1NSI	Langages et programmation Compléments	TP – **Compléments de programmation**

Objectifs :

- ⇨ Créer des tableaux par compréhension
- ⇨ Découvrir les slices

Nous allons compléter notre panoplie de programmeur avec des notions spécifiques à python et facilitant la programmation.

I – Les compréhensions de listes

On souhaite créer un tableau contenant les entiers pairs jusqu'à 100. Pour ce faire, on peut procéder de façon classique avec une boucle :

Version statique avec boucle for	Version par ajout avec boucle for	Version par ajout avec boucle while

`tab = [0]*(100//2 + 1)`
`for i in range(100//2 + 1):`
`    tab[i] = 2*i`
	`tab = []`
	`for i in range(100//2 + 1):`
	`    tab.append(2*i)`
		`tab = []`
		`i = 0`
		`while (2*i) <= 100:`
		`    tab.append(2*i)`
		`    i += 1`

Ces différentes versions prennent entre 3 et 5 lignes. Avec la compréhension de liste (ou liste en compréhension), il est possible de faire cela en une seule ligne :

```
tab = [2*i for i in range(100//2 + 1)]
```

On obtient ainsi un code plus compact, donc plus lisible et facilement maintenable[1].

1) Syntaxe

La syntaxe générale des compréhensions de liste est :

```
liste_generee = [expression for variable in iterable if condition]
```

Les crochets sont indispensables et servent à indiquer que cette instruction va générer une liste. De même les mots-clés `for` et `in` sont obligatoires. `iterable` peut être n'importe quel objet itérable comme une liste un générateur (range) ou même une chaîne de caractère. `valeur` est la variable qui prendra les différentes valeurs renvoyées par l'itérable et `expression` est une expression dépendant généralement de `variable` et qui donnera la valeur à ajouter à la liste générée pour chaque élément fournit par l'itérable.
Enfin la clause `if` est optionnelle. Elle permet de filtrer la liste générée.

Au final la syntaxe compacte de la compréhension de liste fait exactement la même chose que le code plus détaillé suivant :

```
liste_generee = []
for valeur in iterable:
    if condition:
        liste_generee.append(expression)
```

Question 1 :

1) Ecrire la compréhension de liste permettant de créer une liste `carres` contenant les 12 premiers carrés de nombres entiers.
2) Ecrire la compréhension de liste permettant de créer une liste `carres_liste` contenant les carrés des nombres d'un tableau `tab`. Par exemple si `tab = [2, 6, -3, 2.5, 18]` alors `carres_liste` vaudra `[4, 36, 9, 6.25, 324]`.

[1] Il est aussi plus rapide à l'exécution.

2) Filtrage

La compréhension de liste peut également être utilisée pour filtrer un tableau ou une liste existante.

Question 2 :
Ecrire la compréhension de liste permettant de filtrer un tableau `tab` en éliminant tous les éléments négatifs ou nuls. Par exemple si `tab` vaut `[9, -3, 0, 5, 7, -18, 3]`, le tableau filtré vaudra `[9, 5, 7, 3]`.

3) Application d'une fonction

Les listes en compréhension sont aussi pratiques pour appliquer une fonction à tous[2] les éléments d'un tableau.

Question 3 :
Ecrire la compréhension de liste permettant de créer un tableau `tab_entier` à partir d'un tableau `tab` en convertissant tous ses éléments en entier (avec la fonction `int`). Par exemple le tableau `[1, 2.5, 3.14, 4.2, -7, 9.3]` donnera `[1, 2, 3, 4, -7, 9]`.

4) Boucles imbriquées

Il est également possible d'imbriquer les boucles à l'intérieur d'une compréhension.

Par exemple : `p = [a + str(n) for a in "AB" for n in [1,2,3]]`

va générer le tableau : `['A1', 'A2', 'A3', 'B1', 'B2', 'B3']`.

Question 4 :
1) Ecrire une compréhension de liste permettant de créer facilement le tableau ci-contre.

```
['Haut G','Haut Centre','Haut D',
'Milieu G','Milieu Centre', 'Milieu D',
'Bas G', 'Bas Centre', 'Bas D']
```

2) Ecrire la compréhension de liste permettant de générer une grille de morpion vierge (grille de 3x3 cases contenant le caractère `' '` (espace) dans toutes les cases). Vérifiez que l'on a bien 9 cases indépendantes.

5) Compréhension de dictionnaires

Il est également possible d'utiliser la compréhension avec les ensembles et les dictionnaires. Pour cela on remplace simplement les crochets `[]` par des accolades `{}` et dans le cas des dictionnaires, on spécifie la clé et la valeur en les séparant par « `:` ».

Exemples :

```
lettres = {chr(n + ord('A')) for n in range(26)}
```

va donner : `{'T', 'X', 'U', 'Y', 'P', 'W', 'E', 'N', 'J', 'Q', 'R', 'M', 'G', 'B', 'A', 'K', 'D', 'O', 'I', 'V', 'Z', 'S', 'L', 'F', 'H', 'C'}`

On remarque bien que les éléments de l'ensemble ne sont pas ordonnés mais qu'on a bien les 26 lettres de l'alphabet.

