Chapitre 8
Mutabilité et dictionnaires

1 Mutabilité

On a vu au chapitre 5 qu'une fois créée, une liste pouvait être modifiée directement (sans recréer toute la liste), alors que ce n'est pas le cas des chaines de caractères. On dit alors que les listes sont **mutables**, mais que les chaines de caractères ne le sont pas.

```
L = [5, -2, 8]

L[1] = -14 # Modification d'un élément de la liste.

L.pop() # Suppression du dernier élément de la liste.

L.append("Bonjour !") # Ajout d'un élément en fin de liste.

L
```

```
chaine = "belle journée !"

chaine[0] = "B" # Erreur : une chaine n'est pas mutable !

chaine = "Belle journée !" # On doit recréer la chaine entièrement.
```

Dans cette partie, nous allons étudier un peu plus en détails la mutabilité et exposer quelques-unes de ses conséquences surprenantes.

1.1 Identifiants et mutabilité

1.1.1 Identifiant d'une variable

Jusqu'à présent, on s'est intuitivement représenté·e une variable comme une boite dans laquelle on stocke une valeur. Ce n'est pas tout à fait le cas.

L'affectation

```
variable = valeur
```

crée plutôt un pointeur du nom `variable` vers l'adresse d'un emplacement mémoire dans lequel est stockée la valeur `valeur`.

L'adresse dudit emplacement mémoire est appelé l'**identifiant** de la variable `variable` et on y accède à l'aide de la fonction `id` : l'instruction

```
id(variable)
```

renvoie l'adresse de la case mémoire vers laquelle pointe la variable `variable`.

```
a = 3

id(a)

ch = "Bonjour !"

id(ch)
```

1.1.2 Objets non mutables

Lorsque `valeur` est un objet qui n'est pas mutable, la valeur `valeur` se trouve, en général, à un seul emplacement de la mémoire (ce qui permet d'optimiser la mémoire). Si l'on affecte deux variables à cette même valeur, les deux variables pointeront alors vers la même adresse mémoire.

```
x = 4

y = 4

id(x)

id(y) # Logées à la même enseigne.
```

Si l'on affecte une variable à une variable différente, son adresse change : on crée un pointeur vers un nouvel identifiant.

```
x = x + 1

id(x) # On déménage.
```

1.1.3 Objets mutables

Par contre, une variable mutable pourra se retrouver à plusieurs endroits différents de la mémoire. Si l'on crée deux listes, elles auront des adresses différentes, même si elles contiennent initialement les mêmes éléments.

```
L = [1, 2, 3]

M = [1, 2, 3]

id(L)

id(M) # Les variables L et M pointent vers des adresses différentes.
```

La mutabilité des listes s'observe alors de la manière suivante : lorsque l'on modifie la valeur d'un élément d'une liste, on ne change pas l'adresse de la liste, on ne crée pas de nouveau pointeur.
Les opérations de modification d'un élément, d'ajout ou de suppression d'un élément de la liste se font **en place**, sans changer l'identifiant de la liste.

```
L[1] = 14

id(L) # Moi je n'ai pas changé d'adresse.

L.pop()

id(L) # Là non plus.

L.append(5)

id(L) # Toujours pas.

L = [5, 9, -3] # Là, on crée une nouvelle liste.

id(L) # Elle est alors à une adresse différente.
```

1.2 Portée d'un identifiant et effets de bord

1.2.1 Variables locales et globales

On a vu qu'une variable globale non mutable ne peut pas être modifiée à l'intérieur d'une fonction.

```
def fonction(x):
    print(id(x))
    x = x + 1 # Cette modification est locale à la fonction.
    print(id(x))
    return x

a = 42 # Variable globale.

id(a)

fonction(a)

a

id(a)
# Ni la valeur ni l'adresse de la variable globale a
# n'ont été modifiées par la fonction.
```

1.2.2 Effets de bord

En revanche, un objet mutable peut être modifié directement par l'application d'une fonction (c'est le cas par exemple dans toutes les fonctions de tri que nous avons écrites au chapitre 6) ; on parle alors d'**effet de bord**.

```
def fonction(L):
    print(id(L))
    L.append(14)
    print(id(L))
    return L

Liste = [2, -1, 6, 7]

id(Liste)

fonction(Liste)

Liste

id(Liste)
```

Dans certains cas, les effets de bord peuvent être pratiques et rendre les programmes plus élégants. Toutefois, il peut également être dangereux de modifier la liste à chaque application de fonction.

Pour éviter de perdre la liste de départ, on peut commencer par créer une copie de sauvegarde de la liste (voir la section suivante).

1.3 Copies et phénomène d'aliasing

Si une variable `premiere` est définie, alors l'instruction

```
seconde = premiere
```

crée un nouveau pointeur du nom `seconde` vers l'adresse mémoire de la variable `premiere`, c'est-à-dire vers l'identifiant de la variable `premiere`.

