Langages de script (Python)
CMTP n° 2 : Autres types sequences : tuples et listes

I) Listes et tuples

Les chaînes de caractères appartiennent à une famille très particulière des itérables, appelé en Python les *séquences*. La caractéristique des séquences est la possibilité à accéder à ses éléments avec les indexes et les slices : https://docs.python.org/3/library/stdtypes.html#typesseq D'autres exemples de séquences sont les tuples et les listes :

```
>>> #tuples:
>>> t = (1, 2, 'a', 3.2)
>>> t
(1, 2, 'a', 3.2)
>>> t[2]                              #acces par indices
'a'
>>> t[:2]                             #utilisation des slices
(1, 2)
>>> t[::2]
(1, 'a')
>>> type((1)),type((1,))              #attention a' tuple singleton
(<class 'int'>, <class 'tuple'>)
>>> t+('b','c')                       #concatenantion
(1, 2, 'a', 3.2, 'b', 'c')
>>> t*3
(1, 2, 'a', 3.2, 1, 2, 'a', 3.2, 1, 2, 'a', 3.2)
>>> x, y, w, z = t                    #unpacking
>>> x
1
>>> #listes, trois moyens pour les introduire
>>> l1 = [0, 2, 4, 6, 8]              #explicite
>>> l2 = [2 * x for x in range(5)]    #par *comprehension*
>>> l2
[0, 2, 4, 6, 8]
>>> l3 = list(range(0, 10, 2))        #par le constructor list(iterable)
>>> l3
[0, 2, 4, 6, 8]
>>> l1[0]                             #acces par indices
0
>>> l1[:3]                            #utilisation des slices
[0, 2, 4]
>>> l1 + [9, 10]                      #concatenantion
[0, 2, 4, 6, 8, 9, 10]
>>> l1 * 2
[0, 2, 4, 6, 8, 0, 2, 4, 6, 8]
>>> l1 + ['1', 3.0]                   #diff avec OCaml, membres heterogenes
[0, 2, 4, 6, 8, '1', 3.0]
>>> ll = [l1, [4, 3, 2, 1, 3]]        #on peut imbriquer les listes (comme les tuples)
>>> ll
[[10, 2, 4, 6, 8], [4, 3, 2, 1, 3]]
```

Exercice 1 :

Écrire une fonction qui prend en paramètre une liste d'entiers à deux dimensions (rectangulaire) et calcule le nombre d'entrées intérieures de la matrice dont tous les voisins sont strictement plus petits. Chaque

entrée intérieure de la matrice a quatre voisins (à gauche, à droite, vers le haut, vers le bas). Par exemple, pour la matrice

```
1 4 9 1 4
4 8 1 2 5
4 1 3 4 6
5 0 4 7 6
2 4 9 1 5
```

représentée par la liste `[[1,4,9,1,4],[4,8,1,2,5],[4,1,3,4,6],[5,0,4,7,6],[2,4,9,1,5]]`, la fonction devrait renvoyer `2` car il y a deux éléments de la matrice (`8` et `7`) qui ont uniquement des voisins plus petits.

Exercice 2 :

1. Écrire une fonction `inversion` qui demande à l'utilisateur de saisir une ligne de texte et renvoie une liste des mots dans l'ordre inverse (on supposera sans faire de vérification que l'utilisateur ne saisit que des lettres ou des espaces). *Indication : penser à la méthode* ***`split`*** *du type* ***`str`***.
2. Modifier la fonction `inversion` afin qu'elle retourne la chaîne de caractères composée des mots, dans l'ordre inverse, de la ligne de texte entrée par l'utilisateur. *Indication : penser à la méthode* ***`join`*** *du type* ***`str`***.

II) Les listes en compréhension

Une liste en compréhension consiste à placer entre crochets une expression suivie par une clause `for` puis par zéro ou plus clauses `for` ou `if`. Le résultat est une nouvelle liste résultat de l'évaluation de l'expression dans le contexte des clauses `for` et `if` qui la suivent. Par exemples :

```
>>> [x for x in [1, 2, 3, 4, 5]]
[1, 2, 3, 4, 5]
>>> [x for x in [1, 2, 3, 4, 5] if x % 2 != 0]
[1, 3, 5]
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Exercice 3 : en une ligne !

Grâce à une liste en compréhension, écrire les fonctions suivantes en utilisant une seule ligne de code pour le corps de la fonction :

1. `multiple(n)` donnant la liste des multiples de 5 ou 7 inférieurs à `n` ;
2. `multiple_idx(l)` la sous-liste des éléments multiples de 5 ou 7 de la liste `l` donnée en paramètre :
3. `zipping(l, m)` donnant la liste des couples `[(l1, m1), (l2, m2), ...]` étant `l`, `m` deux listes de même longueur `[l1, l2, ...]` et `[m1, m2, ...]`.
   (`zipping` est une simplification de la fonction `zip` native en Python).
4. Une matrice d'entiers est une liste de listes d'entiers (voir Exercice 1). Écrire une fonction `transpose(m)` qui calcule la transposée de la matrice `m` (en supposant `m` pas vide). Par exemple, l'évaluation de `transpose([[1,2,3],[4,5,6]])` renvoie `[[1,4],[2,5],[3,6]]`.
5. `tens(n)` la liste des listes des premières `n` dizaines :

   `[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [10, 11, 12, 13, 14, 15, 16, 17, 18, 19], ....]`
6. `flatten(ll)` retourne une liste contenant chaque élément de chaque liste de ll. Par exemple `flatten([[1,2], [3,4,5], [-6]]) == [1,2,3,4,5,-6]`

Exercice 4 : listes de str

Dans cet exercice, on demande encore d'utiliser la compréhension des listes pour écrire des programmes concises et lisibles.

1. Écrire une fonction `without_e` qui prend en argument une liste de mots et retourne la liste de ceux qui ne contiennent pas la lettre `'e'`. *Indications : vous pouvez utiliser l'opérateur* `in` *ou sa variante* `not in`.
2. Écrire une fonction `anti_begue` qui prend en argument une chaîne de caractères et en crée une nouvelle dans laquelle ont été supprimés les mots consécutifs identiques. La fonction devra transformer le texte d'origine en liste de mots avant tout traitement et transformer la liste nouvellement obtenue en chaîne de caractères. On remarque que si on zippe la liste des mots de la phrase avec la liste des mots décalée de 1, les mots répétés se retrouvent ensemble.

