PROGRAMMATION : TYPES CONSTRUITS

1. Tableaux dynamiques/listes.

Sous Python, une liste est une variable contenant plusieurs variables, représentées par une suite d'éléments separés par des virgules, le tout étant entre crochets.

Vocabulaire : le terme officiel pour ce type est « tableau dynamique ». En Python on parle de liste, d'où l'usage des deux termes.

Ex :

```
>>> matieres =['maths', 'callof', 'svt', 1800, 20.357, 'physique',
'cuisine']
>>> print(matieres)
['maths', 'callof', 'svt', 1800, 20.357, 'physique', 'cuisine']
>>> listeVide = []
>>> listeDeZeros = [0]*5
>>> print(listeDeZeros)
[0,0,0,0,0]
```

Vous pouvez constater que les éléments (items) peuvent être de type distinct. On travaillera cette année sur des tableaux constitués d'éléments du même type, en général des tableaux de nombres.

Une liste/tableau est indexée (ou indicée). C'est à dire que l'on accède à un item précis d'une liste comme suit (attention la numérotation commence à 0) :

```
>>> print(matiere[0])
maths
```

Si on veut les éléments de la deuxième à la quatrième position, on utilisera une « tranche » (slice) :

```
>>> print(matieres[1:4])
   ['philo', 'svt', 1800]
```

Dans l'instruction précédente, remarquez que, comme pour `range`, l'extraction des index se fait avec un intervalle fermé à gauche et ouvert à droite.

Des indexs négatifs peuvent aussi être utilisés, -1 désignant le dernier item de la suite.

```
>>> print(matieres[-3 :])  # idem : print(matieres[-3 :-1])
[20.357, 'physique', 'cuisine']
```

Remarque : attention à ce qui semble être un manque de cohérence dans les intervalle d'extraction d'index. `[1:4]` récupére les index `1`, `2` et `3`, alors que `[-4 :-1]` récupère `-4, -3 et -2` . Pensez que les « slices » fonctionnent comme l'instruction `range` ; `[a : b]` tout comme `range(a , b)` contient le nombre `a`, alors que `b` est exclu.

La longueur d'une liste est donnée par len(*nom_de_la_liste*) :

```
>>> len(matieres)            # renvoie 7
```

On peut supprimer un ou plusieurs élément(s) d'une liste avec del() :

```
>>> del(matieres[3:5])
>>> print(matieres)
['maths', 'philo', 'svt', 'physique', 'cuisine']
```

Enfin, on peut ajouter un item en fin de liste avec la *méthode* append. Attention, ça n'est pas une fonction, la syntaxe est différente. En effet, on n'affecte pas le résultat dans une variable. Une *méthode* modifie directement l'*objet* sur lequel elle s'applique.

```
>>> matieres.append('starcraft')
>>> print(matieres)
      ['maths', 'philo', 'svt', 'physique', 'cuisine', 'starcraft']
```

Copie de tableau.

Comme vu dans le notebook sur les listes, l'instruction `liste2 = liste1` copie dans la variable `liste2` le contenu de la variable `liste1`, qui est l'adresse mémoire de la liste (plus précisément, un pointeur). Toute modification de `liste1` entraîne une modification de `liste2`, et vice-versa.

On peut copier une liste en faisant :

- `liste2 = liste1[ :]`
- `liste2 = liste1.copy()`

Méthodes sur les listes :

On n'utilisera pas toutes ces méthodes dès le début du cours. En effet de nombreux exercices de musculation du cerveau d'informaticien demandent de faire ce que font ces méthodes... sans les utiliser bien sûr !

- `liste.append(truc)` : ajoute l'élément unique `truc` à la fin de la liste
- `liste1.extend(liste2)` : rajoute en fin de `liste1` la `liste2`. Equivaut à faire `liste1 = liste1 + liste2`
- `liste.min()` : donne le minimum d'une liste
- `liste.max()` : devinez…
- `liste.sort()` : trie une liste dans l'ordre croissant (cf. doc pour l'ordre décroissant et les options)
- `liste.remove(valeur)` : supprime la première occurrence de `valeur` dans la liste
- `liste.insert(index, valeur)` : insère `valeur` à l'indice `index`
- `liste.reverse()` : inverse les éléments de la liste
- `liste.count(valeur)` : compte le nombre d'occurrences de `valeur` dans la liste (à récupérer dans une variable)
- `liste.index(valeur)` : donne l'index de la première occurrence de `valeur` dans la liste (à récupérer dans une variable)
- Il y a d'autres méthodes sur les listes.

