ITC – chapitre n°10
Tuples, dictionnaires

Nous voyons dans ce chapitre de nouvelles structures de données composées : après les listes et les chaînes de caractères, voici les tuples et les dictionnaires. Nous aurons ainsi fait le tour des structures de données natives de Python.

I. Tuples

1.1. Décalaration, opérations élémentaires

Un tuple est un **ensemble immuable de variables**, c'est-à-dire que l'on ne peut pas le changer. On les définit (une fois pour toutes) par la syntaxe suivante :

```
In [1]: x = (12, -5, True)  # création d'un tuple - noter les parenthèses
```

> Attention à ne pas confondre avec la création de liste : `x = [12, -5, True]`

On définit également **l'opérateur concaténation** `+` et **l'opérateur répétition** `*`, permettant de créer de nouveaux tuples à partir des précédents :

```
In [2]: y = (34, 2)
In [3]: z = x + y  # z référence le tuple (12, -5, True, 34, 2)
In [4]: t = y*3    # t référence le tuple (34, 2, 34, 2, 34, 2)
```

On peut enfin accéder à la taille et aux éléments du tuple avec des syntaxes similaires à celles des listes, et tester la présence d'un élément avec `in` :

```
In [5]: len(z)        # renvoie la taille de z, ici 5
In [6]: a = z[0] + 2  # z[0] vaut 12 donc a vaut 14
In [7]: z[2]
Out[1]: True
In [8]: u = z[2:4]   # u est le tuple (True, 34)
In [9]: -5 in z  # calcul en O(len(z))
Out[2]: True
```

> Comme toujours, les indices démarrent à 0 et finissent à $n-1$, n étant le nombre d'éléments dans le tuple.

Rappelons à propos des tranches qu'un nouveau tuple est créé, correspondant à la tranche.

1.2. Parcours d'un tuple

On peut parcourir chaque élément d'un tuple avec une boucle `for` :

```
collection = (2, 5, 4, 3)
for elt in collection :
    print(elt)
```

affiche tous les éléments du tuple ; on peut également obtenir le même résultat en parcourant les indices du tuple :

```
n = len(collection)
for i in range(n) :
    print(collection[i])
```

1.3. Gestion en mémoire

Les tuples sont **immuables**, comme les chaînes de caractères mais contrairement aux listes : cela signifie qu'on ne peut pas modifier un élément d'un tuple déjà créé. Autrement dit, l'opérateur `[]` peut être utilisé pour lire un élément (ou une tranche) d'un tuple, mais pas pour assigner des valeurs :

```
In [1]: x = (12, -5, True)  # création d'un tuple
In [2]: x[0] = 3            # interdit : modification d'un tuple
TypeError: 'tuple' object does not support item assignment
In [3]: x = (3, -5, True)   # autorisé : on créé un nouveau tuple
```

Signalons enfin que les affectations multiples, que nous avons déjà croisées, utilisent des tuples : en effet, le membre de droite est d'abord évalué comme un tuple, puis affecté au membre de gauche, lu non comme un tuple mais comme un ensemble de variables (on dit que le tuple est dépaquetté) :

```
In[10]: a, b, c = 3, x[0] + 2, 3.1415
```

S'il n'y a pas le même nombre d'éléments de chaque côté, il y aura erreur :

```
In [1]: a, b = 2, 1, 6
ValueError: too many values to unpack (expected 3)
In [2]: a, b, c = 4, 5
ValueError: not enough values to unpack (expected 3, got 2)
```

> Le PEP8 recommande de ne pas mettre d'espace juste avant la virgule, mais d'en mettre après, sauf si on ajoute une parenthèse : `(3, 6, 9)` ou `(3, 6,)` sont donc corrects. Par ailleurs, on ne met pas d'espace autour des deux-points dans les tranches, sauf s'il y a un autre opérateur : par exemple `x[i:j]` mais `x[i+j : k]` ; l'idée est de considérer les deux-points comme un opérateur de basse priorité.

