Chapitre II : p-uplets et listes

On appelle **collection** un type de données permettant de regrouper des données.
En python, il existe les p-uplets, les listes, les dictionnaires, les ensembles, ...
Ces collections sont itérables, c'est-à-dire qu'il est possible d'utiliser une boucle `for` sur chaque élément de la collection.

> Rappel : la boucle `for` s'utilise toujours en python sous la forme :
> `for` *variable* `in` *iterable* :
> Elle a été utilisée dans le cas où les itérables étaient de la forme `range(n, m, p)` (même si le plus souvent on ne met qu'un seul ou deux arguments) ou une chaîne de caractères.
> Dans ce cas, *variable* prend successivement les valeurs obtenues avec `range` ou chaque caractère si c'est une chaîne de caractères.

Nous utiliserons donc dans ce chapitre cette boucle sur des **n-uplets** ou sur des **listes**.
Les collections de données peuvent être modifiables (dont on peut changer la valeur de chaque donnée) ou non-modifiable.
D'autres langages que python ne possèdent pas toutes ces collections ou privilégient l'une d'elles (les tableaux en C - qui s'apparentent aux listes de python - les listes en Scheme, les tableaux en PHP - dont certains s'apparentent à des dictionnaires - les tableaux, les dictionnaires, les ensembles en Javascript,...).

I - Les p-uplets

a) Mes premiers p-uplets (ou tuples en python)

Les tuples sont des suites de valeurs **non-modifiables**. Les valeurs de cette suite sont séparées par une virgule et délimitées par des parenthèses.
Par exemple :

```
t = (1,2,3)
t = ()              #pour un tuple vide - qui ne présente que peu d'intérêt
t = (7,)            #pour un tuple avec un unique élément (là aussi peu intéressant)
```

Remarque : `t = 1,2,3` définira le même tuple que `t = (1,2,3)`.

L'accès à chaque élément d'un tuple se fait à l'aide de son index (ou indice), comme pour les chaînes de caractères.
Par exemple si on définit `a = (12,14,0,1,5,7)`, que renverra `a[2]` ? `a[-1]` ?

Qu'écrire pour obtenir la valeur 5 ?

Un tuple peut contenir tout type d'objets : entiers, flottants, chaînes de caractères, tuples, ...

Comme pour les chaînes de caractères, il est possible de déterminer le nombre d'éléments d'un tuple : `len(a)`.
Que se passera-t-il si on écrit `a[2] = 10` ?

C'est d'ailleurs cette propriété d'être non-modifiable qui rend ce type intéressant (en plus d'être un peu moins gourmand en ressource système que les listes).

Il est simple d'utiliser la boucle `for` avec un tuple :

```
1 monTuple = (2,3,4,5)
2 for elt in monTuple:
3     print(2**elt)
```

Qu'affiche ce programme ?
Faire un tableau avec les différentes valeurs prises par `elt` et la valeur affichée.

Comme pour tout élément itérable en python, il est facile de tester si un élément appartient ou non au tuple : `x in monTuple` ou bien `x not in monTuple`.

b) Tuples pour créer des chaînes de caractères

Un tuple peut tout à fait contenir des valeurs de différents types. Par exemple :
`eleve = ("Quentin",19,"février")` qui peut indiquer le jour anniversaire d'un élève.
On peut alors essayer de créer la variable `phraseAnniv` qui serait de la forme « L'anniversaire de Quentin est le 19 février ».
Voilà ce que pourrait être une première idée :

```
phraseAnniv = "L'anniversaire de "+eleve[0]+" est le "+eleve[1]+" "+eleve[2]
```

Mais cette syntaxe créé l'erreur :
`TypeError: Can't convert 'int' object to str implicitly`. Que signifie ce message ?

Deux façons de régler le problème :

- **méthode format sur les chaînes de caractères (*string*) :**

  ```
  phraseAnniv = "L'anniversaire de {} est le {} {}".format(eleve[0],eleve[1],eleve[2])
  ```

  Les accolades vides seront remplacées par les éléments successifs présents en termes d'arguments de la méthode format et transformés en chaînes de caractères. Ces derniers sont pris dans l'ordre et il doit y en avoir autant que d'accolades.
  La variable `phraseAnniv` devient alors :

  On peut aussi mettre des nombres (index) dans ces accolades, ce qui est utile en cas de répétition :

  ```
  phraseAnniv = "L'anniversaire de {0} est le {1} {2}.
       Quelqu'un d'autre est né le {1}? ".format(eleve[0],eleve[1],eleve[2])
  ```

  On obtiendra :

- **Solution valable à partir de la version 3.6 de python**

  Cette deuxième méthode est une contraction de ce qui précède :

  ```
  phraseAnniv = f"{eleve[0]} est né un {eleve[1]} {eleve[2]}."
  ```

  Elle est assez troublante à première vue avec le « f » placé avant les guillemets. Elle indique un type de chaîne de caractères, les f-strings.