L'instruction « in ».

L'instruction in permet de tester si une valeur est dans une liste ou non, et aussi de boucler sur une liste.

```
>>> 'maths' in matieres
True
>>> for item in matieres :
      print(item)
# renvoie :
'maths'
...
'starcraft'
```

2. Construction avancée des tableaux.

a Tableaux et matrices.

Un tableau est une liste de listes.

Une matrice est un tableau de nombres. Toutes les lignes doivent avoir la même longueur. Les matrices sont très utilisées en mathématiques

Exemple : une matrice à 3 lignes et 4 colonnes, c'est-à-dire de *dimension* 3×4 , est notée sous la forme :

$$A=\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

L'élément $a_{i,j}$ est à l'intersection de la ligne d'indice *i* et de la colonne d'indice *j*. Pour des raisons de compatibilité avec la programmation en Python, dans ce cours on a débuté les indices à 0. En mathématiques les indices commencent à 1.

Pour accéder à un élément, on met d'abord l'index de ligne puis celui de colonne :

```
>>> mat3x2 = [[1,2],[3,4],[5,6]]
>>>for i in range(len(mat)):
       print (mat[i])

       [1, 2]
       [3, 4]
       [5, 6]
```

```
>>> print(mat3x2[1][0])
3
```

b Construction d'un tableau par compréhension.

On peut construire un tableau plus rapidement que de partir d'un tableau vide, ou rempli de 0, et de le compléter élément par élément. Pour cela, on insère une boucle « `for` » à l'intérieur de la construction du tableau.

Exemples :

Nombres pairs de 0 à 20 :

```
>>> [i for i in range(0,21,2)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Carrés des multiples de 3 compris entre 0 et 21 :

```
>>> [j**2 for j in range(0,22,3)]
[0, 9, 36, 81, 144, 225, 324, 441]
```

Comprendre ce que donne celui-ci :

```
>>> [i*3-1 for i in range(50) if (i*3-1)%4 == 0]
[8, 20, 32, 44, 56, 68, 80, 92, 104, 116, 128, 140]
```

3. Quelques compléments sur les chaînes de caractères.

Ce paragraphe est un complément auquel on peut se référencer, il n'est pas indispensable de le connaître par cœur.

Comme précisé dans le cours Un chaîne de caractère est une liste de caractères, d'un type particulier. En particulier elles ne sont pas modifiables par accès d'index. Par contre certaines des fonctions et méthodes précédentes sont donc utilisables.

Ex :

```
>>> profession = "informaticien"
>>> profession[0] = "I"     # donne une erreur
       Traceback (most recent call last):
         File "<pyshell#1>", line 1, in <module>
           profession[0] ="I"
       TypeError: 'str' object does not support item assignment

>>> profession = "I"+ profession[1:]      # ne donne pas d'erreur
>>> profession
'Informaticien'
>>>
```

On peut comparer des chaînes (plus grand, plus petit, etc.) ; un exemple (à ne pas suivre ☺) ci dessous :

```
>>> "informatique" >= "jeux videos"
False
```

Il existe de nombreuses méthodes spécifiques aux chaînes de caractère. En particulier, on mentionnera :

- `chaine.split()` qui permet de découper une châine de caractères. On choisit le séparateur, par défaut c'est l'espace : on obtient alors tous les mots de la châine. On verra dans le chapitre sur les données qu'un autre séparateur peut être utile (souvent la virgule pour les fichier .csv). On précise le séparateur entre les parenthèses.
- `"séparateur".join(liste)` fait l'inverse du `split`. Il crée une chaîne à partir d'une liste de chaînes. On peut là aussi préciser le séparateur.
- `chaine.find(sousChaine)` cherche la position d'une sous-chaîne à l'intérieur d'une chaîne.
- `chaine.index(caractère)` cherche la position d'un caractère à l'intérieur d'une chaîne.
- `chaine.count(sousChaine)` compte le nombre d'occurences d'une sous-chaîne à l'intérieur d'une chaîne

On ne les détaillera pas toutes les méthodes ici ; sachez qu'il est possible de jouer sur les majuscules/minuscules.

Il y a également des méthodes de formatage de chaîne de caractères :

