B. Types construits

15. P-uplets

Collecter des données en Python

Comme tous les langages de programmation, Python possède des types de données permettant de stocker plusieurs informations sous forme indexée (avec un accès à chaque information individuellement).

Ces types de données sont très utiles pour gérer des fichiers CSV (comma separated values), des pixels d'une image, des données Exif d'une photo…

Python admet quatre types construits de collections de données :

- **Tuple** (P-uplet) : collection ordonnée, non modifiable autorisant les données dupliquées.
- **List** (tableau indexé) : collection ordonnée, indexée et modifiable autorisant les données dupliquées.
- **Dictionnaires** : collections désordonnées, modifiables et indexées n'autorisant pas les données dupliquées.
- **Set** : collection désordonnée, non indexée et n'autorisant pas les données dupliquées. On ne peut pas modifier ses données, mais on peut en ajouter.

Le contenu de ces collections peut être de n'importe quel type simple : Booléen, Entier, Chaîne de caractère, Réel….

Notez que quand un type est modifiable, on parle aussi de « mutable ».

Programmer avec les p-uplets (tuple)

Commençons par évoquer les Tuple (P-Uplet en français). Il s'agit de listes dont le contenu est défini à la création et qui ne sont ensuite plus modifiables. On y placer n'importe quel type de valeurs.

Pour définir un Tuple, on lui donne un nom, comme à une variable ou constante (il s'agit ici d'une constance puisqu'il n'est pas modifiable) et on définit le contenu de la liste entre parenthèses, en séparant chaque élément par une virgule :

Exemple de création de Tuple

```
tup1 = (10,25,50,85,100)
tup2 = ("singe","chat","chien","oiseau")
tup3 = (1.5,0.1,2.8,3.14)
tup4 = ("chien",15,3.8,"jeudi")
```

On accède ensuite au contenu du Tuple par **l'indice de l'élément, placé entre crochets après le nom du Tuple**. Les indices commencent à la valeur zéro qui correspond au premier élément de la liste :

Exemple d'utilisation de Tuple

```
tup1 = (10,25,50,85,100)
tup2 = ("singe","chat","chien","oiseau")
tup3 = (1.5,0.1,2.8,3.14)
tup4 = ("chien",15,3.8,"jeudi")
```

```
print(tup1[0]," ",tup3[2]," ",tup3[1]," ",tup4[3])
```

Résultat

```
10   2.8   0.1   jeudi
```

Cette notation entre crochets sera également utilisée dans les autres types indexés.

16. Tableaux indexés

Les tableaux en Python

Il est beaucoup plus intéressant de pouvoir modifier les éléments d'une liste après la création, que ce soit pour trier cette liste, pour y ajouter des éléments, effectuer des calculs sur les éléments…

Pour cela, Python propose le type « list », également appelé « tableau indexé ». Une liste ressemble à un Tuple, à la différence que l'on peut modifier ses éléments après la création et que la création se fait avec des crochets à la place des parenthèses :

Exemple de création et de modification de tableau indexé

```
tab1 = [10,25,50,85,100]
tab2 = ["singe","chat","chien","oiseau"]
tab3 = ["chien",15,3.8,"jeudi"]
tab1.append("oiseau")

print(tab1)
```

Résultat

```
[10, 25, 50, 85, 100, 'oiseau']
```

Comme le tableau indexé est modifiable, il possède une méthode appelée « **append** » qui permet de rajouter un élément en fin de liste. On remarque dans l'exemple précédent que l'élément « oiseau » a été rajouté au premier tableau indexé.

Par la suite, même si on peut mélanger les types dans un tableau indexé, nous allons travailler sur des tableaux contenant des valeurs de même type.

Gérer les index

Comme nous l'avons vu plus haut, il est possible d'accéder à chaque élément d'un tableau indexé par la position de celui-ci dans le tableau (liste). **Le premier élément de la liste porte le numéro zéro** !!