  Que devient la variable `phraseAnniv` ?

  Remarque : En vitesse d'exécution, cette méthode est plus efficace que la précédente, mais cela ne peut se voir que sur un très grand nombre d'applications de la méthode.

  Informations sur les f-strings : https://cito.github.io/blog/f-strings/

On trouve un site intéressant sur ces formatages :
https://www.docstring.fr/blog/le-formatage-des-chaines-de-caracteres-avec-python/.

c) Affectation multiple

Sans en parler, nous avons déjà créé des tuples en affectant simultanément des valeurs à plusieurs variables :

- pour permuter les valeurs de deux variables `a` et `b` : `a,b = b,a` ;
- Cela fonctionnerait aussi avec trois : `b,c,a = a,b,c` ;
- Pour définir le quotient et le reste dans la division euclidienne de `a` par `b` : `q,r = .....`

  Cela peut aussi se faire à l'aide de la fonction divmod : `q,r = divmod(a,b)`

Cette affectation multiple est également très pratique dans les fonctions. Par exemple, on peut compléter cette fonction pour obtenir la conversion en heures, minutes, secondes un temps exprimé en secondes.
On commentera la fonction puis on effectuera des tests d'erreurs sur la nature de l'argument de la fonction.

```
def convertir_temps(s):
    """s est un nombre de secondes
    On  obtient le nombre d'heures de minutes et de secondes équivalent às secondes"""
    assert (isinstance(s,int) or isinstance(s,float)), "on veut un entier ou un décimal"
    assert s>0, "s doit être positif"
```

```
    return int(heures), int(minutes), secondes
```

Que vaut `t` si on écrit : `t = convertir_temps(5000)` ? `t` vaut dans ce cas .....

Si on écrit `a, b, c = convertir_temps(40000)`, `a` vaut .....

II - Les listes

En Python, les listes sont des objets qui peuvent en contenir d'autres. Elles s'apparentent aux tableaux dans beaucoup d'autres langages. Ce sont donc des séquences, comme les chaînes de caractères, mais au lieu de contenir des caractères, elles peuvent contenir n'importe quel objet.
Elles servent à stocker des données d'un seul type (même s'il est possible de mettre des données de types différents mais c'est à proscrire car le traitement serait source d'erreur).

a) Premières listes

`lst = list()` permet de créer une nouvelle liste qui sera vide (ne contenant aucun élément). Cela peut aussi se faire sous la forme : `lst = []`.
Cas de listes non vides :

- `lst = [1,2,3]` crée une liste de 3 entiers ;
- `lst = [["a","b"], ["c","d","e"], ["f","g"]]` est une liste de 3 éléments, chacun étant une liste de 2 ou 3 caractères ;
- `lst = list(range(10))` construit la liste des 10 entiers allant de 0 à 9 ;
- `lst = list(range(5,15))` construit la liste des 10 entiers allant de 5 à 14 ;
- `lst = list(range(10,20,2))` construit la liste des entiers allant de 10 à 19 avec un pas de 2, ce qui donne `[10,12,14,16,18]`.
- Pour créer rapidement une liste de n termes identiques, on utilise l'opérateur `*` :
  `lst = [0]*10` créer une liste `lst` contenant 10 fois le terme 0.

Comme pour les chaînes de caractères et les tuples, il est possible de récupérer la valeur d'un élément de la liste (`lst[2]` pour le 3ème) et la longueur de la liste (`len(lst)`).

Il est aussi possible de vérifier la présence d'un élément.

- `if 2 in lst:` vérifie si la valeur 2 est dans la liste `lst`.
- `lst.index(2)` donne la première position de la valeur 2 dans la liste `lst`. Il faut cependant avoir vérifié auparavant que cette valeur est bien dans la liste.
- `lst.count(2)` compte le nombre de fois où la valeur 2 est présente dans la liste (nombre d'occurrences de 2).