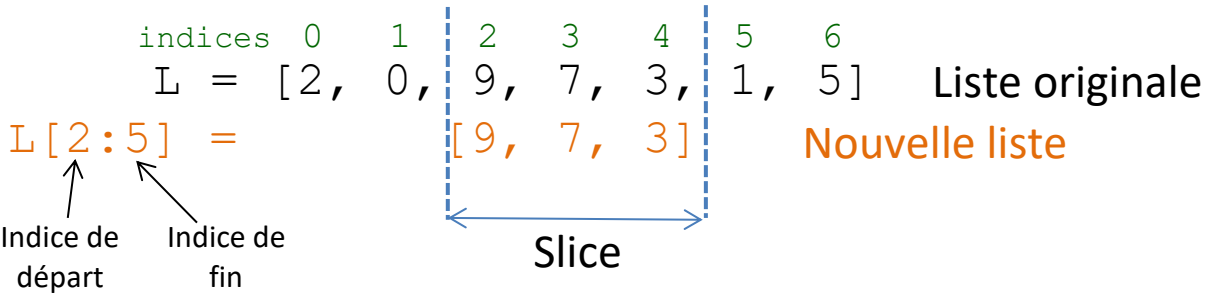
```
codes = [('A', 'o'), ('B', 'K'), ('C', '$'), ('D', 'R'), ('E', '4'), ('F', '(')]
dict_codage = {l:c for l,c in codes}
```

Donne : `{'A': 'o', 'B': 'K', 'C': '$', 'D': 'R', 'E': '4', 'F': '('}`

[2] Ou seulement une partie en utilisant la clause `if`.

II – Les slices

Les slices de python permettent de faire très facilement référence à une *partie* d'une collection ordonnée (principalement des listes ou des chaînes de caractères).
Les slices fonctionnent de manière très similaire au générateur `range` que nous avons déjà vu. On précise l'indice de départ et l'indice de fin. On obtient un sous-ensemble de la collection de départ contenant tous les éléments compris entre l'indice de départ et l'indice juste avant celui de fin.

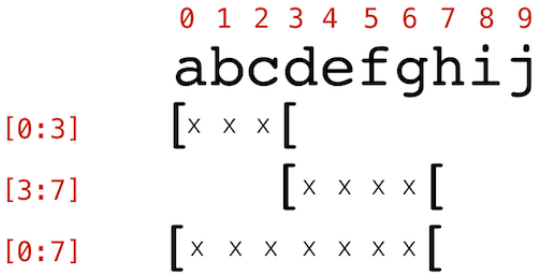


Le *slicing* crée une copie superficielle de la liste qui peut être utilisée et modifiée comme n'importe quelle liste sans modifier l'original.
Vous pouvez exécuter le code ci-contre dans python tutor pour voir comment python manipule les slices.

```
L = [2, 0, 9, 7, 3, 1, 5]
l2 = L[2:5]
l2[1] = 8
print(L)
print(l2)
```

Remarques :

- Il est possible d'omettre l'indice de début dans le slice. Dans ce cas python part du début de la séquence (`L[:4]` est équivalent à `L[0:4]` si on parcourt vers les indices croissants).
- De même, on peut omettre l'indice de fin. Dans ce dernier cas, python prend les éléments jusqu'au dernier de la séquence (inclus) : `L[4:]` est équivalent à `L[4:len(L)]` si on parcourt vers les indices croissants.
- On peut utiliser des indices négatifs (qui partent de la fin) comme on l'a vu sur les tableaux.



- Le slicing fonctionne sur toutes les collections ordonnées, donc notamment sur les **chaînes de caractères**.
- Comme avec `range` on peut, optionnellement donner un troisième argument (après un deuxième caractère « `:` ») qui indique le pas (on prend un élément de la liste d'origine tous les *pas* éléments). Par exemple si `liste = [0,1,2,3,4,5,6,7,8,9]`, `liste[2::2]` donnera `[2, 4, 6, 8]` et `liste[5:1:-1]` donnera `[5, 4, 3, 2]`.

Question 5 :

```
l = "Onomatopée"
```

1) Sans utiliser python, remplir la deuxième colonne du tableau ci-contre.
2) Utilisez la console python pour contrôler vos réponses à la question précédente. Corriger si nécessaire dans la troisième colonne.
3) En utilisant les slices, donner une méthode permettant d' inverser une chaîne de caractères (`"abcd"` devient `"dcba"`).
4) Avec une compréhension de liste, trouver une méthode pour inverser l'ordre des éléments d'une liste.

	Votre réponse	Réponse correcte
`l[2:6]`		
`l[:3]`		
`l[5:]`		
`l[4:4]`		
`l[-5:7]`		
`l[-2:-4]`		
`l[7:18]`		
`l[1:9:3]`		
`l[7:2:-1]`		