Python 3 –TDs-Fipa1

Saison 1 - épisode 3

1 Découvrons et testons le langage Python

*Un **itérable** est un objet dont on peut parcourir les valeurs, à l'aide d'un for par exemple. Les types str, tuple, list, dict et set sont des itérables.*

1.1 Séquences

Une séquence est un conteneur ordonné d'éléments indexés par des entiers indiquant leur position dans le conteneur. Ce sont des données composites.

3 types de séquences : Les chaines (vues précédemment), les listes, les tuples.

1.2 Liste

Sous Python, nous pouvons considérer une liste comme une collection d'éléments séparés par des virgules, l'ensemble entre crochets :

```
>>> action = ['jouer', 'lire', 'courir', 'dormir', 'manger']
>>> print(action)
['jouer', 'lire', 'courir', 'dormir', 'manger']
>>> print(action[1])
lire
>>> print(len(action))
5
>>> action.append(123)
>>> print(action)
['jouer', 'lire', 'courir', 'dormir', 'manger', 123]
>>>
```

La liste ***action*** contient à sa création 5 chaînes de caractères puis un nombre a été ajouté par la fonction intégrée ***append().***

Les éléments de la liste sont accessibles par un indice entre crochet ***action[n].***

La taille de la liste est donnée par la fonction intégrée ***len() : len(action)***

D'autres fonctions intégrées permettent de **manipuler les listes** :

- **append(elt)** permet d'ajouter l'élément elt à la fin de la liste.
- **extend(li)** permet d'ajouter l'ensemble des éléments de la liste li à la suite de la liste.
- **insert(pos, elt)** permet d'insérer l'élément el à l'index pos spécifié.
- **remove(elt)** permet de supprimer la première occurrence de la valeur el.
- **pop([pos])** permet de renvoyer et de supprimer l'élément d'index pos, ou le dernier élément si aucune valeur n'est passée à pop.
- **del** est une instruction intégrée qui permet de supprimer un élément dont on connaît la position.
- **sort()** trie la liste. Elle ne renvoie pas de nouvelle liste.
- **reverse()** inverse l'ordre des éléments de la liste.

Testons !

```
>>> action.append(123)
>>> print(action)
['jouer', 'lire', 'courir', 'dormir', 'manger', 123]
>>> del(action[3]
... )
>>> print(action)
['jouer', 'lire', 'courir', 'manger', 123]
>>> action[4]=666
>>> print(action)
['jouer', 'lire', 'courir', 'manger', 666]
>>>
```

1.2.1 Accès aux éléments par indice

L'opérateur **[]** nous permet de manipuler les listes simples ou complexes.

```
>>> print(action)
['jouer', 'lire', 'courir', 'manger', 666]
>>> print(action[1])
lire
>>> print(action[1:3])
['lire', 'courir']
>>> print(action[2:3])
['courir']
>>> print(action[2:])
['courir', 'manger', 666]
>>> print(action[:2])
['jouer', 'lire']
>>> print(action[-1])
666
>>> print(action[-2])
manger
>>>
```

```
>>> equipe = [["Eloise","courir"],["Antoine","nager"],["Elsa","ramer"]]
>>> print(equipe)
[['Eloise', 'courir'], ['Antoine', 'nager'], ['Elsa', 'ramer']]
>>> print(equipe[1])
['Antoine', 'nager']
>>> print(equipe[1][1])
nager
>>>
```

1.2.2 Modification des listes

```
>>> nombres = [5,9,2,7,10,1]
>>> nombres.sort()
>>> nombres
[1, 2, 5, 7, 9, 10]
```

```
>>> nombres.reverse()
>>> nombres
[10, 9, 7, 5, 2, 1]
>>> nombres.index(5)
3
```

```
>>> nombres.remove(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

```
>>> nombres.remove(7)
>>> nombres
[10, 9, 5, 2, 1]
>>>
```

Quelques exercices de Slicing (tranche):