Exercice : Écrire en python une fonction `somme_liste` d'arguments `liste` et `n` où `liste` est une liste et `n` un entier naturel.
Cette fonction devra renvoyer la somme des `n` premières valeurs de `liste` (on vérifiera que `n` n'est pas plus grand que la longueur de la liste).

```
def somme_liste(liste,n):
    """liste est une liste et n un entier naturel
    On obtient la somme des n premiers termes de la liste
    C'est la somme de tous les termes si n est supérieur àla longueur de la liste"""
```

Remarques :

- pour ajouter tous les termes d'une liste, il existe la fonction `sum` ;
- cette fonction peut aussi fonctionner lorsque les éléments de la liste sont des chaînes de caractères, l'opérateur + indiquant une concaténation de chaîne.
  Il faut cependant modifier l'initialisation de la variable `somme` et lui donner la valeur `""`.

Exercice : On dispose des listes `l1 = [[1,2],[3,4,5],[6,7,8,9],[10,11]]` et `l2 = [12,13]`.

1. Comment accéder à la valeur 5 dans la liste `l1` ?
2. On écrit `l1.extend(l2)`. Que renvoie alors `len(l1)` ?

.....

.....

.....

b) Méthodes pour modifier une liste

Contrairement aux tuples, on peut modifier le contenu d'une liste.
Méthodes sur les listes :
Les exemples ci-dessous seront tous construits à partir de la liste `l1 = [12,15,9,-4,17,12,5]`.

- Modifier une valeur de la liste : `l1[1] = -7` modifiera le terme d'index 1 de la liste.
  `l1` devient alors `[12,-7,9,-4,17,12,5]`.
- Modifier l'ordre des éléments de la liste :
  - sort() : ordonne les valeurs dans l'ordre croissant.
    `l1.sort()` modifie `l1` qui devient `[-4, 5, 9, 12, 12, 15, 17]`
    Remarque : `l2 = sorted(l1)` permet de créer une nouvelle liste `l2` qui contiendra les valeurs de `l1` rangée dans l'ordre croissant. `l1` n'est pas modifié dans ce cas.
  - reverse() : renverse l'ordre des valeurs de la liste.
    `l1.reverse()` donne `[5,12,17,-4,9,15,12]`.
  - Pour obtenir les valeurs de la liste dans l'ordre décroissant, on peut ainsi écrire `l1.sort().reverse()` ou plus simplement `l1.sort(reverse=true)` (même remarque avec la fonction sorted).
- Ajouter des éléments à la liste :
  - `l1.append(2)` ajoute le terme 2 à la fin de la liste.
  - `l1.extend(l2)` ajoute les éléments de la liste `l2` à ceux dans la liste `l1`.
    C'est un équivalent de `l1 = l1+l2` ou `l1 += l2`. On utilise plutôt l'opérateur `+` pour mettre les éléments de deux listes dans une troisième sans modifier les listes de départ.
  - `l1.insert(1,25)` : insère le nombre 25 à l'index 1 de la liste.
    `l1` devient alors `[12,25,15,9,-4,17,12,5]`.
- Enlever des éléments à la liste.

- `l1.pop()` enlève le dernier élément de la liste `l1`. Dans ce cas, `l1` devient `[12,15,9,-4,17,12]`.
  Il est possible d'enlever n'importe quel élément d'index compris entre 0 et `len(l1)-1` :
  `l1.pop(2)` supprime l'élément d'index 2, `l1` devient alors `[12,15,-4,17,12,5]`.
- `del(l1[2])` supprime l'élément d'index 2 de la liste.
  Remarque : : sur `l1`, le résultat est le même qu'avec `l1.pop(2)`, la différence est que la méthode `pop` renvoie la valeur supprimée.
- `l1.remove(-4)` permet de supprimer la valeur `-4` de la liste. Attention, si la valeur n'est pas présente dans la liste, python lèvera une erreur, donc il faudra s'assurer de sa présence pour utiliser `remove` (par exemple `if -4 in l1:`).
- `l1.clear()` vide entièrement la liste : ce n'est que très rarement utile.

– Extraire des éléments d'une liste (*slice*)

Une liste `l1` étant construite, il est possible d'en extraire une partie dans une autre liste `l2`.

Prenons par exemple `l1=list(range(20))`.

