Partie 1

Cours NSI : Langages 2 - structures de données natives en python

1.1 Données structurées et structures de données

Structurer les données signifie les organiser de telle sorte qu'elles aient les mêmes cractéristiques, la même apparence et les mêmes composantes. Il se peut ainsi que lors d'un traitement de données, celles relatives aux dates soient dans différents formats (chaines, objets...). Pour exploiter ces informations, qui peuvent venir de plusieurs sources, il faudra alors uniformiser ces données.

Un autre travail peut consister à organiser les données : les mettre ensemble dans une liste, un tableau, les trier dans l'ordre, mettre en relation certaines de ces données...

Cela dependra aussi de l'activité attendu sur ces données : privilégie-t-on l'entrée/sortie, la recherche, la relation, ou autre...

- Les *structures de données* définissent la manière avec laquelle sont stockées les données dans un langage de programmation.

Parmi les structures de données *natives* les plus courantes en python, on peut citer :

- Celles de *type élémentaire, primitives* : les variables numériques, booléennes,... : **voir le cours Langages1 : variables, opérations et types**
- Les autres : les sequences (listes), les mappages (dictionnaires), les sequences textuelles, les classes et les objets.

On trouvera une documentation plus détaillée à l'adresse : docs.python.org

1.2 Séquences

Une séquence est une structure de données qui stocke une collection d'éléments dans un ordre déterminé : listes, tuples, str

1.2.1 Listes

Une liste est entourée de crochets [ ]. Les données y sont stockées comme on note une liste sur un papier. Python numérote les éléments d'une liste, en partant de 0 (1er élément). C'est une collection d'objets accessibles grâce à leur indice. Exemple : `voyelles[0]`

Un *indice négatif* donne accès à la liste à partir du dernier élément.

- Constructeur : `[]`
- itérable : `in` exemple : `for elem in L`

Les méthodes associées aux listes sont :

- append
- remove

- pop
- index
- extend

Exemples :

```
aeroports = ['CDG','ORY','LIS','NY']
aeroports.remove('LIS')
aeroports
# affiche ['CDG','ORY','NY']
del aeroports[0]
aeroports
# affiche ['ORY','NY']
aeroports.extend(nouveaux_aeroports)
aeroports
['ORY','NY', 'AJA','RTM','LCY']
```

On peut aussi ajouter les fonctions :

- `len` pour avoir la taille de la liste : `len(L)`
- `del` pour effacer un élément : `del(voyelle[0])`
- `list` pour transformer un objet en liste : `L = list(objet)` comme par ex : `alphabet = list('ABCDEFGH')`

Une liste se prête bien à la concaténation, avec l'opérateur +.

```
listA = listB + listC
```

1.2.2 Tuples

Un tuple est une collection utilisée pour créer des listes complexes. Il est non mutable. Il est possible d'imbriquer des tuples l'un dans l'autre, en créant une hierarchie.

Ex avec 3 niveaux d'imbrication : T = (1,2,3,(4,5,6,(7,8,9)))

- Constructeur : `()`
- méthode : `index`
- fonction : `tuple`

1.3 Mappages : les dictionnaires

> Un mappage est une structure de données qui relie 2 informations ou plus, appelées paires clé : valeur. C'est aussi une table de hashage. En python, cette structure est le dictionnaire. Les éléments ne sont plus accessible à l'aide d'un index numérique. Ce n'est pas à proprement parlé une séquence.

Un dictionnaire est entouré d'accolades { }.

- Constructeur : `{}`

```
D = {"A":1,"B":2,"C":3}
```

Les **valeurs** ne sont plus accessibles par leur indice mais par leur **clé** : `dict[cle]`

```
>>> D["A"]
1
```

pour créer un itérable, on peut utiliser les méthodes :

- keys
- values
- items

Exemples : quel itérable pour quelle méthode de dictionnaire

```
for elem in D.keys() -> ["A","B","C"]
for elem in D -> ["A","B","C"]
for elem,value in D.items() -> [("A",1),("B",2),'("C",3)]
for value in D.values() -> [1,2,3]
```

Pour créer une nouvelle entrée, il suffit de faire, si la clé n'existe pas encore : `dict[cle] = value`

Fonctions utiles pour les dictionnaires :

- fonction `dict` : Copie d'un dictionnaire par valeurs
- fonction `del` : effacer un *item*. `del(D["A"])`

