Chapitre 10

Dictionnaires

10.1 Notion de dictionnaire

Considérons la liste suivante, supposée contenir des informations sur une personne :

```
informations = ["Jean", "Thomas", 42, 0, 180, 80]
```

Sans informations supplémentaires, nous ne savons pas à quoi correspondent ces informations.
Nom de la personne ? Prénom ? Âge ?

Si l'on souhaite « étiqueter » chacune des valeurs de la liste précédente, il faut recourir à un **dictionnaire**.
Comme les listes, les dictionnaires sont des **conteneurs**, dans lequels on trouve d'autres objets. Mais contrairement aux listes où les éléments sont identifiés par leurs indices, les éléments d'un dictionnaire sont identifiés par une **clé**, qui est généralement une chaîne de caractères.

10.1.1 Définir un dictionnaire

Pour créer un dictionnaire, on commence par créer un dictionnaire vide avec des accolades :

```
>>> dictionnaire = {}
```

On ajoute ensuite des éléments au dictionnaire en précisant la clé et la valeur :

```
>>> dictionnaire["prenom"] = "Jean"
>>> dictionnaire["nom"] = "Thomas"
>>> dictionnaire["age"] = 42
>>> dictionnaire
{'prenom': 'Jean', 'age': 42, 'nom': 'Thomas'}
```

Clé	Valeur
age	42
nom	Thomas
prenom	Jean

☞ *Lorsque Python affiche un dictionnaire, les clés apparaissent généralement dans l'ordre alphabétique, mais ce n'est pas toujours le cas !*

Pour accéder un élément du dictionnaire, on renseigne simplement sa clé :

```
>>> dictionnaire["nom"]
'Thomas'
```

Si la clé n'existe pas dans le dictionnaire, on aura droit à une jolie erreur du type `KeyError`.
Enfin, un dictionnaire ne peut pas contenir 2 clés identiques. Si on modifie la valeur associée à une clé déjà existante, alors cette valeur est remplacée.

```
>>> dictionnaire
{'prenom': 'Jean', 'age': 42, 'nom': 'Thomas'}
>>> dictionnaire["prenom"] = "Alice"
>>> dictionnaire
{'prenom': 'Alice', 'age': 42, 'nom': 'Thomas'}
```

La clé n'est pas obligatoirement une chaîne de caractères, mais ce doit être un objet **non mutable**, comme un entier, un flottant ou un tuple. Pratique par exemple pour enregistrer le nom d'une ville en fonction de ses coordonnées géographiques :

```
villes = {}
villes[(41.9263991, 8.7376029)] = "Ajaccio"
villes[(42.7065505, 9.452542)] = "Bastia"
villes[(42.3052904, 9.1511935)] = "Corte"
```

Enfin, il est possible de définir directement un dictionnaire, de la manière suivante :

```
fruits = {"pommes": 10, "poires": 8, "bananes": 14}
```

10.1.2 Supprimer des éléments

Pour supprimer des éléments d'un dictionnaire, on dispose :

- d'un mot clé : `del`
- d'une méthode : `pop()`

Le mot clé `del` n'est pas spécifique aux dictionnaires, il permet de supprimer une variable.
Comme `return` et `def` , on l'utilise sans parenthèses.

```
>>> a = 2
>>> a
2
>>> del a   # Supprime la variable a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

On l'utilise également pour supprimer une valeur dans un dictionnaire :

```
>>> fruits = {"pommes": 10, "poires": 8, "bananes": 14}
>>> del fruits["pommes"]
>>> fruits
{'poires': 8, 'bananes': 14}
```

La méthode `pop(cle)` supprime la clé précisée, et renvoie la valeur supprimée :

```
>>> fruits = {"pommes": 10, "poires": 8, "bananes": 14}
>>> nombrePoires = fruits.pop("poires")
>>> fruits
{'pommes': 10, 'bananes': 14}
>>> nombrePoires
8
```

10.1.3 Parcourir un dictionnaire

On ne parcourt pas un dictionnaire comme on parcourt une liste. On peut encore utiliser une boucle `for` mais le parcours s'effectue sur les clés et non sur les valeurs :

```
1 fruits = {"pommes": 10, "poires": 8, "bananes": 14}
2
3 for cle in fruits:
4     print(cle)
```

Résultat :