```
>>> chaine = "moyenne de nsi = {}"
>>> note1 = 15
>>> note2 = 22
>>> print(chaine.format((note1+note2)/2))
```

```
moyenne de nsi = 18.5
```

4. Tuples ou p-uplets.

Les tuples (nom anglais utilisé par Python) ou p-uplets (nom français et utilisé en mathématiques) sont des variables qui mélangent propriéts des tableaux et propriétés des chaînes de caractères.

L'exemple le plus simple est le couple (2-uplet) de coordonnées d'un point A dans le plan $(x_A;y_A)$, exemple que vous pouvez très bien imaginer prolongé dans l'espace en triplet (3-uplet) de coordonnées $(x_A;y_A;z_A)$. Il est clair qu'un point dans le plan n'a que deux coordonnées, ce qui explique qu'on puisse plus difficilement modifier un tuple qu'une liste. On utilise les tuples lorsque l'on souhaite que éviter les modifications, notamment par erreur.

Les tuples sont notés comme suit :

```
>>> monTuple = ('a' , 'b', 'c', 'd', 'e')        # ici un quintuplet de
                                                 # caractères
>>> eleveDateNaissance = ('Hubert' , '31', '02', '2001') # avec des types
                                                         # différents
```

On ne peut pas modifier les éléments d'un tuple :

```
>>> eleveDateNaissance[0] = 'Batman'
      Traceback (most recent call last):
        File "<pyshell#13>", line 1, in <module>
          eleveDateNaissance[0] = 'Batman'
      TypeError: 'tuple' object does not support item assignment
```

L'intérêt des tuples peut se voir par exemple dans la création d'une base de données élèves, que l'on pourra alors trier facilement :

```
>>> eleves = [('Batman' , '31', '02', '2001'), \
              ('Maitre Yoda' , '01', '14', '1459'), \
              ('Mickey' , '10', '10', '1910'),…]
```

Les tuples sont utilisés à chaque fois que l'on rentre des données qui ne seront pas modifiées (exemple : robot pour chat dont la fonctionnalité est de faire des jeux de mots).

5. Dictionnaires.

Les dictionnaires sont une forme Python du type de variable appelé p-uplet nommé (le dictionnaire est plus « puissant » que le p-uplet nommé). Contrairement à une liste ou à un tuple, un dictionnaire n'est pas ordonné. Un des avantages des dictionnaires est que la recherche y est extrêmement rapide, elle utilise ce que l'on appelle une « table de hachage ».

Un dictionnaire associe une valeur, qu'elle soit numérique ou chaîne de caractères, à une clé. Un dictionnaire s'écrit entre accolades. On rajoute des éléments dans un dictionnaire, éventuellement vide, avec la syntaxe `d[clé] = valeur`. Comme pour les autres types construits, la fonction `len(dictionnaire)` donne le nombre d'éléments. Par contre, lors de l'affichage, il n'y a aucun ordre spécifique, l'ordre est arbitraire.

Exemple :

```
>>> antagoniste = {"Batman" : "Joker" , "Gandalf" : "Sauron" , "Harry
Potter" : "Voldemort"}
      # Les clés sont les « gentils », les valeurs sont les « méchants ».
>>>antagoniste[Luke] = Palpatine # rajouter un élément
     >>>antagoniste
{"Gandalf" : "Sauron" , "Harry Potter" : "Voldemort" , "Luke" :
"Palpatine", "Batman" : "Joker"}
>>> for cle in antagoniste :
print ("valeur : ",antagoniste[cle]," associée à la clé : ",cle)
valeur : Voldemort associée à la clé : Harry Potter
valeur : Palpatine associée à la clé : Luke
```

Quelques fonctions et méthodes sur les dictionnaires :

- `len(dictionnaire)` renvoie le nombre d'éléments dans le dictionnaire
- `dictionnaire.keys()` renvoie une séquence avec toutes les clés. A combiner avec un tuple

ou une liste :