- `l2=l1[:5]` : est une copie des 5 premiers éléments de la liste `l1`.
- `l2=l1[8:]` : est une copie des éléments de la liste `l1` de l'index 8 jusqu'à la fin.
  Remarque : `l2=l1[-2:]` sera alors une copie des deux derniers éléments de la liste `l1`.
- `l2=l1[8:11]` : est une copie des éléments de la liste `l1` de l'index 8 jusqu'à l'index 10.

Une méthode très classique de construction de liste :

– on part d'un liste vide ;

– on ajoute des termes un à un à l'aide d'une boucle.

Exemple : On peut créer une fonction qui récupère sous la forme d'une liste toutes les lettres comprises entre `"a"` et `"m"` :

```
def extraire_lettres(chaine):
    lst = []                    #On crée la liste vide
    for let in chaine:
      if "a" <=let<= "m" :      #Il existe une relation d'ordre sur les chaînes
        lst.append(let)         #On ajoute un élément à la fin de la liste
    return lst
```

Exercice : Pour chaque ligne du programme suivant, donner le contenu de la liste `lst` :

```
lst=list(range(3,6))
lst.append(3)
lst.remove(3)
lst.extend([7, 10, 5, 2])
lst.reverse()
b = lst.pop()
lst.pop(3)
lst.insert(2, b)
sorted(lst)
lst.append(1)
lst.sort(reverse=True)
```

c) Duplication de liste

Une liste étant donnée, il est courant d'en faire une copie pour effectuer des modifications sur son contenu tout en conservant son contenu initial.

Écrire les lignes ci-dessous dans un éditeur python.

Voilà ce qui devrait être votre première idée :

```
l1 = [2,4,8,3,1,7,5,6,9,7]
l2 = l1
l2[3] = 0
```

En effectuant `print(l2)`, on obtient : .....

En effectuant `print(l1)`, on obtient : .....

Cette instruction a modifié `l2` comme souhaité mais elle a aussi modifié `l1`.

Trois façons pour y remédier :
`l2 = list(l1)`, `l2 = l1[:]` et `l2 = l1.copy()`
Ces trois méthodes sont équivalents et résolvent le problème précédent.
Cependant, elles ne suffisent pas pour des listes plus complexes.
Par exemple :

```
l1 = [["a","b"],["c","d","e"],["f","g"]]
l2 = list(l1)
l2[1][0] = "k"
```

Dans ce cas `l1` devient `[["a","b"],["k","d","e"],["f","g"]]` alors qu'en écrivant `l2[1] = ["k","d","e"]`, `l2` vaudrait `[["a","b"],["k","d","e"],["f","g"]]`, alors que `l1` serait toujours égal à `[["a","b"],["c","d","e"],["f","g"]]`.

Pour résoudre le problème avec des listes de listes, il faut effectuer une copie en profondeur de la liste. Cela s'effectue à l'aide du module `copy` :

```
import copy
l1 = [["a","b"],["c","d","e"],["f","g"]]
l2 = copy.deepcopy(l1)
l2[1][0] = "k"
```

III - Les compréhensions de listes

Les compréhensions de liste (*list comprehensions* en anglais) sont un moyen de filtrer ou modifier une liste très simplement. La syntaxe est déconcertante au début mais c'est très puissant.

a) Des exemples pour comprendre l'intérêt

Exemple 1 : **La légende de l'échiquier de Sissa**
Le roi Belkib (Indes) promit une récompense fabuleuse à qui lui proposerait une distraction qui le satisferait.
Lorsque le sage Sissa, fils du Brahmine Dahir, lui présenta le jeu d'échecs, le souverain, demanda à Sissa ce que celui-ci souhaitait en échange de ce cadeau extraordinaire.
Sissa demanda au prince de déposer un grain de riz sur la première case, deux sur la deuxième, quatre sur la troisième, et ainsi de suite pour remplir l'échiquier en doublant la quantité de grain à chaque case.
Le prince accorda immédiatement cette récompense sans se douter de ce qui allait suivre.

Son conseiller lui expliqua qu'il venait de précipiter le royaume dans la ruine car les récoltes de l'année ne suffiraient pas à payer Sissa.

Nous allons dans un premier temps créer une liste permettant de déterminer le nombre de grains de riz sur chaque case de l'échiquier.
Cela se fait facilement à l'aide d'une boucle :

```
echiquier = []
for n in range(64):
    echiquier.append(2**n)
```

Il est cependant possible de condenser cette écriture en une seule ligne.
Passons pour ça par une traduction littérale de ce qui est fait :
« `echiquier` est une liste des puissances de 2 (écrites 2^n) pour n allant de 0 à 64 ».
Les éléments de cette phrase se traduisent simplement en python :

- « pour n allant de 0 à 64 » : .....