1.3.1 Duplication d'un objet non mutable

Dans le cas de variables non mutables, l'instruction précédente permet bien de créer une copie `seconde` de la variable `premiere`.

```
x = 5

y = x

y

id(x)

id(y)
```

Les deux variables pointent initialement vers le même emplacement mémoire, mais elles ne sont pas associées définitivement. En effet, si l'on modifie la valeur de la première variable, son adresse changera mais la seconde variable ne sera pas affectée.

```
x = 3

id(x) # La variable x a changé d'adresse.

y

id(y) # La copie y n'a pas bougé.
```

1.3.2 Aliasing

Observons à présent ce qui se passe dans le cas des listes.

```
L = [1, 2, 3]

M = L

id(L)

id(M)

L[2] = -8
```

```
L

M # La liste M a été modifiée aussi !
```

```
M.append(7)

L # La liste L a été modifiée aussi !
```

Dans cet exemple, les variables `L` et `M` pointent vers la même adresse mémoire, donc vers la même liste. Or cette liste est mutable, donc les modifications de la liste se feront sans changer son adresse mémoire. Ainsi, lorsque l'on modifie la valeur de la variable `L`, on modifie également celle de `M` (puisqu'on modifie le même objet). Et inversement !

Ce phénomène porte le nom d'**aliasing** : les deux noms `L` et `M` sont deux *alias* différents pour la même liste (au même emplacement mémoire).

1.3.3 Copie d'une liste

Le phénomène d'aliasing est particulièrement dangereux lorsqu'il se combine avec les effets de bord : il est facile d'oublier que les modifications d'une liste se répercutent sur tous ses alias. On souhaite donc créer une véritable copie de sauvegarde de la liste, indépendante de la liste de départ.

Une solution consiste à utiliser la technique du **slicing** : pour copier une liste `Liste`, on crée la tranche de la liste `Liste` constituée de tous ses éléments, c'est-à-dire que l'on crée une nouvelle liste (avec un nouvel identifiant), dont les éléments sont ceux de la liste initiale.

```
Copie = Liste[ : ]
```

Testons cette technique sur l'exemple précédent.

```
L = [1, 2, 3]

M = L[ : ] # Copie de la liste L.

M

id(L)

id(M) # L'adresse de M est différente de celle de L.

L[2] = -8

M # Ouf, M n'a pas changé !

M.append(7)

L # La liste L n'a pas changé non plus.
```

Tous nos problèmes d'aliasing semblent alors résolus. Et pourtant...

```
LL = [[3, 5], [-7, 9, 0, 2]]

MM = LL[ : ] # Copie de sauvegarde de LL ?

LL[1][2] = 140

LL

MM # Oh, la liste MM a été modifiée !
```

La commande `LL[ : ]` crée une nouvelle liste, qui contient les mêmes éléments que la liste `LL` : à chaque élément de la copie est associé un pointeur vers l'identifiant de l'élément correspondant de la liste initiale.

Mais dans cet exemple, les éléments de la liste `LL` sont eux-mêmes des listes, donc des objets mutables. Ainsi, lorsque l'on modifie en place l'un de ces éléments de `LL`, on modifie également cet élément dans la copie, puisqu'il pointe vers le même identifiant.

Pour régler le problème, il faut appliquer une fonction de copie à la liste, ainsi qu'à chacun de ses éléments, des éléments de ses éléments, etc. C'est ce que fait — récursivement (voir un prochain chapitre) — la fonction `deepcopy` de la bibliothèque `copy`.
Pour créer une copie profonde d'une liste `Liste`, on procède alors de la manière suivante.

```
import copy

Copie = copy.deepcopy(Liste)
```

Testons cette technique sur l'exemple précédent.

```
import copy

LL = [[3, 5], [-7, 9, 0, 2]]

MM = copy.deepcopy(LL)

MM

LL[1][2] = 140

LL

MM # Ouf, la copie de sauvegarde n'a pas été modifiée !
```

2 Dictionnaires

2.1 Définitions et création d'un dictionnaire

Un **dictionnaire**, de type `dict`, est une collection non ordonnée, donnée entre accolades, de couples

`clé : valeur`

séparés par des virgules. Un dictionnaire associe donc à chaque clé une valeur.

```
Groupe_colle = {'Tabara' : 12, 'Timothé' : 11, 'Romane' : 5}

type(Groupe_colle)
```

Le dictionnaire `Groupe_colle` contient trois entrées. Les clés sont les chaines de caractères `'Tabara'`, `'Timothé'` et `'Romane'`. Les valeurs associées sont respectivement les entiers 12, 11 et 5 (qui représentent le numéro du trinôme de colle de la personne donnée en clé).

2.1.1 Type des entrées

Voici quelques précisions sur les clés d'un dictionnaire.