```
>>> gentils = list(antagonistes.keys())
>>> gentils
["Batman" , "Gandalf" , "Luke" , "Harry Potter"}
```

- `dictionnaire.values()` renvoie une séquence avec toutes les valeurs. A combiner avec un tuple ou une liste :

```
>>> mechants = tuple(antagonistes.keys())
>>> mechants
("Voldemort , "Joker" , "Sauron" , "Palpatine")
```

- `dictionnaire.items()` renvoie une séquence des tuples (*clé*, *valeur*).
- `dictionnaire.get(clé)` renvoie la valeur associée à *clé*, et `None` si la clé n'existe pas. On peut choisir de renvoyer une valeur particulière si la clé n'existe pas ; onprécise cette valeur en deuxième paramètre de la méthode.
- `dictionnaire[clé]` fait la même chose, mais renvoie une erreur si la clé n'existe pas.
- `dictionnaire.pop(clé)` supprime l'élément associé à la clé, et renvoie cette valeur (ce qui permet de la récupérer au moment de la suppression) :

```
>>> empereur = antagoniste.pop("Luke")# suppression de l'item
>>>empereur
"Palpatine"
```

Analogies avec les listes

Comme pour les listes, la copie d'un dictionnaire par `dico2 = dico1` copie l'**adresse** du dictionnaire et non son contenu ; dans ce cas toute modification sur `dico1` entraîne une modification sur `dico2`, et vice-versa.

La copie du contenu se fait avec la méthode `copy() : dico2 = dico1.copy()`

L'instruction `in` fonctionne de la même manière : `for cle in dico…`

Différents types de clés

Les clés peuvent être de n'importe quel type, par exemple on peut avoir des coordonnées avec des tuples.

Exemple : positions de départ sur un échiquier