De même dans les fonctions, les `return` contenant plusieurs valeurs sont en fait des tuples :

```
def fonction_nulle(x):
    return x, x**2, x**3

# est équivalente à
def fonction_nulle(x):
    t = (x, x**2, x**3)
    return t
```

Une conséquence à garder en tête du caractère immuable des tuples est le comportement très différent des listes lorsqu'ils sont passés en paramètres à une fonction. En effet, comme ils sont immuables, on ne peut pas changer le contenu d'un tuple depuis l'intérieur d'une fonction. Par exemple :

```
def une_fonction(t: tuple):
# ici t est une variable locale pointant sur un tuple existant en-dehors
    x = t[0]
    t = (x**2) + t[1:]  # on créé un nouveau tuple et on assigne t
    return t

tuptup = (8, 4, 2, 5)
tup2 = une_fonction(tuptup)    # tuptup n'a pas été modifié
```

Notons que le code précédent pourrait fonctionner de la même manière avec une liste : la ligne 4 créé une **nouvelle** variable et y attache l'étiquette locale. En revanche l'exemple suivant :

```
def une_procédure(l: list):
# ici l est une variable locale pointant sur une liste existant en-dehors
    l[0] = l[0]**2  # on change le contenu de la liste

lili = [8, 4, 2, 5]
une_procédure(lili)    # lili a été modifiée
```

ne pourrait pas fonctionner pour un tuple (immuable) mais est acceptable pour une liste.

1.4. Complexités

Les complexités sur les tuples sont assez intuitives :

- les opérations du type `len` ou lire un élément `t[i]`, sont en $\mathcal{O}(1)$;
- créer un tuple de taille n est en $\mathcal{O}(n)$;
- l'opérateur `in` doit parcourir tout le tuple pour tester la présence d'un élément, il est donc en $\mathcal{O}(n)$;
- extraire une tranche de longueur n' est en $\mathcal{O}(n')$.

II. Dictionnaires

2.1. Définition

Un dictionnaire est un **ensemble mutable d'associations clé-valeur**. La différence avec les listes est importante : au lieu d'avoir un ensemble ordonné d'éléments accessibles par leur **indice** (nécessairement entier et compris entre 0 et $n-1$), l'ensemble des éléments n'est pas ordonné, mais rangé à partir d'une clé qui peut être de n'importe quel type **non mutable**. On les définit avec l'opérateur `{ }`, par exemple ci-dessous on utilise comme clé le nom des speedrunners du jeu Celeste (de type `str`) et comme valeur leur temps en millisecondes (de type `int`) :

```
In [1]: records = {"secureaccount": 1592594, "Isaactayy": 1559070, \
                   "Zkad": 1574523, "Marlin": 1583312}
```

> Comme pour les listes, on peut théoriquement utiliser des types différents pour chaque clé et chaque valeur d'un dictionnaire ; par exemple `{"a": 3, 42: True, -2.8: "oui", False: False}` est un dictionnaire acceptable pour la machine. Cependant, comme pour les listes, ceci est une mauvaise pratique : on souhaite que toutes les paires clé-valeur d'un dictionnaire aient le même comportement !

On peut également créer les dictionnaires avec un syntaxe en compréhension, comme les listes et les tuples : à partir de calculs ou de conteneurs pré-existants :

```
In [1]: d = {i: i**2-1 for i in range(n)}
In [2]: d2 = {noms[i]: vals[i] for i in range(n)}
```

2.2. Manipulations élémentaires

Une fois le dictionnaire défini, on peut retrouver et modifier une valeur en indiquant la clé dans l'opérateur `[ ]`, et obtenir le nombre d'éléments avec `len` :

```
In [2]: records['secureaccount']
Out[1]: 1592594
In [3]: len(records)
Out[2]: 4
In [4]: records['secureaccount'] = 1571888  # il a fait un nouveau record
In [5]: records['secureaccount']
Out[3]: 1571888
```

Si on souhaite ajouter une nouvelle clé, on peut simplement utiliser l'opérateur d'affectation (avec la nouvelle clé à gauche) ; attention cependant, on ne peut pas **lire** une clé qui n'a pas encore été définie. Par exemple, en repartant du dictionnaire initial,

```
In [1]: records = {"secureaccount": 1592594, "Isaactayy": 1559070, \
                   "Zkad": 1574523, "Marlin": 1583312}
In [2]: a = records['koralreef']  # 'koralreef' n'est pas une clé définie
KeyError: 'koralreef'
In [3]: records['koralreef'] = 1585675  # on définit la clé 'koralreef'
In [4]: len(records), records['koralreef']
Out[1]: 5, 1585675
```

> ⚠ Encore une fois, on voit que l'opérateur d'affectation n'est pas symétrique !