```
pommes
bananes
poires
```

▷ ***Pourquoi le programme n'affiche-t-il pas les clés dans l'ordre où on les a définies ?***

Contrairement à une liste où l'on parcourt les éléments par indice croissant, un dictionnaire n'est pas ordonné. Le parcours des clés ne se fait donc pas toujours dans l'ordre croissant.

▷ ***Comment parcourt-on les valeurs d'un dictionnaire ?***

On peut simplement récupérer la valeur de la manière suivante :

```
1 fruits = {"pommes": 10, "poires": 8, "bananes": 14}
2
3 for cle in fruits:
4     print(fruits[cle])
```

Résultat :

```
10
14
8
```

Attention cependant : les clés étant parcourues aléatoirement, il en est de même des valeurs ! Si vous exécutez ce programme, il se peut que l'ordre d'affichage ne soit pas le même.

Une autre façon de parcourir les valeurs d'un dictionnaire est d'utiliser la méthode `values()` :

```
1 fruits = {"pommes": 10, "poires": 8, "bananes": 14}
2
3 for valeur in fruits.values():
4     print(valeur)
```

Résultat :

```
10
14
8
```

Il existe d'ailleurs une méthode `keys()` qui nous permet de parcourir... les clés du dictionnaire.

```
1 fruits = {"pommes": 10, "poires": 8, "bananes": 14}
2
3 for cle in fruits.keys():
4     print(cle)
```

Résultat :

```
pommes
bananes
poires
```

▷ ***Pourquoi utiliser la méthode `keys()` ? Cela revient au même si on ne l'utilise pas ?***

En effet, mais ceci permet de rendre le code moins ambigu. En lisant le code, on « voit » directement que l'on parcourt les clés et non les valeurs.

Parcourir les clés ET les valeurs

Il existe une troisième méthode, `items()`, qui permet de récupérer simultanément les clés et les valeurs de notre dictionnaire dans un tuple :

```
1  fruits = {"pommes": 10, "poires": 8, "bananes": 14}
2
3  for cle, nombre in fruits.items():
4      print(cle, nombre)
```

Résultat :

```
pommes 10
bananes 14
poires 8
```

Existence d'une clé, d'une valeur

On peut tester l'existence d'une clé ou d'une valeur avec le mot-clé `in` :

```
>>> moyennes = {"maths": 14, "nsi": 18, "si": 13, "physique": 8, "français": 11}
>>> "maths" in moyennes
True
>>> "anglais" in moyennes
False
>>> 14 in moyennes
False
>>> 8 in moyennes.values()
True
```

10.1.4 Fonctions et dictionnaires

Comme les listes, les dictionnaires sont des objets **mutables**, c'est à dire qu'ils ne sont pas copiés lorsqu'ils sont passés en argument à une fonction. Toute modification du dictionnaire dans la fonction sera visible depuis « l'extérieur » de la fonction.

```
1  # Incrémente le nombre de chiens
2  # Ne retourne rien mais modifie le dictionnaire
3
4  def ajouterChien(d):
5      d["chiens"] = d["chiens"] + 1
6
7  dico = {"chiens": 2, "chats": 4, "poulpes": 1}
8  print(dico)
9
10 ajouterChien(dico)
11 print(dico)
```

Résultat :

```
{'chiens': 2, 'chats': 4, 'poulpes': 1}
{'chiens': 3, 'chats': 4, 'poulpes': 1}
```

Comme pour une liste, il est possible de créer une copie d'un dictionnaire avec la fonction `dict()`.

```
dico = {"nom": "Alice", "age": 21}
copieDico = dict(dico) # Crée une copie du dictionnaire
```

10.2 Listes de dictionnaires

10.2.1 Définir une liste de dictionnaires

Considérons le dictionnaire suivant, contenant des informations sur une ville :

```
ville = {"nom": "Ajaccio", "population": 70659, "densite": 861, "code_commune": "2A004"}
```

On peut créer une liste contenant des dictionnaires similaires, composés de clés identiques (ou avec des clés différentes, mais la liste n'aurait pas beaucoup de sens) mais de valeurs différentes :

```
villes = [{"nom": "Ajaccio", "population": 70659, "densite": 861, "code_commune": "2A004"}, {"nom":
 ↪  "Bastia", "population": 45715, "densite": 2359, "code_commune": "2B033"}, {"nom": "Corte",
 ↪  "population": 7446, "densite": 50, "code_commune": "2B096"}]
```

Les éléments de la liste sont séparés par des virgules (comme pour une liste d'entiers). On accède alors aux différentes informations de manière « classique » :

