Python d'une file de priorité

Une file de priorité, bien que représentée par un arbre binaire, peut aussi être astucieusement représentée par une liste, cf. les vidéos citées plus haut ou Sedgewick, Wayne.

Si vous vous intéressez à l'implémentation de files de priorité (par exemple, vous voulez en écrire une en C, en C++, en Cython ou encore en Numba + Numpy), le code ci-dessous peut vous aider à comprendre le fonctionnement de push et pop :

```
class heap:
    def __init__(self):
        self.t=[]

    def push(self, x):
```

```
        t=self.t
        t.append(x)
        k=len(t)-1
        while k:
            m=(k-1)//2
            if t[k]<t[m]:
                t[k], t[m]=t[m], t[k]
                k=m
            else:
                break
    def pop(self):
        t=self.t
        if not t:
            raise IndexError("pop from empty heap")
        if len(t)==1:
            return t.pop()
        root=t[0]
        t[0]=t.pop()
        k=0

        while True:
            L=[i for i in [2*k+1, 2*k+2] if i<len(t)]
            if not L:
                break
            mini=min(t[i] for i in L)
            if t[k]<mini:
                break
            for i in L:
                if t[i]==mini:
                    t[i],t[k]=t[k], t[i]
                    k=i
                    break

        return root


from random import randrange


n=6
h=heap()


for _ in range(n):
    x=randrange(10,100)
    h.push(x)
    print(x, h.t)
```

```
print()

for _ in range(n):
    print(h.pop())
```

qui affiche

```
18 [18]
45 [18, 45]
66 [18, 45, 66]
70 [18, 45, 66, 70]
26 [18, 26, 66, 70, 45]
15 [15, 26, 18, 70, 45, 66]

15
18
26
45
66
70
```