```
positionEchec = {(0,0):"tour_blanc" , (0,1):"cavalier_blanc" , …}
```

EXERCICES TYPES CONSTRUITS

LISTES

Pour tous ces exercices, sont autorisés liste.append(truc), len(liste), et aucune autre méthode/fonction sur les listes sauf précision dans l'énoncé. Pas d'indices négatifs, pas de « élément in liste ».

Ex 1. Écrire une fonction `singleton(liste)` qui prend une liste en paramètres et renvoie `Vrai` si la liste est égale à `[0]`.

Ex 2. Écrire une fonction `même_longueur(liste1, liste2)` qui prend deux listes en arguments et renvoie vrai si les listes sont de même longueur.

Ex 3. Écrire une fonction `appartient(élément, liste)` qui prend en entrée une liste et une variable du même type que les éléments de la liste, et renvoie `Vrai` si l'élément appartient à la liste.

Ex 4. Ecrire une fonction `nb_occurrences(élément, liste)` qui renvoie le nombre d'occurrences de *élément* dans *liste*.

Ex 5. Écrire une fonction `bégaye(liste)` qui prend une liste en entrée, et renvoie en sortie une liste où tous les éléments sont doublés.

Ex 6. Écrire une fonction `croissante(liste)` qui admet comme paramètre une liste et retourne `Vrai` si la liste est ordonnée croissante.

Ex 7. Ecrire une fonction `max(liste)` qui renvoie le maximum d'une liste. Bien sûr, utiliser la fonction `max()` c'est tricher !

Ex 8. Ecrire une fonction `mix(liste1, liste2)` qui renvoie une seule liste ordonnée composée des éléments des deux listes entrée, liestes qui sont triées (on a le droit d'utiliser la méthode `sort()` pour les listes d'entrée uniquement, par pour construire la liste de sortie).

Ex 9.

1. Ecrire une première fonction `echange(liste, i, j)` qui échange les éléments d'indices *i* et *j* dans *liste.*
2. Ecrire une fonction `symétrie(liste)` qui échange le premier et le dernier élément de *liste*, ainsi que le deuxième et l'avant-dernier etc. On pourra utiliser la fonction du 1.
3. Mélange de Knuth.
   a Créer un tableau d'entiers aléatoires ordonné. On a le droit d'utiliser la méthode `liste.sort()`.
   b On veut désordonner le tableau… et oui cela arrive ! Pour cela on parcourt le tableau de la gauche vers la droite, et on échange chaque élément d'indice *i* avec un élément d'indice inférieur ou égal (entre 0 et *i* inclus). Ecrire cette fonction `melange(liste)`.

Ex 10.

1. Créer un tableau de 10 nombres aléatoires (ou plus) compris entre 1 et 10.
   Le trier ; on utilisera la méthode `valeurs_en_vrac.sort()` . Il est inutile de faire une affectation, recopiez juste cette instruction.
   À partir de ce tableau, en créer un autre qui contiendra les valeurs présentes dans le tableau trié, mais uniquement en un seul exemplaire. On peut utiliser la méthode `liste.sort()`.
   Il est très fortement conseillé d'utiliser la méthode `tableau_valeurs.append(bidule)` pour rajouter « `bidule` » en fin de tableau.
   On affichera les 3 tableaux
2. On reprend le programme précédent en le complétant. Le tableau initial contiendra 50 valeurs.
   Comme précédemment, on le trie puis on crée cette fois-ci deux tableaux :
   - Le tableau des valeurs présentes `tableau_valeurs` ;
   - Le tableau des effectifs correspondants à ces valeurs `tableau_effectifs` .

   On peut ensuite calculer la moyenne (remarque : on pouvait le faire dès le début, ce n'est pas le but de cet exercice !)

Ex 11. Écrire un programme qui stocke la décomposition en facteurs premiers d'un nombre entier strictement positif dans un tableau et ensuite affiche les éléments de ce tableau sous la forme 18 = 2*3*3.

Ex 12. Écrire un programme qui affecte des valeurs aléatoires comprises entre 13 et 50 (inclus) à un tableau de 10 entiers, trie le tableau par ordre croissant et l'affiche. Bien sûr, il est interdit d'utiliser la méthode `liste.sort()` !

Ex 13. Reproduction des lapins

Un couple de lapins a sa première portée à 2 mois, puis une portée tous les mois. Chaque portée est un couple de lapins. Tous les couples ainsi obtenus se reproduisent de la même manière.

1. On distinguera le nombre de couples de nouveaux lapins nouveaux, le nombre de couples de lapins ayant un mois, un_mois, et le nombre de couples de lapins « vieux » ayant 2 mois ou plus, vieux. Calculer « à la main » le nombre de couples de lapins de chaque type, ainsi que leur nombre total, pour les 10 premiers mois.
2. Écrire un algorithme nb_lapins calculant le nombre de lapins obtenus au bout de nb_mois mois à partir de nb_couples couples jeunes, et renvoyant le résultat.
3. Écrire un algorithme lapins_un_milliard calculant au bout de combien de temps les lapins sont plus d'un milliard (on supposera qu'aucun lapin ne meurt pendant cette période), en partant d'un couple de lapins.
4. Mettre en œuvre ces algorithmes dans un programme Python.

LISTES DE LISTES = TABLEAUX (dans le sens usuel en français) = MATRICES
*Pour tous ces exercices, la **seule et unique** méthode autorisée est append, sauf précision dans l'énoncé*

Ex 14. Créer une matrice 10×10 donnant toutes les tables de multiplication.

Ex 15.

1. Créer une matrice 10x10 de nombres aléatoires entre 1 et 1000.
2. Ecrire une fonction donnant le minimum des maximums de chaque ligne.
3. Complément : reprendre les questions précédentes, avec une matrice composée de mots aléatoires. On se contentera de lettres minuscules non accentuées, les « mots » étant des suites de lettres aléatoires.

Ex 16. Construction de matrices par compréhension : utilisation d'un « for » à l'intérieur d'une création de liste, cf. paragraphe 2b.

1. Construire en une seule instruction la liste `[-5, -1, 3, 7, 11, 15, 19, 23, 27, 31]`.
2. Construire en une seule instruction la liste `[0, 2, 8, 18, 32, 50, 72, 98, 128, 162]` (petite énigme ☺).
3. Construire en une seule instruction une matrice 10×10 composée de 10 fois la ligne `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`.
4. Construire en une seule instruction un tableau pyramidal de 10 lignes, où la première liste est `[0]`, la deuxième `[0,1]`, etc.

Ex 17. Le but de cet exercice est de créer une matrice 4×10 donnant, pour l'indice *i* de ligne entre 0 et 3, les 10 premieres valeurs de j^3, où *j* commence à -20, et progresse par intervalles de longueur *i* + 2.

Exemple : la première ligne est composée des cubes de -20, -18, etc… :

