STRUCTURES DE DONNEES

I) Petit rappel

On a déjà rencontré des types composés dans le § 4 sur les expressions : nous avions vu les tuples, les chaînes de caractères et les listes.

Nous avions vu pour ces différents types comment les obtenir, comment accéder à leur longueur, à un de leurs éléments, voire à une sous-structure, et également comment on peut les concaténer.

Seulement nous étions restés à un stade assez simple puisque, par exemple, nous pouvions écrire explicitement tous les éléments d'une liste car sa longueur était à la fois petite et connue. Il n'est évidemment pas envisageable de créer par ce moyen ne serait-ce que la liste des 100 premiers entiers.

II) Chaines de caractères

Création

Outre la possibilité d'écrire directement la chaine de caractères directement ou en transformant en chaîne de caractères un des autres types composés lorsque c'est possible (avec **str**), une possibilité est de créer de façon itérative en utilisant le processus de concaténation en allant chercher le suffixe à ajouter dans une certaine variable ou à l'aide d'une fonction.

Par exemple, la séquence d'instruction suivante

```
alphabetmin = ""
for k in range (97,123):
    alphabetmin += chr(k)
```

permet d'obtenir la chaine de caractères 'abcdefghijklmnopqrstuvwxyz' (chr(i) étant le i-ème caractère Unicode. Les majuscules étant obtenues pour i entre 65 et 90, et les chiffres entre 48 et 57)

Accès à un caractère, à une sous-chaine

On considère une chaine de caractères s. Pour les exemples nous travaillerons avec
s = 'Les consonnes sont bcdfghjklmnpqrstvwxz et les voyelles sont aeiouy'

- s[i] permet d'obtenir le caractère d'indice i (sachant que l'on numérote à partir de 0)
  Par exemple, la séquence d'instructions suivante construit la chaine de caractères select

  ```
  select = ""
  for k in range(len(s)):
      if k % 10 == 0 or k % 7 == 2:        select += s[k]
  ```

  select vaut alors 'Lsnnncgpxeeeln u'
- Remarquons que s[-1], s[-2], s[-3]… désigne respectivement les dernier, avant-dernier, antépénultième… caractères de la chaine s

- On peut obtenir une sous-chaîne par la syntaxe s[i:j]. s[i:j] est la sous-chaîne des caractères successifs entre le numéro i (inclus) et le numéro j (exclus). Si le premier indice est absent, on part de 0 ; si le second indice est absent, on arrive jusqu'au dernier. Par exemple s[7:20] est la chaine 'sonnes sont b'
- La technique de sélection précédente se complète en ajoutant un certain pas. s[i:j:k] est la sous-chaîne des caractères pris de k en k entre le numéro i (inclus) et le numéro j (exclus). Cette technique est appelé la technique de **slicing**.
  Par exemple s[7:50:9] est la chaîne 'snjtl' et s[27:2:-3] est 'lhd osnn '

On rappelle pour finir que l'on ne peut pas changer un caractère d'une chaîne.

III) Tuples

Il s'agit du type Python associé aux n-uplets. Dans certains langages, le type est appelé séquence.

Création

Outre la possibilité d'écrire directement le tuple directement ou en transformant en tuple un des autres types composés lorsque c'est possible (avec **tuple**), une possibilité est de créer de façon itérative en utilisant le processus de concaténation en allant chercher le suffixe à ajouter dans une certaine variable ou à l'aide d'une fonction.

Par exemple, la séquence d'instruction suivante

```
carre = ( )
for k in range (2,21,2):
    carre += (k**2, )
```

permet d'obtenir le n-uplet (4, 16, 36, 64, 100, 144, 196, 256, 324, 400)

Accès à un élément, à un sous-tuple

On considère un tuple s. Pour les exemples nous travaillerons avec
s = (4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 'a', 'b', 'c', 'd', 'e', 'ghij', 'klm')