On peut tester l'existence d'une clé par l'opérateur `in` :

```
In [5]: "secureaccount" in records
Out[2]: True
In [6]: "Licari-Guillaume" in records
Out[7]: False  # sniff :(
```

> ⚠ L'opérateur `in` est **très différent** pour les dictionnaires et pour les listes/tuples/chaînes de caractère ! Pour les listes/tuples/chaînes de caractères, `v in conteneur` teste si la **valeur** v est présente, ce qui suppose de parcourir tout le conteneur pour tester ; tandis que pour les dictionnaires, on teste l'existence de la **clé**.

On peut supprimer une clé avec `del` :

```
In [7]: del records['Marlin']
In [8]: 'Marlin' in records
Out[8]: False
```

2.3. Parcours exhaustif d'un dictionnaire

On peut itérer sur un dictionnaire avec une boucle `for` :

- en itérant directement sur les paires clé-valeur à l'aide de la méthode `items` : la boucle `for` parcourt alors un ensemble de tuples clé-valeur par exemple, pour enlever 5 millisecondes à tous les scores suite à la découverte d'un bug dans le chronomètre, ,

```
for clé, val in records.items():
    records[clé] = val - 5
```

ou encore, si on ne veut pas dépaquetter le tuple (moins lisible, donc à ne pas faire, mais fonctionnel),

```
for paire in records.items():
    # paire est donc un tuple (clé, valeur)
    records[paire[0]] = paire[1] - 5
```

- en itérant sur les clés seulement à l'aide de la méthode `keys` :

```
for clé in records.keys():
    records[clé] -= 5
```

Selon le besoin de lire simplement les valeurs ou non, on pourra utiliser l'une ou l'autre méthode.

> ℹ On peut simplement écrire `for clé in records` : cela fonctionne et se comporte comme `for clé in records.keys()`. Ceci est déconseillé, car écrire explicitement l'appel à `keys` permet de voir explicitement que l'on itère sur les clés uniquement.

2.4. Gestion en mémoire

Comme pour les listes, les dictionnaires sont **mutables**. Les difficultés associées dans le cas des listes se retrouvent donc ici.

En premier lieu, il faut faire attention à la copie : l'opérateur d'affectation donnera une étiquette qui pointe sur le même dictionnaire. On peut également utiliser la fonction `copy` :

```
In [1]: records = {"secureaccount": 1592594, "Isaactayy": 1559070, \
                   "Zkad": 1574523, "Marlin": 1583312}
In [2]: records_2 = records  # records_2 est une étiquette sur le même dict
In [3]: records_2 is records
Out[1]: True
In [4]: records_3 = records.copy()  # records_3 est un autre dict
In [5]: records_3 is records
Out[2]: False
In [6]: records["Zkad"] = 1573214  # mise à jour
In [7]: records["Zkad"], records_2["Zkad"], records_3["Zkad"]  # lecture
Out[2]: 1573214, 1573214, 1574523
```

On peut effectuer une copie « à la main » avec une boucle explicite ou la syntaxe en compréhension :

```
copie_1 = {}
for nom, score in records.items():
    copie_1[nom] = score

copie_2 = {nom: score for (nom, score) in records.items()}
```

De même, lorsqu'on passe un dictionnaire en paramètre à une fonction, l'étiquette locale à la fonction pointe sur le même dictionnaire que celui existant dans l'espace global : on peut donc modifier le dictionnaire depuis la fonction (c'est un potentiel effet de bord) :

```
def fonction_nulle(dico_à_sacrifier):
    for k in dico_à_sacrifier.keys():
        dico_à_sacrifier[k] = 0

records = {"secureaccount": 1592594, "Isaactayy": 1559070, \
           "Zkad": 1574523, "Marlin": 1583312}
fonction_nulle(records)
# à partir de là, les valeurs sont 0 pour toutes les clés
```

2.5. Complexités

Le fonctionnement détaillé des dictionnaires est à votre programme de 2e année ; nous allons ici en donner une vision simplifiée pour comprendre d'où viennent les complexités des opérations élémentaires.