Les tableaux indexés étant modifiables, il est possible d'y écrire des valeurs ou d'y lire des valeurs à l'aide du numéro d'index de la case du tableau.

Index des cases d'un tableau de 6 cases :					
0	1	2	3	4	5

Exemple : à partir du tableau indexé suivant

```
tableau1 = [10,20,30,40,50,60,70]
```

L'instruction « `print(tableau1[1])` » affichera la valeur « 20 », correspondant à la seconde case du tableau qui est numéroté « 1 » (puisque la première case est numérotée « 0 »).

Pour modifier une valeur du tableau, on pourra écrire l'instruction : « `tableau1[0] = 5` » qui correspond donc à une nouvelle affectation du contenu de cette case du tableau.

L'instruction « `print(tableau1)` » reverra maintenant « `[5, 20, 30, 40, 50, 60, 70]` » puisque la première case du tableau a été modifiée.

- ***Tableaux indexés et chaîne de caractère***

Notez qu'une chaîne de caractère, de type string (str), possède des propriétés similaires aux tableaux indexés. Il est ainsi possible d'accéder aux différents caractères de la chaîne par leur position :

Exemple d'accès à un caractère d'une chaîne string

```
texte = "lycée"
print(texte[2])
```

Le code ci-dessus reverra « c » qui est le caractère possédant l'index « 2 » dans la chaîne « lycée » :

Caractère	l	y	c	é	e
Index	0	1	2	3	4
Index négatif	-5	-4	-3	-2	-1

Notez qu'il est possible d'accéder aux éléments d'un tableau indexé en commençant par la fin et en utilisant des indexes négatifs. L'élément [-1] étant le dernier élément du tableau, [-2] l'avant-dernier…

- ***Quelques fonctions supplémentaires sur les tableaux indexés (listes)***

Nous avons déjà vu la commande « append » qui permet d'ajouter un élément en fin de liste. La commande « **pop** » **efface un élément** quand on donne l'index, ou le dernier élément de la liste si on ne donne pas d'index :

```
tableau1 = [10,20,30,40,50,60,70]

tableau1.pop()
```

supprime la dernière valeur (« 70 ») du tableau qui fera maintenant une case de moins.

Il est possible de **vérifier si un élément est présent** dans le tableau avec la commande « element in tableau », par exemple « `20 in tableau1` » renverra « true ».

On peut **trier un tableau** par ordre croissant de ses valeurs avec la commande « sort » de la façon suivante : « `tableau1.sort()` ».

On peut aussi **inverser l'ordre d'un tableau** (même non trié) avec la commande « reverse » : « `tableau1.reverse()` » qui donnera « `[70, 60, 50, 40, 30, 20, 10]` »

Enfin, il est possible **d'effacer un élément d'un tableau** avec la commande « remove » en utilisant sa valeur, ou « del » en utilisant son index :

Effacer des éléments de tableau

```
tableau1 = [10,20,30,40,50,60,70]
tableau1.remove(30)
print(tableau1)

del tableau1[0]
print(tableau1)
```

résultat

```
[10, 20, 40, 50, 60, 70]
[20, 40, 50, 60, 70]
```

Construire un tableau par compréhension

Pour créer le contenu d'un tableau indexé, on peut le lire depuis un fichier (par exemple CSV), ce que nous verrons dans le chapitre suivant, ou créer une boucle pour ajouter des éléments au tableau de façon calculé.

Python propose une fonction dédiée à cette méthode de création de tableaux indexés (liste) appelée « compréhension ». Cette méthode intégrée à Python est généralement d'une exécution assez rapide.

La syntaxe en est la suivante :

```
[fonction(objet) for objet in liste if condition(objet)]
```

La « fonction » est une fonction mathématique sur les objets du tableau indexé.

« liste » peut être un tableau indexé ou un « range » par exemple.

Enfin la condition n'est pas nécessaire, mais peut être utilisée au besoin.