```
>>> villes[0]
{'code_commune': '2A004', 'population': 70659, 'nom': 'Ajaccio', 'densite': 861}
>>> villes[1]["nom"]
'Bastia'
>>> villes[2]["population"]
7446
```

10.2.2 Trier une liste de dictionnaires

Pour trier une liste, on dispose de la fonction `sorted(liste)` qui retourne une copie de `liste` triée dans l'ordre croissant. Cependant, si on tente d'appliquer la fonction `sorted` à notre liste, Python renverra une erreur, car il ne sait pas ordonner deux dictionnaires. Et c'est normal ! Comment peut-il savoir si la ville d'Ajaccio est supérieure ou inférieure à celle de Bastia ? Pour répondre, il faut choisir un critère, il faut comparer des **valeurs correspondantes et ordonnables**.

Supposons que l'on veuille trier les villes par taille de population, dans l'ordre croissant. Il faut l'indiquer à la fonction `sorted` avec l'argument `key` :

```
sorted(villes, key = ...)
```

`key` doit être égal à une fonction qui retourne la valeur que l'on veut ordonner, relativement à un élément de la liste. Pour trier par taille de population, on définit donc une fonction qui retourne la taille de la population pour un élément donné, c'est à dire la valeur associée à la clé `population` :

```
def valeurTri(dictionnaire):
    return dictionnaire["population"]
```

Pour trier la liste par population croissante, on écrit alors :

```
sorted(villes, key = valeurTri)
```

▷ ***Il ne manque pas les parenthèses après `valeurTri` ?***

Non, on passe la fonction `valeurTri` (en tant qu'objet) directement dans la fonction `sorted`. Ainsi, Python ne va pas chercher à ordonner les dictionnaires de la liste (c'est impossible) mais il va ordonner les valeurs de `valeurTri(dictionnaire)` pour tout `dictionnaire` de la liste.

▷ ***C'est compliqué, il faut définir une fonction `valeurTri` juste pour faire le tri...***

En réalité, on peut s'en passer, et utiliser une **fonction lambda**. Les fonctions lambda sont des fonctions définies « à la volée », sur une seule ligne. Ainsi créées, les fonctions ne sont jamais bien compliquées, mais ceci s'avère très pratique quand on veut passer une fonction en argument d'une autre fonction.
Par exemple, notre fonction `valeurTri` peut se définir comme ceci :

```
valeurTri = lambda dictionnaire : dictionnaire["population"]
```

Ou encore, de façon raccourcie :

```
valeurTri = lambda d : d["population"]
```

`valeurTri` est définie comme une simple variable. Mais c'est totalement équivalent à la définition classique avec `def`.

Pour trier notre liste par population croissante, il suffit alors d'écrire :

```
sorted(villes, key = lambda d : d["population"])
```

▷ ***Et si on veut trier par population* décroissante *?***

On utilise un 3ème argument dans la fonction `sorted`, nommé `reverse`, égal à `False` ou `True`.
Ainsi, pour trier dans l'ordre inverse (décroissant), on écrira :

```
sorted(villes, key = lambda d : d["population"], reverse = True)
```

10.3 Exercices

Exercice 01 ***Clé et valeur***

Écrire une fonction `recherche(valeur, dictionnaire)` qui retourne la clé de `valeur` si `valeur` est une valeur présente dans `dictionnaire`, et `None` sinon.

Exercice 02 ***Vainqueur***

Une compétition d'informatique oppose plusieurs participants, auxquels on demande d'écrire le plus rapidement possible un programme répondant à un problème donné. Les résultats sont inscrits dans un dictionnaire sous la forme *nom : temps* où `nom` est le nom du participant et `temps` le temps qu'il a mis pour répondre au problème, en secondes.

Écrire une fonction `vainqueur(resultats)` qui retourne le nom du participant ayant gagné la compétition.

