Programmer avec Python
Cours

Séance n° 2	**Révision des listes - Dictionnaires**

1 Les collections et leurs opérations

Une collection est une structure de données qui permet de regrouper un ensemble de valeurs. Les chaînes de caractères sont un exemple de collection (une collection de caractères), les listes un autre, etc. Certaines opérations sont possibles sur tous les types de collections. Ainsi pour `a` et `b` des collections et i, j des entiers, on a :

- **Concaténation :** `a + b` produit une nouvelle collection qui est composée des éléments de `a` suivis des éléments de `b`. Par exemple, `"bon"+"jour"` donne `"bonjour"`.
- **Multiplication :** `a * i` donne la concaténation de i collections égales à `a`. Par exemple, `5 * "b"` donne `"bbbbb"`.
- **Accès :** `a[i]` donne l'élément de `a` à l'indice i. `a[i:j]` donne la sous-collection de `a` commençant à l'indice i et terminant à l'indice $j-1$. Le premier élément d'une collection est en position 0.
- **Taille :** La fonction **`len`**`(a)` renvoie le nombre d'éléments de la collection `a`. Par exemple, **`len`**`("bonjour")` renvoie 7.

2 Les listes

2.1 Déclaration et accès

Une liste est une collection d'éléments, séparés par des virgules, l'ensemble étant entouré de crochets. Exemples :

```
liste_entiers = [65, 242, 8]
courses = ["pommes", "pâtes", "courgettes", "café"]
```

On peut accéder à chaque élément à partir de son indice. Attention, les indices commencent à zéro ! Ainsi, `liste_entiers[0]` vaut 65 et `courses[2]` vaut `courgettes`.
Pour afficher le contenu d'une liste, on peut utiliser directement la fonction **`print`**

2.2 Modifications

Il est donc possible de modifier un élément d'une liste à partir de son indice : on réalise une simple affectation à l'indice visé :

```
courses = ["pommes", "pâtes", "courgettes", "café"]
courses[2] = "chocolat"
print(courses)
>> ['pommes', 'pâtes', 'chocolat', 'café']
courses[1] = courses[1] + " fraiches"
print(courses)
>> ['pommes', 'pâtes fraiches', 'chocolat', 'café']
```

2.3 Suppression

La fonction **`del`** permet de supprimer un élément quelconque d'une liste à partir de son indice :

```
courses = ["pommes", "pâtes", "courgettes", "café"]
del(courses[1])
print(courses)
>> ['pommes', 'courgettes', 'café']
```

2.3.1 Ajout

L'ajout d'un élément se fait à l'aide d'une sorte de fonction un peu particulière, appelée **méthode**. Nous apprendrons plus précisément ce qu'est une méthode au cours de cette année. Elle s'utilise en écrivant le nom de la variable sur laquelle on l'appelle, puis un point, puis le nom de la méthode, et enfin le ou les arguments de la méthode. Pour nos listes, voici comment appeler la méthode "append" :

```
courses = ["pommes", "pâtes", "courgettes", "café"]
courses.append("pain")
print(courses)
>> ['pommes', 'pâtes', 'courgettes', 'café', 'pain']
```

2.4 Parcours

Le parcours d'une liste se fera préférentiellement avec une boucle `for .. in ..` :

```
# Ex 1 : affichage
courses = ["pommes", "pâtes", "courgettes"]
for element in courses:
    print(element)
>>
pommes
pâtes
courgettes

# Ex 2 : somme des valeurs d'une liste
valeurs = [93, 32, 56, 4, 465, 6]
somme = 0

for element in valeurs:
    somme += element

print(somme)

>> 656
```

2.5 Test d'appartenance

```
courses = ["pommes", "pâtes", "chocolat"]
if "chocolat" in courses:
    print("Le chocolat est présent dans votre liste de courses")
else:
    print("Pas de chocolat dans la liste")

>> Le chocolat est présent dans votre liste de courses
```

3 Les tuples

Il existe un autre type de données appelé **tuple** qui est semblable à une liste mais, comme les chaînes de caractères, n'est pas modifiable. Un tuple est une collection d'éléments séparés par des virgules mais qui n'est pas entouré de crochets :

```
tup = 3, 6, 4, 9
```

Bien que ce soit facultatif, il est vivement conseillé de mettre en évidence le tuple en l'entourant avec des parenthèses, comme le fait elle-même la fonction `print`.

```
tup = (3, 6, 4, 9)
print(tup)
>> (3, 6, 4, 9)
```

Tout ce que vous avez vu avec les listes est applicable aux tuples, à l'exception près qu'un tuple n'est pas modifiable : vous ne pouvez donc utiliser aucune opération de modification : ajout, modification/remplacement d'élément, suppression.