```
listeTordue[0] =[-8000, -5832, -4096, -2744, -1728, -1000, -512, -216, -64, -8]
```

Les questions suivantes peuvent donner un ordre de progression, mais il n'est pas obligatoire de les suivre.

1. Construire cette matrice à partir d'une liste vide, ou d'une matrice de zéros.
2. Construire cette matrice ligne à ligne, « par compréhension ».
3. ☠ Il doit être possible de construire cette matrice en une seule ligne (assez incompréhensible… on évite en général les syntaxes illisibles, mais ici on est dans le cadre d'un exercice). Vous pouvez essayer…

Ex 18. Le même exercice que le précédent, en plus dur ! Le but de cet exercice est de créer une matrice 4×10 donnant, pour l'indice *i* de ligne entre 0 et 3, les 10 premieres valeurs de j^3, où *j* est un multiple de $i+2$ commençant à -20 au plus petit.

Exemple : la première ligne est identique à celle de l'exercice précédent, par contre la deuxième commence ce différement :

```
listeTordueVersion1[1][ :2] =[-8000, -4913] # exercice précédent
listeTordueVersion2[1][ :2] =[-5832, -3375] # cet exercice -5832 = -18**3
```

Ex 19. Triangle de Pascal.

Le triangle de Pascal permet de calculer les coefficients du développement de $(a+b)^n$. Il a la forme suivante :

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1

La colonne de gauche et la diagonale ne contiennent que des 1. L'élément $x_{i,j}$ à la ligne *i* et la colonne *j* vaut : $x_{i,j} = x_{i-1,j-1} + x_{i-1,j}$.

On peut, à partir de la cinquième ligne du triangle ci-dessus, déduire :

$(a+b)^4 = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^3$

Écrire une fonction Python prenant comme paramètre un entier *n* qui renvoie le triangle de Pascal jusqu'à la ligne d'indice *n* incluse.

Le programme principal :

- Soit affiche la ligne correspondante du triangle de Pascal (version « facile »)
- Soit affiche le développement de $(a+b)^n$ (version « à peine plus difficile »). Cette version permet de travailler un peu sur les chaînes. On pourra écrire a^2b^3 pour a^2b^3.

On utilisera une fonction suivant() prenant comme paramètres les deux dernières lignes calculées, effectuant le calcul d'une nouvelle ligne, et mettant à jour les paramètres pour que ce soient toujours les deux dernières ligne calculées. Une deuxième fonction fera l'affichage. On commencera par réfléchir aux structures de données

Ex 20. Enigme ☠☠☠

La résolution de cette difficile énigme permet de montrer qu'on peut **numéroter toutes les fractions** (positives ici, mais si on rajoute les négatives c'est à peine plus dur). En effet une fraction $\frac{p}{q}$ peut s'écrire sous la forme d'un couple (p,q). Ainsi $2=(2,1)$, $\frac{3}{4}=(3,4)$. Mais aussi $2=(4,2)$ et $(5,0)$ ne correspond à aucune fraction (division par 0). La conséquence fondamentale et très surprenante est que l'on peut numéroter toutes les fractions, ce qui veut dire **qu'il n'y a pas plus de fractions que d'entiers** (il y en a autant en fait).

Construire une fonction qui permet de ranger les entiers jusqu'à un entier N donné, dans une matrice sous la forme :

$$Entiers = \begin{bmatrix} 0 & \rightarrow & 1 & & 5 & \rightarrow & 6 \\ & \swarrow & & \nearrow & & \swarrow & \\ 2 & & 4 & & 7 & & \\ \downarrow & \nearrow & & \swarrow & & & \\ 3 & & 8 & & & & \\ & \swarrow & & \iddots & & & \\ 9 & \rightarrow & 10 & & & & \end{bmatrix}$$

Les flèches ne sont pas forcément une indication. Il y a plusieurs méthodes, dont certaines utilisent l'ordre indiqué par les flèches et d'autres non.

DICTIONNAIRES & TUPLES

Ex 21. On donne un tuple ("clé1", "clé2", "clé3", … , "valeur1", "valeur2", "valeur3", …), de longueur paire . Créer à partir de ce tuple un dictionnaire {"clé1" : "valeur1" , "clé2" : "valeur2" , …}

Ex 22. Créer un dictionnaire comportant 3 items. Chaque item a pour clé un mot en français, et pour valeur sa traduction en verlan (ou en anglais). Créer une fonction qui permet d'ajouter un item uniquement si la clé n'est pas déjà présente dans le dictionnaire.

Ex 23. Créer une fonction qui prend en entrée deux dictionnaires dic1 et dic2, et renvoie dic1 comme fusion des deux dictionnaires, et dic2 en dictionnaire vide.

Ex 24. On donne un dictionnaire dont les valeurs sont des listes de lettres. Créer une fonction qui, à partir de ce dictionnaire, renvoie toutes les combinaisons possibles formées avec une lettre de chaque item.

Exemple :