Voici quelques exemples de création de tableaux indexés par compréhension et le contenu de chaque tableau ainsi créé :

Création de tableaux par compréhension :

```
nombres = [x for x in range(10,21)]
```
résultat : `[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]`

```
paires_nombres = [x for x in nombres if x%2==0]
```
résultat : `[10, 12, 14, 16, 18, 20]`

```
decal3 = [x+3 for x in paires_nombres]
```
résultat : `[13, 15, 17, 19, 21, 23]`

```
tableau2=[[chr(x)+chr(y) for x in range(97,100)]for y in range(97,100)]
```
résultat : `[['aa', 'ba', 'ca'], ['ab', 'bb', 'cb'], ['ac', 'bc', 'cc']]`

Dans le dernier exemple, on a construit un tableau à deux dimensions, en utilisant la fonction « `chr(valeur)` » qui renvoi un caractère en fonction de sa position dans la table des caractères (voir le chapitre précédent).

Tableaux à deux dimensions : matrices

Comme nous venons de le voir dans le dernier exemple, il est possible de créer des tableaux indexés à plusieurs dimensions. C'est particulièrement pratique pour représenter des tableaux de valeurs comme dans un tableur, ou des matrices, à partir de lignes et de colonnes, mais on est pas limité à deux dimensions et on pourra également utiliser trois dimensions pour des positions dans l'espace, quatre dimensions pour des vecteurs dans l'espace…

Pour **créer** un tableau à deux dimensions, on utilisera deux crochets imbriqués pour les lignes successives (les éléments de chaque ligne constituant les colonnes) : [[ligne1col1, ligne1col2, ligne1col3],[ligne2col1, ligne2col2, ligne2col3],…].

Exemple

```
tableau1=[["a1", "a2", "a3"], ["b1", "b2", "b3"], ["c1", "c2", "c3"]]
```

Qui représenterait le tableau suivant :

Ligne ⬇	Colonne➔	0	1	2
	0	a1	a2	a3
	1	b1	b2	b3
	2	c1	c2	c3

Notez que le tableau précédent aurait pu être créé plus simplement par compréhension :

```
Tableau1 = [[chr(y)+str(x-96) for x in range(97,100)]for y in range(97,100)]
```

On peut ensuite **accéder** à chaque élément du tableau par ses indices de ligne et de colonne entre deux crochets successifs [ligne][colonne] :

« `print(tableau1[0][1])` » renverra « a2 »

« `print(tableau1[2][0])` » renverra « c1 »

Naviguer dans un tableau par itération

À partir de là il est bien sûr possible de parcourir toutes les cases du tableau avec une boucle (for ou while). Pour savoir où arrêter la boucle, on peut utiliser la fonction « `len(nom_du_tableau)` » pour en connaître la longueur.

Exemple de parcours de tableau avec une boucle for en utilisant l'index

```
tableau1 = [10,20,30,40,50,60,70]
for i in range(len(tableau1)):
    print(tableau1[i])
```

L'exemple ci-dessus renverra tous les éléments du tableau à raison d'une valeur par ligne (car chaque « print » commence sur une nouvelle ligne ici).

Attention, **len** renverra le nombre de cases du tableau, pas l'index de la dernière case : il faudra donc parfois arrêter une boucle (comme while) à « `len(nom_du_tableau) - 1` ».

Il est également possible de parcourir l'ensemble d'un tableau sans utiliser l'index en procédant avec une boucle for écrite de la façon suivante :

Itération sur les éléments d'un tableau

```
for valeur in tableau1:
    print(valeur)
```

Dans ce cas on n'aura pas d'indication sur la position de la valeur dans le tableau, mais directement la valeur de chaque case du début à la fin du tableau.

Si le tableau est à deux dimensions, on utilisera deux boucles imbriquées pour parcourir les lignes et les colonnes.