```
>>> resultats = {"Alice": 322, "Bob": 210, "Charlie": 1290}
>>> vainqueur(resultats)
'Bob'
```

Exercice 03 ***Fusion !***

Écrire une fonction `fusion(d1, d2)` qui « fusionne » les dictionnaires `d1` et `d2` en un dictionnaire composé des clés de `d1` et `d2` et retourne ce dernier.

- Si une clé est dans `d1` et `d2`, la valeur associée est la somme des valeurs
- Sinon, c'est la valeur contenue dans `d1` ou `d2`

```
>>> d1 = {"pommes": 4, "poires": 6}
>>> d2 = {"melons": 2, "pommes": 3}
>>> fusion(d1,d2)
{"pommes": 7, "melons": 2, "poires": 6}
```

Exercice 04 ***Utilisateurs***

On souhaite enregistrer des couples *utilisateur/mot de passe* dans un dictionnaire. Chaque clé représente un nom d'utilisateur, et la valeur correspondante est le mot de passe associé. Par exemple :

```
users = {"toto": "azertY", "jo_le_rigolo": "F9dc8_**J", "keke2a": "motdepasse"}
```

Dans cet exemple, l'utilisateur *toto* a pour mot de passe *azertY*.

1. Écrire une fonction `ajouterUtilisateur(utilisateur, motDePasse, dictionnaire)` qui ajoute `utilisateur` au dictionnaire avec le mot de passe `motDePasse` si cet utilisateur ne figure pas déjà dans le dictionnaire.
2. Écrire une fonction `supprimerUtilisateur(utilisateur, dictionnaire)` qui supprime le couple *utilisateur/mot de passe* de `dictionnaire`, si `utilisateur` figure dans `dictionnaire`.
3. Écrire une fonction `login(utilisateur, motDePasse, dictionnaire)` qui renvoie `True` si le couple *utilisateur/mot de passe* est dans le dictionnaire, et `False` sinon.

Exercice 05 ***Inventaire***

Afin de gérer le stock d'une épicerie, on utilise un dictionnaire `produits` dont les éléments sont de la forme *nom : quantité*. Par exemple :

```
stock = {"beurre": 10, "lait": 4, "huile": 5, "AMD Ryzen 7 3700X": 1}
```

1. Écrire une fonction `acheter(produit, quantite, stock)` qui ajoute `quantite` au produit `produit` dans `stock`. Si le produit `produit` n'est pas référencé, on l'ajoute au dictionnaire avec une quantité égale à `quantite`.

```
>>> stock = {"beurre": 10, "lait": 4, "huile": 5, "AMD Ryzen 7 3700X": 1}
>>> acheter("lait", 3, stock)
>>> acheter("sucre", 6, stock)
>>> stock
{'beurre': 10, 'lait': 7, 'huile': 5, 'AMD Ryzen 7 3700X': 1, 'sucre': 6}
```

2. Écrire une fonction `vendre(produit, quantite, stock)` qui retire une quantité `quantite` au produit `produit` dans `stock`. Si le produit n'est pas disponible en quantité suffisante, la quantité n'est pas modifiée et on affiche un message d'erreur. Si toutes les quantités ont été vendues, le produit est supprimé du dictionnaire.

```
>>> stock = {"beurre": 10, "lait": 4, "huile": 5, "AMD Ryzen 7 3700X": 1}
>>> vendre("lait", 3, stock)
>>> vendre("AMD Ryzen 7 3700X", 1, stock)
>>> vendre("beurre", 15, stock)
Impossible : quantités insuffisantes.
>>> stock
{'beurre': 10, 'lait': 1, 'huile': 5}
```

3. Écrire une fonction `disponibilites(panier, stock)` qui retourne `True` si les produits contenus dans `panier` sont disponibles en fonction du `stock`, et `False` sinon.

```
>>> stock = {"beurre": 10, "lait": 4, "huile": 5, "AMD Ryzen 7 3700X": 1}
>>> panier = {"beurre": 5, "huile": 1}
>>> disponibilites(panier, stock)
True
>>> panier = {"lait": 3, "Intel Core i7": 1}
>>> disponibilites(panier, stock)
False
```

Exercice 06 ***Occurences***

Écrire une fonction `occurences(texte)` qui compte le nombre de chaque caractère dans la chaîne de caractères `texte`, et renvoie les résultats sous forme de dictionnaire.

```
>>> occurences("J'adore les dictionnaires !")
{'e': 3, "'": 1, 'a': 2, 'o': 2, 's': 2, 'n': 2, '!': 1, 't': 1, 'l': 1, 'd': 2, 'r': 2, ' ': 3, 'i': 3,
 ↪  'J': 1, 'c': 1}
```