```
>>> mots=['livre','tablette','montre','reveil']
>>> mots[1]='null'
>>> mots
['livre', 'null', 'montre', 'reveil']
>>> mots[1]=10
>>> mots
['livre', 10, 'montre', 'reveil']
>>> mots[2:2]="pot"
>>> mots
['livre', 10, 'p', 'o', 't', 'montre', 'reveil']
>>> mots[2:2]='poterie'
>>> mots
['livre', 10, 'p', 'o', 't', 'e', 'r', 'i', 'e', 'p', 'o', 't', 'montre', 'reveil']
>>> mots[2:2]=['pile']
>>> mots
['livre', 10, 'pile', 'p', 'o', 't', 'e', 'r', 'i', 'e', 'p', 'o', 't', 'montre', 'reveil']
>>>
```

1.2.3 Création d'une liste avec la fonction range()

La fonction intégrée ***range()*** renvoie une séquence de nombres entiers de valeurs croissantes et différent d'une unité. On peut aussi utiliser range() avec deux ou trois arguments séparés par des virgules, afin de générer des séquences de nombres plus spécifiques.

Les 3 arguments : range (1ere valeur à générer, dernière valeur à générer, le pas à sauter)

range (10,-10,-3) « de 10 à -9 avec un pas de -3 » ou range (10) « de 0 à 9 » ou range (8,14) « de 8 à 13 » ou range (5,25,5) « de 5 à 24 avec un pas de 5 » …

Testons !

range() sera associé à la fonction intégrée list() pour générer une liste :

```
>>> list(range(8))
[0, 1, 2, 3, 4, 5, 6, 7]
>>> list(range(10,-10,-3))
[10, 7, 4, 1, -2, -5, -8]
>>> list(range(5,25,5))
[5, 10, 15, 20]
>>> list(range(8,14))
[8, 9, 10, 11, 12, 13]
>>>
```

```
>>> nombres
[10, 9, 5, 2, 1]
>>> for n in nombres:
...     print(n,end=' ')
...
10 9 5 2 1 >>>
>>> for n in nombres:
...     print(n*2)
...
20
18
10
4
2
>>>
```

- ***Parcours d'une liste à l'aide de for, range() et len()***

```
>>> for a,b in equipe:
...     ch= "la personne est {} pour l'action de {}"
...     print(ch.format(a,b))
...
la personne est Eloise pour l'action de courir
la personne est Antoine pour l'action de nager
la personne est Elsa pour l'action de ramer
>>>
```

1.2.4 Opérations sur les listes

Les opérations de **concaténation, de multiplication, de copie sont** applicables aux listes

Testons !

```
>>> nombres=[10,9,5,2,1]
>>> nombres
[10, 9, 5, 2, 1]
>>> nombres2=nombres*2
>>> nombres2
[10, 9, 5, 2, 1, 10, 9, 5, 2, 1]
>>> n=nombres[1]*10
>>> n
90
>>> nombres[2]=n/3
>>> nombres
[10, 9, 30.0, 2, 1]
>>>
```

```
>>> mots
['livre', 10, 'pile', 'p', 'o', 't', 'e', 'r', 'i', 'e', 'p', 'o', 't', 'montre', 'reveil']
>>> motsimple=mots
>>> motsimple
['livre', 10, 'pile', 'p', 'o', 't', 'e', 'r', 'i', 'e', 'p', 'o', 't', 'montre', 'reveil']
>>>
>>>
>>> motsimple=motsimple[0]+motsimple[2]
>>> motsimple
'livrepile'
>>>
```

```
>>> mots=["livre","boite","montre","tablette"]
>>> mots2=["lait","sucre","miel"]
>>> mots+mots2
['livre', 'boite', 'montre', 'tablette', 'lait', 'sucre', 'miel']
>>>
```

1.3 Tuple

Le tuple est une collection d'éléments séparés par des virgules. Il est recommandé d'enfermer le tuple dans une paire de parenthèse bien que cela ne soit pas obligatoire pour une meilleure lisibilité.

Comme la chaine de caractères, ***le tuple est non modifiable.***