- ⋆ Les clés d'un même dictionnaire doivent toutes être distinctes les unes des autres : deux couples différents doivent avoir des clés différentes.

  ```
  {"A" : 1, "B" : 2, "A" : 3} # La première entrée ("A" : 1) est écrasée.
  ```

- ⋆ De plus, les clés d'un dictionnaire ne peuvent pas être mutables. En particulier, les clés ne pourront être ni des listes, ni des dictionnaires.

  ```
  {[4, 5] : 2, [[3, 4, -1]] : 3, [] : 0} # Erreur.
  ```

- ⋆ Les clés peuvent être de n'importe quel type non mutable et ne sont pas nécessairement toutes de même type dans un même dictionnaire.

Les valeurs, elles, peuvent être complètement quelconques.

Donnons plusieurs exemples variés de dictionnaires.

```
Min_to_maj = {"a" : "A", "b" : "B", "é" : "É", "alpha" : "A"}

Diviseurs = {13 : [1, 13], 14 : [1, 2, 7, 14], 15 : [1, 3, 5, 15]}

Dico_vide = {}

Kamoulox = {True : "lapalissade", 5 : [2, "Python"], "Python" : False}
```

2.1.2 Ordre des entrées

Contrairement aux éléments d'une liste, les entrées d'un dictionnaire ne sont pas ordonnées.

```
{"A" : 1, "B" : 2} == {"B" : 2, "A" : 1}

[1, 2] == [2, 1]
```

2.1.3 Définition en compréhension

On peut également définir un dictionnaire en compréhension.

```
Longueur = {ch : len(ch) for ch in ["Bien", "le", "bonjour", "", "!"]}

Longueur
```

2.2 Accès aux valeurs

Dans un dictionnaire `Dico`, on accède à la valeur associée à la clé `cle` avec l'instruction suivante.

```
Dico[cle]
```

```
Groupe_colle = {'Tabara' : 12, 'Timothé' : 11, 'Romane' : 5}

Groupe_colle['Romane']
```

2.3 Modification et ajout d'une entrée

Les dictionnaires sont, comme les listes, des objets mutables.

Si `Dico` est un dictionnaire, `cle` une clé et `valeur` une valeur, alors l'instruction

```
Dico[cle] = valeur
```

permet de modifier ou de définir l'entrée du dictionnaire associée à la clé `cle` :

- ⋆ si la clé `cle` existe déjà dans le dictionnaire `Dico`, alors la valeur associée à la clé `cle` est modifiée et devient `valeur` ;
- ⋆ sinon, l'entrée `cle : valeur` est ajoutée au dictionnaire `Dico`.

Modifions les dictionnaires précédents.

```
Groupe_colle = {'Tabara' : 12, 'Timothé' : 11, 'Romane' : 5}

Groupe_colle['Tabara'] = 7 # Modification d'une entrée existante.

Groupe_colle

Groupe_colle['Matti'] = 15 # Création d'une nouvelle entrée.

Groupe_colle

Diviseurs = {13 : [1, 13], 14 : [1, 2, 7, 14], 15 : [1, 3, 5, 15]}

Diviseurs[14].append(-7)
# Ceci est l'ajout d'un élément dans une liste (la valeur associée
# à la clé 14), pas d'une nouvelle entrée dans le dictionnaire.

Diviseurs
```

2.4 Parcours d'un dictionnaire

Pour parcourir un dictionnaire, on procède de la manière suivante.

```
for cle in Dico: # La variable cle parcourt les clés de Dico.
    bloc d'instructions
    dépendant de cle et de Dico[cle]
```

La variable de boucle `cle` prend alors pour valeurs toutes les clés du dictionnaire `Dico`.

```
for prenom in Groupe_colle:
# La variable prenom parcourt les clés du dictionnaire Groupe_colle.
    print(prenom)
    print(Groupe_colle[prenom])
```

2.5 Recherche d'une clé dans un dictionnaire

La liste des clés d'un dictionnaire `Dico` est donnée par :

```
Dico.keys()
```

Testons cette commande sur le dictionnaire `Groupe_colle` précédent.

```
Groupe_colle.keys()

type(Groupe_colle.keys()) # Ce n'est pas tout à fait une liste.

list(Groupe_colle.keys())
```

Pour déterminer si une clé donnée existe déjà dans un dictionnaire `Dico`, on pourrait faire une recherche dans la liste `Dico.keys()` définie plus haut (c'est un bon exercice).

Toutefois, il est plus rapide d'utiliser la commande suivante :

```
cle in Dico
```

qui renvoie `True` si la clé existe dans le dictionnaire `Dico` et `False` sinon.

En effet, les dictionnaires sont implémentés de sorte que cette recherche de clé soit beaucoup plus rapide que la recherche d'un élément dans une liste non triée.

```
'Timothé' in Groupe_colle

'Élisa' in Groupe_colle
```

Nous verrons en TP de jolies applications des dictionnaires.