- s[i] permet d'obtenir l'élément d'indice i (sachant que l'on numérote à partir de 0)
  Par exemple, la séquence d'instructions suivante construit le tuple select

```
select = ( )
for k in range(len(s)):
    if k % 10 == 0 or k % 7 == 2:          select += (s[k],)
```

  select vaut alors (4, 36, 400, 'a', 'klm')
- Remarquons que s[-1], s[-2], s[-3]… désigne respectivement les dernier, avant-dernier, antépénultième… éléments du tuple s
- On peut obtenir un sous-tuple par la syntaxe s[i:j]. s[i:j] est le sous-tuple des éléments successifs entre le numéro i (inclus) et le numéro j (exclus). Si le premier indice est

absent, on part de 0 ; si le second indice est absent, on arrive jusqu'au dernier. Par exemple s[7:20] est le tuple (256, 324, 400, 'a', 'b' , 'c', 'd', 'e', 'ghij', 'klm')

- La technique de sélection précédente se complète en ajoutant un certain pas. s[i:j:k] est le sous-tuple des éléments pris de k en k entre le numéro i (inclus) et le numéro j (exclus). Cette technique est appelé la technique de **slicing**.
  Par exemple s[2:20:3] est la chaîne (36, 144, 324, 'b', 'e')

On rappelle pour finir que l'on ne peut pas changer un élément d'un tuple.

IV) Listes

Il s'agit du type Python associé aux n-uplets modifiables. Ils correspondent a priori à des tableaux à une dimension constitués d'objets pouvant être de nature différente.

Création

Outre la possibilité d'écrire directement la liste directement ou en transformant en liste un des autres types composés lorsque c'est possible (avec **list**), une possibilité est de créer de façon itérative en utilisant le processus de concaténation en allant chercher le suffixe à ajouter dans une certaine variable ou à l'aide d'une fonction.

Par exemple, la séquence d'instruction suivante

```
cube = [ ]
for k in range (2,21,3):
    cube += [k**3]
```

permet d'obtenir la liste [8, 125, 512, 1331, 2744, 4913, 8000]

Accès à un élément, à une sous-liste

On considère une liste L1. Pour les exemples nous travaillerons avec
L1 = [8, 125, 512, 1331, 2744, 4913, 8000, 'a', [4, 36, 400]]

- s[i] permet d'obtenir l'élément d'indice i (sachant que l'on numérote à partir de 0)
  Par exemple, la séquence d'instructions suivante construit la liste select

```
select = [ ]
for k in range(len(L1)):
    if k % 5 == 0 or k % 3 == 2:          select += [L1[k]]
```

select vaut alors [8, 512, 4913, [4, 36, 400]]

- Remarquons que L1[-1], L1[-2], L1[-3]… désigne respectivement les dernier, avant-dernier, antépénultième… éléments de la liste L1
- On peut obtenir une sous-liste par la syntaxe L1[i:j]. L1[i:j] est la sous-liste des éléments successifs entre le numéro i (inclus) et le numéro j (exclus). Si le premier indice est absent, on part de 0 ; si le second indice est absent, on arrive jusqu'au dernier.
  Par exemple L1[5:] est la liste [4913, 8000, 'a', [4, 36, 400]]

- La technique de sélection précédente se complète en ajoutant un certain pas. L1[i:j:k] est la sous-liste des éléments pris de k en k entre le numéro i (inclus) et le numéro j (exclus). Cette technique est appelé la technique de **slicing**.
- Dans la liste L1, on voit que le dernier élément est lui-même une liste. L1[8] est la liste [4, 36, 400] et L1[8][1] est l'élément d'indice 1 de la liste L1[8].

Manipulations avancées

Copie

Il semble un peu curieux de placer cette action parmi les manipulations avancées mais on constate un comportement inhabituel lors d'une affectation L1 = L0 si L0 est une liste donnée.

En effet, si on considère la séquence d'instructions suivante:

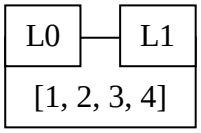
```
L0 = [1, 2, 3, 4]
L1 = L0
L1[2] = 8
```

alors, à la fin de ces différentes affectations, L1 **et L0** ont pour valeur [1, 2, 8, 4]

En fait on peut un peu mieux comprendre ce qui se passe en utilisant la fonction id qui donne l'adresse mémoire où est stockée la variable.