Un dictionnaire peut être vu comme un tableau associé à une fonction qui transforme la clé en indice valide : la clé `'Zkad'` est transformée en un indice entier (mettons 3827) et la case correspondante du tableau est remplie avec la valeur correspondante (ici 1574523). La plupart des cases sont vides.

On comprend ainsi les complexités suivantes :

- lire / écrire dans une case associée à une clé : `dico[clé] = truc` ou `un_calcul = dico[clé] + machin` ; il faut simplement calculer le numéro de case associé et regarder, donc $\mathcal{O}(1)$;
- ajouter ou supprimer une clé : `dico[nvelle_clé] = val` ou `del dico[clé]` : il faut calculer le numéro de case associée et la remplir/la vider, donc $\mathcal{O}(1)$;
- tester l'existence d'une clé : `clé in dico` ; il faut simplement calculer le numéro de case associé et regarder si elle est remplie ou vide, donc $\mathcal{O}(1)$.

On voit encore une fois la différence majeure du comportement de `in` entre les dictionnaires d'une part et les autres conteneurs d'autre part !

Cette explication succinte explique aussi pourquoi on ne peut pas utiliser de variables mutables (listes/dictionnaires) comme clé : si la variable change, le résultat de la fonction qui calcule l'indice de la case correspondante peut changer aussi, ce qui brise la cohérence de la structure de donnée.

III. Exercices

1 – La ferme de Gaston contient 5 lapins, 7 vaches, 2 cochons et 4 chevaux.

1.1 – Définir un dictionnaire `ferme_gaston` dont les clés sont les noms des animaux (de type `str`, par exemple `"lapin"`) et les valeurs leur nombre (entier).

1.2 – Écrire une procédure qui affiche la liste des animaux de la ferme.

1.3 – Écrire une fonction qui calcule le nombre total d'animaux dans la ferme.

1.4 – Écrire une fonction `achat(ferme, espèce, nb)` qui modifie le dictionnaire `ferme` pour représenter l'achat de n animaux de l'espèce proposée ; par exemple, `achat(ferme, 'poules', 12` mente de 12 le nombre de poules.

2 – On travaille sur un dictionnaire `objets` contenant des objets et leur prix entre 1 et 100€.

2.1 – Écrire les instructions pour créer un tel dictionnaire (mettre 6 à 12 objets, selon votre inspiration).

2.2 – Écrire une fonction `fourchette(objets)` pour obtenir les prix le plus bas et le plus haut.

2.3 – Écrire une fonction `prix_moyen(objets)` pour calculer le prix moyen de la collection.

2.4 – Pour les soldes, tous les objets contenant un « a » dans leur nom sont à moitié prix. Écrire une procédure modifiant de dictionnaire selon ce principe.

3 – Écrire une fonction `maxi_dict(dico)` qui renvoie la paire clé-valeur pour laquelle la valeur est la plus élevée du dictionnaire.

4 – Écrire une fonction `fréquences_lettres(texte)` qui prend le nom d'un fichier de texte en paramètre et calcule la fréquence d'apparition de chaque caractère présent dans le texte, puis les affiche. On fournit un fichier « roman.txt » pour calculer les fréquences des caractères, et vérifier que cela est cohérent avec vos connaissances de la langue française.

5 – (Difficile) On fournit un fichier `decimales_pi.txt` contenant environ 130 000 décimales de π. On cherche, pour un ensemble de séquences de longueur donnée (par exemple 0; 2; 4 ou encore 34; 22; 98; 23) à déterminer les fréquences d'apparition de ces séquences dans les décimales de π. Pour le premier exemple, on s'attend à avoir environ 1/3 pour chacun des éléments. On pourra afficher l'ensemble en graphe de barres à l'aide de Pyplot : par exemple si les fréquences sont placées dans un dictionnaire `freqs`, on pourra utiliser :

```
n = len(freqs)
plt.bar(range(n), [v for k, v in freqs.items()])
plt.xticks(range(n), [k for k in freqs.keys()])
```

Écrire un ensemble de fonctions / procédures qui permette de faire cela pour de des séquences au choix. On pourra ensuite tester avec l'ensemble des séquences d'un chiffre `[k for k in range(10)]`, ou à deux chiffres `[k for k in range(100)]`.