```
>>> begue
['ceci', 'est', 'un', 'un', 'test']
>>> begue_decale
['est', 'un', 'un', 'test']
>>> list(zip(begue, begue_decale))
[('ceci', 'est'), ('est', 'un'), ('un', 'un'), ('un', 'test')]
```

   Il suffit alors de choisir les bons mots (attention au dernier mot de la phrase. . .).

Exercice 5 : mots sans cube

On considère les mots sur un alphabet à deux lettres $\{a, b\}$. On dit qu'un mot (c'est-à-dire une suite de lettres) possède un cube s'il possède un facteur de la forme uuu où u est un mot non vide. Par exemple *baaa* et *bababa* possèdent un cube alors que *bababb* est sans cube. On peut montrer qu'il existe un mot infini sans cube (mais ce n'est pas l'objet de cet exercice). Nous allons être plus modestes et écrire une fonction qui teste si un mot est sans cube.

1. Écrire une fonction `prefixes(w, long_max)` qui renvoie la liste des préfixes du mot `w` de taille inférieure à l'entier `long_max` passé en argument.
2. Écrire une fonction `no_cube_prefixes(w)` qui prend en argument une chaîne de caractères et détermine si elle possède un cube comme préfixe. *Indication : pour tester si un mot possède un préfixe cube, on va regarder si un mot possède un préfixe dont le cube est aussi un préfixe de ce mot.*
3. Écrire une fonction `no_cube(w)` qui prend en argument une chaîne de caractères `w` et détermine si elle est sans cube. *Indication : il suffit tester les suffixes de* `w` *grâce à la fonction précédente.*

III) Mutables vs immutables

Une différence fondamentale entre les listes et les tuples comme les chaînes de caractères est que ces deux dernières sont immutables, c.-à-d. elles ne permettent pas de changer la valeur de ces éléments, alors que les listes de Python (au contraire que celles de OCaml par exemple) sont mutables.

```
>>> l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[1] = "toto"
>>> l
[0, 'toto', 2, 3, 4, 5, 6, 7, 8, 9]
>>> s = "0123456789"
>>> s[1] = "toto"
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    s[1] = "toto"
TypeError: 'str' object does not support item assignment
>>> t = (0,1,2,3,4,5,6,7,8,9)
>>> t[1] = "toto"
```

```
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    t[1] = "toto"
TypeError: 'tuple' object does not support item assignment
```

Le statut mutable des listes permet une plus large palette d'opérations natives que sur les autres types de séquences immutables : https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types. Attention à la différence entre les fonctions qui génèrent des nouvelles listes et celles qui modifient les listes passées en paramètre (dans ce dernier cas, normalement avec la notation pointée) :

```
>>> l = list(range(5))
>>> l
[0, 1, 2, 3, 4]
>>> new = l + [5]
>>> new, l
([0, 1, 2, 3, 4, 5], [0, 1, 2, 3, 4])
>>> no_new = l.append(5)
>>> no_new, l
(None, [0, 1, 2, 3, 4, 5])
```

L'utilisation des listes mutables permet de mettre en œuvre plusieurs structures de données intéressantes comme les piles et les queues, grâce à des fonctions comme `pop` et `append`.

Exercice 6 : bon parenthésage

Écrire une fonction `is_balanced(s)` vérifiant si une chaîne de caractères `s` est correctement parenthésée par rapports aux parenthèses rondes (`"("` et `")"`) et carrées (`"["` et `"]"`). À chaque parenthèse ouvrante doit correspondre une parenthèse fermante du même type, et réciproquement. Par ailleurs, si une parenthèse ouvrante est ouverte à l'intérieur d'un autre couple de parenthèses, sa parenthèse fermante doit elle aussi de trouver à l'intérieur du même couple.
Par exemple, `"([aa](bb[]))"` est correctement parenthésé, alors que `"([aa)](bb[])"` ou `"([aa](bb[])"` ne les sont pas.
Indication : utilisez une pile (implémentée par une liste de Python) pour stocker les types de parenthèses ouvrantes rencontrées. Lorsqu'on rencontre une parenthèse fermante, il faut s'assurer qu'il reste dans la pile une parenthèse ouvrante à fermer, et que son type correspond à celui de la parenthèse fermante. À la fin du parcours de la chaîne de caractères `s` la pile sera vide si et seulement si `s` était correctement parenthésée.