```
>>> tupo = ('a','t','t','i','t','u','d','e')
>>> print(tupo)
('a', 't', 't', 'i', 't', 'u', 'd', 'e')
>>>
```

Les opérations sur les tuples sont les mêmes que sur les listes sauf la modification ou la suppression (del, remove())

La taille d'un tuple est retournée par ***len().***

Le tuple est préférable à la liste pour protéger des listes de valeurs non modifiables.

1.4 Dictionnaire (dict)

Les types de données composites que nous avons testés comme les *chaînes*, les *listes* et les *tuples* sont des *séquences*, c'est-à-dire des suites ordonnées d'éléments.

Dans une séquence on accède à un élément quelconque à l'aide d'un index (un nombre entier). Il faut connaitre la valeur de l'index.

Les ***dictionnaires*** sont un autre *type composite.* Ils sont modifiables, mais ce ne sont pas des séquences. Ce sont des tableaux associatifs. Le dictionnaire est composé de couples constitué d'une clé associée à une donnée.

La clé, index spécifique, permet d'accéder à une et une seule donnée. La ***clé***, peut être alphabétique, numérique, ou même d'un type composite sous certaines conditions.

Comme dans une liste, les données mémorisées dans un dictionnaire peuvent être de type numérique, chaine, liste, tuple, dictionnaire …

Le dictionnaire est itérable mais il n'est pas ordonné.

1.4.1 Création d'un dictionnaire

Dictionnaire vide : d = {} ou d1=dict()

Affectation de valeur : d1['nom ']=3 ou d1={'nom' :3, 'exemple' :4}

Recherche : print(d1['nom '] => 3, print(d1) => {'nom' : 3, 'exemple' :4}

Testons !

```
>>> dico = {}
>>> dico['arbre1']='Peuplier'
>>> dico['arbre2']='Platane'
>>> dico['arbre3']='Chêne'
>>> print(dico)
{'arbre2': 'Platane', 'arbre3': 'Chêne', 'arbre1': 'Peuplier'}
>>> print(dico['arbre2'])
Platane
```

```
>>> docu=dict(fleur1='IRIS',fleur2='BLEUET')
>>> docu
{'fleur2': 'BLEUET', 'fleur1': 'IRIS'}
>>> bouquet=([('IRIS',3),('TULIPE',6)])
>>> bouquet
[('IRIS', 3), ('TULIPE', 6)]
>>> bouquet2=dict([('IRIS',3),('TULIPE',6)])
>>> bouquet2
{'TULIPE': 6, 'IRIS': 3}
```

```
In [17]: bouquet2=dict([('IRIS',3),('TULIPE',2)])
    ...: print (bouquet2)
    ...:
{'IRIS': 3, 'TULIPE': 2}
```

1.4.2 Opérations sur les dictionnaires

- **Instructions intégrées ou méthodes**

  del *pour supprimer des éléments d'un dictionnaire* : **del d1['toctoc ']**

  len() *renvoie la taille du dictionnaire (le nombre d'éléments)* : **d1.len()**

  keys() *renvoie la valeur des clés du dictionnaire* **: d1.keys()**

  items() *extrait du dictionnaire une séquence équivalente de tuples* : **d1.items()**

  copy() *effectue une vraie copie du dictionnaire* : **d1.copy()**

  values() *renvoie les valeurs des éléments du dictionnaire* **: d1.values()**

  sorted() *pour trier* **: sorted(d1.values())**

  list() ou tuple() *pour convertir le dictionnaire si besoin* : **list(d1.values())**

Testons !

```
>>> dico={}
>>> dico['arbre1']='Peuplier'
>>> dico['arbre2']='Platane'
>>> dico['arbre3']='Chêne'
>>> print(dico)
{'arbre2': 'Platane', 'arbre1': 'Peuplier', 'arbre3': 'Chêne'}
>>> dico.keys()
dict_keys(['arbre2', 'arbre1', 'arbre3'])
>>> dico.values()
dict_values(['Platane', 'Peuplier', 'Chêne'])
>>> sorted(dico.values())
['Chêne', 'Peuplier', 'Platane']
>>> del dico['arbre2']
>>> print(dico)
{'arbre1': 'Peuplier', 'arbre3': 'Chêne'}
```

Ecrivez un script pour tester la méthode items() et les 2 copies de dictionnaire :