Pour des variables prenant une valeur de types simples, l'adresse change lorsque la valeur de la variable change. Par contre pour une liste, l'adresse ne change pas lorsque l'on change un élément de cette liste.

Aussi dans l'affectation L1 = L0, tout se passe comme si Python avait juste copié l'adresse de L0 dans L1. Or cette adresse ne contient pas la liste en elle-même mais des informations du style "L0 est une liste pour laquelle vous trouverez les éléments en suivant tel cheminement"… Et donc à la fin de l'affectation, l'état des variables est le suivant :



: la liste [1, 2, 3, 4] porte deux noms L0 et L1 : l'affectation a juste donné un alias à la liste L0

Une façon pour éviter cette copie d'adresse est d'utiliser la syntaxe suivante :

```
L0 = [1, 2, 3, 4]
L1 = []
for x in L1: L1 += [x]
```

Cela marche bien pour des listes dont les éléments sont de types simples et un changement sur un des éléments de l'une des deux listes n'affecte pas l'autre liste.

Malheureusement, ce n'est toujours pas satisfaisant lorsque les éléments de L0 sont eux-mêmes des listes. On a une copie superficielle (de niveau 1).

Par exemple, les instructions de gauche donneront les valeurs des variables suivantes pour L0 et L1 :

Instruction	L0	L1	explication
L0 = [1, 2, [3, 4]]	[1, 2, [3, 4]]	non affectée	
L1 = []	[1, 2, [3, 4]]	[]	
for x in L1: L1 += [x]	[1, 2, [3, 4]]	[1, 2, [3, 4]]	
L0[0] = 8	[8, 2, [3, 4]]	[1, 2, [3, 4]]	L'élément de rang 0 étant de type simple, le changement sur L0 n'affecte pas L1
L0[2][0] = 9	[8, 2, [9, 4]]	[1, 2, [9, 4]]	L'élément de rang 2 dans L1 et L0 est une liste, c'est une copie d'adresse qui est faite pour L1[2]
L0[2] = 5	[8, 2, 5]	[1, 2, [9, 4]]	La copie étant de niveau 1, le changement d'affectation de l'élément de rang 2 de L0 n'affecte pas L1

Pour ne pas avoir de problème d'alias, il faudrait analyser le type de chaque élément de la liste à copier et en faire une copie complète…

Dans la plupart des cas que l'on utilisera cette année, on pourra le faire simplement car nous aurons souvent des listes d'éléments homogènes.

A noter que le module **copy** possède une fonction qui fait directement cette analyse (de façon récursive) et effectue une copie en profondeur de la liste : il s'agit de la fonction **deepcopy**

Ajout d'un élément

Si L est une liste et x un élément on veut ajouter l'élément x à la liste L. Il y a non seulement plusieurs façons de le faire mais aussi plusieurs façons de comprendre la question.

La façon la plus naturelle est l'ajout à la fin de la liste.

- On peut vouloir utiliser le processus de concaténation :
  L += [x] fait effectivement cet ajout en bout de liste

Syntaxiquement, cela semble assez simple à comprendre. Par contre algorithmiquement, ce n'est pas très optimal . Evidemment pour une liste constituée de quelques éléments ce n'est pas bien grave. Par contre pour une liste très longue, cet ajout à un coût non négligeable en temps. En effet, cela demande la création d'une liste contenant le résultat de la concaténation et l'affectation de la liste à un nouveau nom de variable (même si au bout du compte ce sera le même) et donc de l'affectation des éléments à L[0], L[1]…..

La programmation Python étant "orientée objet" , sa caractéristique est de créer des classes d'objets (list, tuple, window (fenêtre graphique),…) et d'associer à ces objets des méthodes (des fonctions particulières) qui agissent sur l'objet lui-même. Ici une méthode est parfaitement adaptée

- la méthode **append**
  L.**append**(x)

Par exemple, si L possède un million de termes, l'ajout d'un terme par append va 4 fois plus vite que par concaténation, et l'ajout successif de 100 termes ne change pas le temps d'exécution avec append alors qu'il est logiquement multiplié par 100 pour la concaténation

Par contre, pour ajouter l'élément x en début de liste, L = [x] + L fait bien ce que l'on veut mais pas append…

Suppression d'un élément

Python possède également des méthodes pour supprimer un élément d'une liste.