```
dictionnaire = {'1':['a','b'], '2':['c','d']}
resultat = ["ac","ad","bc","bd"] # valeur renvoyee
```

Ex 25. Ecrire une fonction qui donne les 3 plus grandes clés d'un dictionnaire. Idem avec les trois plus grandes valeurs. Pour chacune des fonctions, on renverra la valeur, respectivement la clé, correspondante.

Ex 26. Ecrire une fonction qui prend en entrée un dictionnaire et une valeur potentielle, et renvoie le nombre d'occurences de la valeur dans le dictionnaire.

Ex 27. Ecrire une fonction qui, à partir d'une chaîne de caractères (qui peut être aussi longue que l'on veut, comme une encyclopédie), renvoie un dictionnaire dont chaque item a pour clé un caractère et comme valeur le nombre d'occurrences de ce caractère.

Ex 28. Ecrire une fonction qui prend en entrée une liste de dictionnaires, ayant au moins une clé en commun, qui sera en paramètre d'entrée aussi, et renvoie le nombre de fois ou la valeur associée à cette clé est identique au troisième paramètre d'entrée.

Exemple :

```
listDico = [{'nom':'albert', 'bac':True, 'age':18}, {'nom':'berangere',
   'bac':True, 'age':14}, {'nom':'charlot', 'bac':False, 'age':81}]
cle = 'bac'
valeur = True
resultat = 2     # valeur renvoyée
```

Ex 29. Ecrire une fonction qui prend en entrée deux dictionnaires et une clé potentielle, et renvoie vrai si la clé est présente dans les deux dictionnaires.

Ex 30.

1. Ecrire une fonction qui crée un dictionnaire dont les clés sont les entiers entre 1 et 1000, et les valeurs les racines carrées de ces entiers.
2. Idem sauf que les clés sont les entiers multiples de 3 entre 1 et 1000.
3. Idem sauf que l'item est crée uniquement si les valeurs sont multiples de 13, les valeurs étant les carrés des clés + 1.

Ex 31. Ecrire une fonction qui affiche un dictionnaire par ordre ascendant/descendant des clés/valeurs. Un ou plusieurs paramètres préciséront si l'ordre est ascendant ou descendant, et si l'on souhaite se baser sur les clés ou les valeurs.

Ex 32. Ecrire une fonction qui échange clés et valeurs d'un dictionnaire. Que se passe-t-il si plusieurs valeurs sont identiques ?