```
>>> dico ={}
>>> dico={'ponmmes':45,'orange': 78,'banane':34}
>>> print(dico.items())
dict_items([('ponmmes', 45), ('banane', 34), ('orange', 78)])
>>> tuple(dico.items())
(('ponmmes', 45), ('banane', 34), ('orange', 78))
>>> test=dico
>>> print("dico :", dico)
dico : {'ponmmes': 45, 'banane': 34, 'orange': 78}
>>> print("test :", test)
test : {'ponmmes': 45, 'banane': 34, 'orange': 78}
>>> diconew=dico.copy()
>>> print("dico 1 :", dico)
dico 1 : {'ponmmes': 45, 'banane': 34, 'orange': 78}
>>> print("diconew 1 :", diconew)
diconew 1 : {'ponmmes': 45, 'banane': 34, 'orange': 78}
>>> diconew['poire']=35
>>> print("dico 2 :", dico)
dico 2 : {'ponmmes': 45, 'banane': 34, 'orange': 78}
>>> print("diconew 2 :", diconew)
diconew 2 : {'ponmmes': 45, 'banane': 34, 'orange': 78, 'poire': 35}
>>>
```

Itération

Pour obtenir clés et valeurs **la boucle For :**

```
for k in dico1.keys() :

    print("cle : ",k," valeur :
"dico1[k])
```

Test d'appartenance

L'instruction **in** associée à **If :**

```
If 'truc' in dico1 :
    print("hello truc")
else :
    print("perdu")
```

Testons !

```
>>>
>>> if "arbre2" in dico:
...     print("gagné")
... else:
...     print("perdu")
...
perdu
>>>
```

```
>>> dicArbre = {}
```

```
>>> dicArbre[(1,1)]= 'Peuplier'
>>> dicArbre[(1,2)]= 'Peuplier'
>>> dicArbre[(1,3)]= 'Peuplier'
>>> dicArbre[(2,3)]= 'Chataignier'
>>> dicArbre[(2,4)]= 'Chataignier'
>>> dicArbre[(3,4)]= 'Chêne'
>>> print(dicArbre)
{(1, 2): 'Peuplier', (1, 3): 'Peuplier', (2, 3): 'Chataignier', (3, 4): 'Chêne',
(1, 1): 'Peuplier', (2, 4): 'Chataignier'}
>>> for k in dicArbre.keys():
...     print("clé : ",k,"...valeur : ",dicArbre[k])
...
clé :  (1, 2) ...valeur :  Peuplier
clé :  (1, 3) ...valeur :  Peuplier
clé :  (2, 3) ...valeur :  Chataignier
clé :  (3, 4) ...valeur :  Chêne
clé :  (1, 1) ...valeur :  Peuplier
clé :  (2, 4) ...valeur :  Chataignier
>>>
```

Ensemble (set)

Def : Collection itérable, non ordonnée d'éléments hachables et uniques

Deux ensembles : **set** (modifiable) et **frozenset** (non modifiable)

Testons !

Création

```
>>> x=set('sosie')
>>> y=set('sissi')
>>> x
{'s', 'i', 'o', 'e'}
>>> y
{'s', 'i'}
```

Opérations ensemblistes

```
>>> x-y
{'o', 'e'}
>>> y-x
set()
>>> x|y
{'s', 'e', 'i', 'o'}
>>> x & y
{'s', 'i'}
>>> x ^ y
{'o', 'e'}
>>>
```

Test

```
>>> 's' in y
True
>>> 'o' in y
False
```

Mise en pratique

Exercice 1

Ecrivez un programme qui recherche le plus grand élément présent dans une liste donnée. Par exemple, si on l'appliquait à la liste [32, 5, 12, 8, 3, 75, 2, 15], ce programme devrait afficher : **le plus grand élément de cette liste a la valeur 75.**

Exercice 2

Ecrivez un programme qui analyse un par un tous les éléments d'une liste de nombres (par exemple celle de l'exercice précédent) pour générer deux nouvelles listes. L'une contiendra seulement les nombres pairs de la liste initiale, et l'autre les nombres impairs.

Par exemple, si la liste initiale est celle de l'exercice précédent, le programme devra construire une liste pairs qui contiendra [32, 12, 8, 2], et une liste impairs qui contiendra [5, 3, 75, 15]. Astuce : pensez à utiliser l'opérateur modulo (%) déjà cite précédemment.

Exercice 3

Ecrire une boucle de programme qui demande à l'utilisateur d'entrer des notes d'élèves. La boucle se terminera seulement si l'utilisateur entre une valeur négative. Avec les notes ainsi entrées, construire progressivement une liste. Apres chaque entrée d'une nouvelle note (et donc à chaque itération de la boucle), afficher le nombre de notes entrées, la note la plus élevée, la note la plus basse, la moyenne de toutes les notes.

Exercice 4

Soient les listes suivantes :

```
t1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

```
t2 = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet', 'Aout', 'Septembre', 'Octobre', 'Novembre', 'Décembre']
```

Ecrivez un petit programme qui insère dans la seconde liste tous les éléments de la première, de telle sorte que chaque nom de mois soit suivi du nombre de jours correspondant : `['Janvier', 31, 'Février', 28, 'Mars', 31, etc.]`.