– « puissances de 2 » : .....

– « liste » : les crochets indiquent que nous construisons une liste

– « `echiquier` est » : .....

On peut ainsi écrire .....

Remarque : Pour avoir un ordre d'idée de la quantité de grains de riz nécessaires pour la récompense promise, il suffit d'utiliser la syntaxe `sommeListe(echiquier,len(echiquier))`.
Cela nous donne plus de 18 milliards de milliards de grains de riz ce qui représente environ 1000 fois la production mondiale actuelle de riz réalisée en un an...

Exemple 2 : Construction d'une liste sous une certaine condition
On cherche à construire la liste des multiples de 7 inférieurs à 500. Cela peut se faire par une boucle :

```
multiples7 = []
for n in range(500):
    if n\%7 == 0 :
        multiples7.append(n)
```

Une nouvelle fois traduisons ceci de manière littérale :
« `multiples7` est une liste des entiers n pour n allant 0 à 499 si n est un multiple de 7 ».
Traduisons les termes de cette phrase en python :

– « si n est un multiple de 7 » : .....

– « pour n allant de 0 à 499 » : .....

– « liste » : les crochets indiquent que nous construisons une liste

– « `multiples7` est » : .....

On peut ainsi écrire .....

On peut aussi construire une telle liste avec un « si ... alors ... sinon » mais l'ordre est inversé.
Par exemple en itérant sur une chaîne de caractères :

```
texte="Je suis en NSI pour apprendre la programmation en Python"
voyelles="aeiouyàéèêùAEIOUYÀÉÉÊÙ"
lst = [e.lower() if e in voyelles else e.upper() for e in texte]
```

En français cela se traduit par « la liste des lettres en minuscules si ces lettres sont des voyelles, en majuscules sinon pour chaque lettre de la chaîne texte ».

Remarque : Ce résultat peut sembler non satisfaisant car on préférerait obtenir une chaîne de caractères et non une liste.
Cette conversion peut cependant se faire très rapidement en Python : `"".join(lst)` (on peut même remplacer `lst` par sa valeur pour se limiter à une seule ligne).

Exemple 3 : **Construire une liste avec deux itérables**
Cette construction se fait le plus naturellement par des boucles imbriquées :

```
lst = []
for i in range(4):
    for j in range(3):
        lst.append(i+j)
```

Quelle liste est ainsi construite ?
`[0,1,2,1,2,3,2,3,4,3,4,5]`

Ces quatre lignes peuvent être contractées en une seule en compréhension :

.....

b) Synthèse

Une boucle simple est construite sous cette forme :

```
for <the_element> in <the_iterable>:
   <the_expression>
```

La structure de la liste en compréhension est la suivante :

```
[<the_expression> for <the_element> in <the_iterable>]
```

Syntaxe avec condition :

```
for <the_element> in <the_iterable>:
   if <the_condition>:
     <the_expression>
```

En compréhension, cela s'écrit :

```
[<the_expression> for <the_element> in <the_iterable> if <the_condition>]
```

Toujours avec une condition, mais cette fois-ci avec un else :

```
for <the_element> in <the_iterable>:
   if <the_condition>:
     <the_expression>
   else :
     <other_expression>
```

En compréhension, cela s'écrit :

```
[<the_expression> if <the_condition> else <other_expression> for <the_element> in <the_iterable>]
```

Exercice : Construire en compréhension les listes suivantes :

1. la liste des carrés des nombres entiers allant de -5 à 6 :

   .....

2. la liste des images de nombres présents dans la liste `lst1` par une fonction `f` définie auparavant :

   .....

3. créer une liste `lstnb` de 20 valeurs choisies aléatoirement entre 10 et 30 ;

   .....

   .....

4. en se servant de la liste `lstnb` précédente, créer en compréhension une liste qui contienne uniquement les valeurs supérieures à 20 de cette liste.

   .....

5. une chaîne de caractères `chaine` étant donnée, on veut récupérer la liste des voyelles ;

   .....

6. écrire la liste des diviseurs d'un entier naturel `n` :

   .....

7. une liste `lst1` de nombre entiers étant donnée, on modifie les éléments de cette liste de la manière suivante :

   - on divise par 2 les nombres pairs ;
   - on ajoute 1 aux nombres impairs.

   .....