Itération dans un tableau à deux dimensions

```
tableau2=[[chr(y)+str(x-96) for x in range(97,100)]for y in range(97,100)]
resultat = ""
for x in range(len(tableau2)): # Parcours par ligne
    for y in range(len(tableau2[0])): # Parcours par colonne
        resultat+=tableau2[x][y]+" "
print(resultat)
```

Résultat du programme

```
a1 a2 a3 b1 b2 b3 c1 c2 c3
```

On remarque que « `len(tableau2)` » donne le nombre de lignes alors que « `len(tableau2[0])` » donnera le nombre de colonnes du tableau.

17. Dictionnaires

Gérer des ensembles de données de natures différentes

Nous avons vu que les tableaux indexés permettaient de retrouver un élément à partir de sa position dans le tableau. Si nous voulons ranger des éléments et les retrouver à partir d'une « clé » d'identification, nous pouvons utiliser un autre type de données proposé par Python : le dictionnaire.

Un dictionnaire (type « `dict()` ») contient des éléments qui sont des paires « `clé : valeur` » séparées par des virgules. Il n'y a pas de limite au nombre de couples clé/valeur dans le dictionnaire, mais il y a des restrictions sur ces deux éléments :

- **Clé** : chaque clé doit être **unique** et de type chaîne de caractères (str), nombre (int ou float) ou tuple.
- **Valeur** : elle peut être de n'importe quel type

Voici un exemple de dictionnaire :

```
agent007 = {"nom":"Bond", "prenom":"James", "Taille":1.85}
```

Il est alors possible d'accéder aux éléments de ce dictionnaire par leur clé, par exemple : « `print(agent007["nom"])` » va afficher « `Bond` ».

En revanche, il ne sera pas possible d'accéder aux éléments par un numéro d'index comme dans un tableau indexé. Si on affiche le contenu du dictionnaire, l'ordre d'affichage peut être différent de l'ordre de création puisqu'il n'y a pas d'indexation.

Il est possible d'imbriquer les dictionnaires pour en faire des dictionnaires de dictionnaires de la façon suivante :

Dictionnaire de dictionnaire

```
agent007 = {"nom":"Bond", "prenom":"James", "Taille":1.85}
agent117 = {"nom":"Bonisseur de La Bath", "prenom":"Hubert",
"Taille":1.81}
agents = {7:agent007,117:agent117}
print(agents[117])
```

Résultat

```
{'nom': 'Bonisseur de La Bath', 'prenom': 'Hubert', 'Taille': 1.81}
```

Construire un dictionnaire

Nous avons déjà vu qu'il est possible de créer un dictionnaire en le remplissant directement avec des couples de valeurs, comme dans l'exemple ci-dessus.

Il est également possible de rajouter ou de modifier des valeurs dans un dictionnaire à partir de leurs clés :

Ajout de couples de valeurs à un dictionnaire

```
agent007 = {}
agent007["nom"]="Bond"
agent007["prenom"]="James"
agent007["cocktail"]="Vodka Martini"
print(agent007)
```

Résultat

```
{'nom': 'Bond', 'prenom': 'James', 'cocktail': 'Vodka
Martini'}
```

On aurait pu ici remplacer la première ligne par la définition du type dictionnaire : « `agent007=dict()` ».

Il est également possible de rajouter des éléments dans un dictionnaire en utilisant la méthode « update() » de la façon suivante :

```
agent007.update({"arme" : "walter PPK"}
```

D'autres méthodes sont utilisables, comme « clear() » qui efface toutes les valeurs du dictionnaire ou « `del(agent007["arme"])` » qui permet d'effacer un couple clé/valeur.

Notez enfin qu'il est possible de transformer une liste contenant des couples d'éléments en dictionnaire avec l'écriture « `dico=dict(liste)` ». Cela peut être utile pour des exercices de cryptographie simple (code césar par exemple), mais il faut veiller à ce que les clés soient bien **uniques** !

Retrouver des données dans un dictionnaire

Nous avons déjà vu qu'il était possible d'accéder à une valeur à partir de sa clé avec la syntaxe : « `agent007["arme"]` ».