Affichez la liste.

Exercice 5

Ecrire un programme qui permette de trouver la fréquence d'une lettre dans une liste de mots et les mots contenant la lettre en question ainsi que leur nombre.
Il faut :

- Créer une liste de 10 mots en lettre minuscule, sans accent pour simplifier ;
- Saisir la lettre au clavier en forçant la casse en minuscule ;
- Initialiser un compteur de lettre et un compteur de mot à zéro;

- Créer une boucle qui recherche cette lettre dans la liste des mots, si la lettre est trouvée dans un mot, comptez-la. Stockez le mot dans une liste de mots trouvés. Attention, le mot ne doit pas être enregistré plusieurs fois !

A la fin du programme, affichez la fréquence de la lettre et le nombre de mots ainsi que la liste des mots trouvés.

Exercice 6

Un nombre premier est un nombre qui n'est divisible que par un et par lui-même. Ecrivez un programme qui établit la liste de tous les nombres premiers compris entre 1 et 1000, en utilisant la méthode du crible d'Eratosthène :

• Créez une liste de 1000 éléments, chacun initialisé à la valeur 1.

• Parcourez cette liste à partir de l'élément d'indice 2 : si l'élément analysé possède la valeur 1, mettez à zéro tous les autres éléments de la liste, dont les indices sont des multiples entiers de l'indice auquel vous êtes arrivés.

Lorsque vous aurez parcouru ainsi toute la liste, les indices des éléments qui seront restés à 1 seront les nombres premiers recherchés.

En effet : A partir de l'indice 2, vous annulez tous les éléments d'indices pairs : 4, 6, 8, 10, etc. Avec l'indice 3, vous annulez les éléments d'indices 6, 9, 12, 15, etc., et ainsi de suite. Seuls resteront à 1 les éléments dont les indices sont effectivement des nombres premiers.

Affichez cette liste de nombres premiers.

Exercice 7

Ecrivez un script qui crée un mini-système de base de données fonctionnant à l'aide d'un dictionnaire, dans lequel vous mémoriserez les noms d'une série de copains, leur âge et leur taille.

Dans le dictionnaire, le nom de l'élève servira de clé d'accès, et les valeurs seront constituées de tuples (âge, taille), dans lesquels l'âge sera exprimé en années (donnée de type entier), et la taille en mètres (donnée de type réel).

Votre script devra comporter deux blocs de code : un pour le remplissage du dictionnaire, et l'autre pour sa consultation.

Dans le bloc de remplissage, utilisez une boucle pour accepter les données entrées par l'utilisateur.

Le bloc de consultation comportera elle aussi une boucle, dans laquelle l'utilisateur pourra fournir un nom quelconque pour obtenir en retour le couple ≪ âge, taille ≫ correspondant.

Le résultat de la requête à afficher devra être une ligne de texte bien formatée, telle par exemple :

≪ Nom : Alain Terieur - âge : 25 ans - taille : 1.84 m ≫.

Pour obtenir ce résultat utilisez ce code :

```
print("Nom : {} - âge : {} ans - taille : {} m.".\ format(nom, age, taille))
```

Exemple d'affichage lors de l'exécution du programme :

```
Choisissez : (R)emplir - (C)onsulter - (T)erminer : R
Entrez le nom (ou <enter> pour terminer) : lol
Entrez l'âge (nombre entier !) : 5
Entrez la taille (en mètres) : 8
Entrez le nom (ou <enter> pour terminer) : lila
Entrez l'âge (nombre entier !) : 9
Entrez la taille (en mètres) : 6
Entrez le nom (ou <enter> pour terminer) :
Choisissez : (R)emplir - (C)onsulter - (T)erminer : C
Entrez le nom (ou <enter> pour terminer) : lol
Nom : lol - âge : 5 ans - taille : 8.0 m.
Entrez le nom (ou <enter> pour terminer) :
Choisissez : (R)emplir - (C)onsulter - (T)erminer : T
>>>
```