1.4 Résumé

type	list	tuple	dict	str
constructeur	`= []`	`= ()`	`= {}`	`= ""`
méthodes	append, pop, insert, index	index	keys, values, items	split, join,...
fonctions	len del list random.shuffle	len tuple	dict keys values item del	str
mutable ?	oui	non	oui	non
copie par	reference	valeur	reference	valeur
ajouter à la fin	append	`a+(4,)`	D[new] = value	+=

Partie 2

Copie d'objets mutables et non mutables

*Rappel : Les variables de type **mutables** peuvent être **modifées** après l'affectation.*

Par contre, pour les non mutables, il n'est pas possible de modifier la donnée en partie (voir exemple avec les tuples). Il faut tout recréer lors d'une nouvelle affectation.

2.1 Copie par valeur d'un objet non mutable

Pour un objet **non mutable** : (immuable) la copie de la variable a dans b va créer un nouvel objet b. La réaffectation de b ne modifiera pas a. C'est comme une copie par **valeur**.

```
>>> a = 3
>>> b = a
>>> b = 4
>>> print(a)
3
```

FIGURE 1 – copie d'un objet non mutable

Pour les **non mutables**, lorsque l'on refait une affectation, cela recréé une nouvelle variable, avec le même nom, mais une autre valeur.

Pour 2 objets copiés par valeur, on peut modifier l'un sans modifier l'autre

2.2 Copie par référence d'un objet mutable

Lorsque l'on utilise le signe = pour une copie d'un objet mutable, cette copie se fait par référence :

```
a = [1, 2, 3]
b = a
```

la copie créé un nouvel objet qui pointe vers la même donnée ; qui a la même **référence**. Il est possible d'avoir plusieurs noms (étiquettes) sur une même valeur.

Il s'agit de la même chose lorsque l'on place un argument de type mutable devant un paramètre lors de l'appel d'une fonction :

```
def ma_fonction(b):
    return b
ma_fonction(a) # a->b
```

La copie d'une variable **mutable** peut alors poser problème de gestion des variables de type construit.

- Exemple de copie par reference* :

```
a = [1, 2, 3]
b = a
b.append(4)
# Le = fait une copie par reference
```

```
print(a)
# affiche
[1, 2, 3,4]
```

FIGURE 2 – copie par référence d'un objet mutable

D'où vient cette bizarrerie ? On peut tester l'égalité entre 2 valeurs avec ==, mais l'identité (emplacement en mémoire de la valeur) se teste avec `is` :

```
values1, values2 = [1, 2, 3], [1, 2, 3]
values1 == values2
# True
values1 is values2
# False
id(values1) == id(values2)
# False
# On peut alors modifier l'un sans modifier l'autre:
values2.append(4)
print(values1)
# affiche
[1, 2, 3]
```

Avec la copie par référence :

```
a = [1, 2, 3]
b = a
b.append(4)
a == b
True
a is b
True
id(a) = id(b)
True
```

2.3 Copie par valeur d'objet mutable

Pour les objets de type `list`, la copie peut se faire par valeur sous certaines conditions :

Pour faire une copie par valeur d'une liste, utiliser l'une des techniques suivantes :

```
# copie par valeur dans les cas suivants
a = [1, 2, 3]
b = list(a) # copie par valeur a->b
a is b # False
b = [:] # copie par valeur a->b
b = a + [] # copie par valeur a->b
```

Partie 3

Construction par comprehension

- Principe

```
new_list = [function(item) for item in <iterable> if <condition(item)>]
```

Prenons un exemple d'une liste :

```
a = [1,4,2,7,1,9,0,3,4,6,6,6,8,3]
```

Nous voulons filtrer les valeurs de cette liste et ne garder que ceux dont la valeur est supérieure à 5 :

```
b = []
for x in a:
  if x > 5:
    b.append(x)

b
# affiche [7, 9, 6, 6, 6, 8]
```

Il est possible de faire exactement ce que fait ce bloc de code en une seule ligne :

```
[x for x in a if x > 5]
[7, 9, 6, 6, 6, 8]
```

- Matrice : Correction de l'exercice *echiquier*

```
lign = list('ABCDEFGH')
col = list('12345678')

ligne1 = [[(l,c) for l in lign] for c in col]
```

- Et pour un dictionnaire :

```
liste = ["liste", "avec", "des", "mots"]
dictionnaire = {len(e) : e for e in liste}
print(dictionnaire)
# Affiche {5: 'liste', 4: 'mots', 3: 'des'}
```