L'intérêt des tuples est qu'ils garantissent que les données ne seront pas modifiées par erreur et qu'ils occupent moins de place en mémoire d'une liste.

4 Les dictionnaires - au programme cette année !

4.1 Définition

Les structures que nous avons vues jusqu'à présent, chaîne de caractères, listes et tuples sont tous des séquences, c'est à dire des suites ordonnées d'éléments. Dans une séquence, on peut facilement accéder à un élément, mais à condition de connaître son indice. Les dictionnaires que nous allons voir sont un autre exemple de structures permettant de stocker plusieurs valeurs mais les éléments qui y seront placés ne sont pas dans un ordre immuable. Dans un dictionnaire, on accèdera à un élément à l'aide d'une **clé** qui peut être numérique, alphabétique ou même un tuple. Les éléments placés dans un dictionnaire peuvent comme une liste être de n'importe quel type : nombres, chaînes, listes, tuples et même dictionnaires.

En Python, un dictionnaire est entouré d'accolades . Un dictionnaire vide sera donc noté `{}`. Pour créer un dictionnaire, on utilise la syntaxe suivante :

```
annuaire = {} # Un dictionnaire vide
```

Chaque élément du dictionnaire est composé d'une paire d'objets : une clé et une valeur. Pour créer un nouvel élément dans un dictionnaire, on note le nom du dictionnaire, suivi de crochets contenant la clé et on affecte la valeur correspondante :

```
annuaire = {} # Un dictionnaire vide
annuaire['Tom'] = "0601020304" # La clé Tom correspond à la valeur
    0601020304
annuaire['Lili'] = "0609080706"
annuaire['Jean'] = "0605040708"
```

Ici, notre dictionnaire est un annuaire contenant des paires de noms, les clés, correspondant à des numéros de téléphone, les valeurs. On pourra ainsi si on le souhaite construire un dictionnaire contenant tous les numéros de téléphone des élèves de la classe. Pour accéder à un élément en particulier, comme avec les listes, on y accède à l'aide de la clé :

```
print(annuaire['Tom'])
>> 0601020304
```

Pour afficher le dictionnaire en entier :

```
print(annuaire)
>> {'Tom': '0601020304', 'Lili': '0609080706', 'Jean': '0605040708'
   }
```

Le dictionnaire apparaît sous la forme d'une série d'éléments séparés par des virgules, le tout étant entouré de deux accolades. Les clés et les valeurs sont séparés de deux points.

4.2 Opérations sur les dictionnaires

4.2.1 Suppression d'un élément

On utilise la fonction **del** là encore :

```
del(annuaire['Jean'])
print(annuaire)
>> {'Tom': '0601020304', 'Lili': '0609080706'}
```

4.2.2 Taille d'un dictionnaire

Là aussi, on utilise la fonction **len** qui renverra le nombre de paires clé-valeur contenues dans le dictionnaire :

```
print(len(annuaire))
>> 2
```

4.3 Test d'appartenance

L'instruction **in** est utilisable avec les dictionnaires, elle permet de tester si une **clé** est présente dans l'annuaire.

```
if "Tom" in annuaire:
    print("On a le numéro de Tom")
```

4.4 Parcours d'un dictionnaire

Pour parcourir un dictionnaire, on utilise une boucle **for**. Chaque clé du dictionnaire sera parcourue dans la variable du **for** :

```
for cle in annuaire:
    print(cle, " : ", annuaire[cle])

>> Tom  :  0601020304
Lili  :  0609080706
```

Attention, les dictionnaires ne sont pas des séquences, l'ordre dans lequel les éléments sont parcourus est donc imprévisible ! Le numéro de Lili pourrait tout à fait être affiché avant celui de Tom !

Lorsque vous avez besoin de parcourir clés et valeurs d'un dictionnaire, il existe une méthode plus rapide, la méthode `.items()`. Elle renvoie un tuple de chacune des paires clé-valeurs et on peut ainsi écrire :

```
for cle, valeur in annuaire.items():
    print(cle, " : ", valeur)

>> Tom  :  0601020304
```

```
Lili   :   0609080706
```

4.5 Extraire toutes les clés ou toutes les valeurs

La méthode `.keys()` renvoie la séquence de toutes les clés utilisées dans le dictionnaire.

```
print(annuaire.keys())
>> dict_keys(['Tom', 'Lili'])
```

La méthode `.values()` renvoie la séquence de toutes les valeurs utilisées dans le dictionnaire.

```
print(annuaire.values())
>> dict_values(['0601020304', '0609080706'])
```

Les dictionnaires seront plus pratiques que les listes lorsqu'on veut sauvegarder des données en fonction d'un nom, d'un identifiant ou tout autre paramètre.