Quand on ne sait pas si un couple clé/valeur est présent dans le dictionnaire, il est possible de tester sa présence avec « in » :

```
if "nom" in agent007:
    print(agent007["nom"])
```

Même s'il est possible de savoir combien il y a d'éléments dans le dictionnaire avec la méthode « len() », ce n'est pas très utile pour parcourir un dictionnaire, car il n'est pas indexé !

Pour parcourir le contenu d'un dictionnaire, on fera appel aux méthodes « keys() », « values() » ou « items() :

- ***keys : parcours par les clés***

Si l'on souhaite afficher toutes les clés d'un dictionnaire, on pourra utiliser le code suivant :

```
for cle in agent007.keys():
    print(cle)
```

cela affichera la liste de toutes les clés, une par ligne.

- ***values : parcours par valeur***

D'une façon similaire, le code suivant :

```
for val in agent007.values():
    print(val)
```

va afficher toutes les valeurs du dictionnaire, une par ligne.

- ***items : parcours des couples cle/valeur***

Enfin le code suivante :

```
for (cle,val) in agent007.items():
    print(cle,":",val)
```

va afficher tous les couples clé/valeur du dictionnaire, un couple par ligne. On voit ici que (cle,val) est en fait un tuple vu plus haut.

Application aux données EXIF

Les données EXIF (Exchangeable Image File Format), sont des informations associées à une photo par un dispositif de prise de vue (appareil photo, téléphone portable…) afin de transmettre, en même temps que l'image, un certain nombre d'informations techniques liées à la prise de vue.

Le format a été établi par la Japan Electronic Industrie Development Association (JEIDA) et est associé aux formats Jpeg, Tiff ou Heif, mais pas PNG ou GIF. Il se présente sous la forme d'une table comprenant un code correspondant à une information, suivie d'une valeur. Chaque information doit être d'un type particulier. On peut retrouver une liste sur le site : https://www.exiv2.org/tags.html.

Par exemple, le code $(272)_{10}$ correspond au nom du modèle d'appareil photo et doit contenir une chaîne de caractère. Il est possible de ne pas remplir tous les champs, ce

qui arrive souvent pour tous les champs GPS si l'appareil photo ne possède pas de capteur GPS.

La plupart des logiciels de traitements d'images permettent d'afficher les métadonnées contenues dans l'EXIF d'une image en remplaçant les codes par des informations compréhensibles :

Du fait de leur forme « code/contenu », les données Exif se prêtent bien à un stockage dans un dictionnaire Python.

Afin de traiter facilement les images sous Python, on peut installer la bibliothèque Pillow qui est une version mise a jour (un « fork ») de la bibliothèque PIL (Python Image Library). Cette installation peut se faire dans la fenêtre « terminal » de Pycharm avec la commande :

```
pip install Pillow
```

Il est ensuite possible d'utiliser la méthode « _getexif() » de l'objet Image de la bibliothèque PIL pour copier les données EXIF d'une photo dans une bibliothèque Python.

Exemple de lecture des informations EXIF d'une photo

`from PIL import Image`
`photo = Image.open("photo.jpeg")`
`exif_data = photo._getexif()`
`print(exif_data[272])`

Résultat
`ILCE-7`

On voit ici les 3 étapes du programme :

- Importation de l'objet « Image » depuis la bibliothèque « PIL »
- Chargement d'un fichier Jpeg nommé « photo.jpeg » dans une nouvelle variable de type « Image » et nommée « photo ».
- Copie des données EXIF dans une nouvelle bibliothèque « exif_data » dont on affiche ensuite la valeur correspondant à la clé « 272 » correspondant au modèle d'appareil photo (ici un Sony Alpha 7).

Attention, dans le code très simple ci-dessus, si on cherche la valeur d'une clé qui n'était pas dans les informations EXIF de l'image, le programme Python va générer une erreur, car la clé n'existe pas dans la bibliothèque.