- On peut vouloir enlever le dernier élément. Dans ce cas la sélection de la sous-liste **L[:-1]** répond bien à la question mais la liste L reste entière : il faut procéder à l'affectation L = L[:-1] pour supprimer le dernier élément de la liste L

- On peut vouloir supprimer le terme de rang i dans la liste L. On utilise alors la fonction **del** : **del**(L[i]) supprime de la liste L l'élément de rang i

- La méthode pop fait à peu près la même chose :
  L.**pop**(i) donne l'élément L[i] et supprime dans la liste L le terme de rang i

- On peut également vouloir supprimer de la liste un terme donné. La méthode **remove** répond à la question… mais elle n'enlève que la première itération de l'élément voulu de la liste.

Insertion d'un ou plusieurs éléments

Python possède en fait un processus complet pour remplacer une sous-liste par une autre liste.

Ainsi l'instruction : L[i:j] = L2 remplace la sous-liste L[i:j] par la liste L2

On en déduit des nouveaux procédés pour ajouter un élément dans une liste, supprimer un élément dans une liste ou insérer de nouveaux éléments dans une liste…

V) Tableaux

La plupart des langages de programmation permettent de définir des tableaux à une, deux, trois, … n dimensions. Souvent ils sont obtenus par le type (ou la fonction) array, ou matrix dans des modules plus explicitement adaptés aux mathématiques.

En fait on peut considérer les listes de nombres comme des tableaux de dimension 1, les listes de listes de nombres comme des tableaux de dimension 2, les listes de listes de listes de nombres comme des tableaux de dimension 3 et plus généralement, les listes de tableaux de dimensions n comme des tableaux de dimensions n+1. A chaque étape, si on veut un tableau complet, il faut que le nombre d'éléments dans une liste d'un niveau donné soit le même.

Par exemple tab = [[1,2,3],[4,5,6],[7,8,9],[10,11,12]] est une définition du tableau bidimensionnel $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$

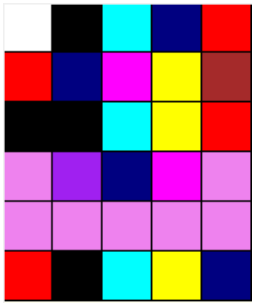
On obtient l'élément de ligne i et de colonne j par la syntaxe tab[i][j]

Si on affecte à une case une couleur correspondant à la valeur de cette case, on obtient une image : c'est le principe de l'image bitmap.

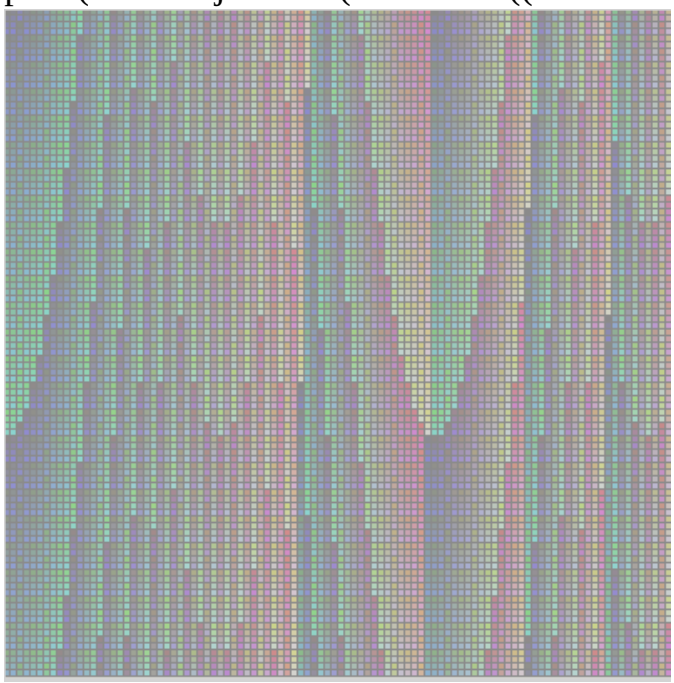
Par exemple si on prend la liste des couleurs

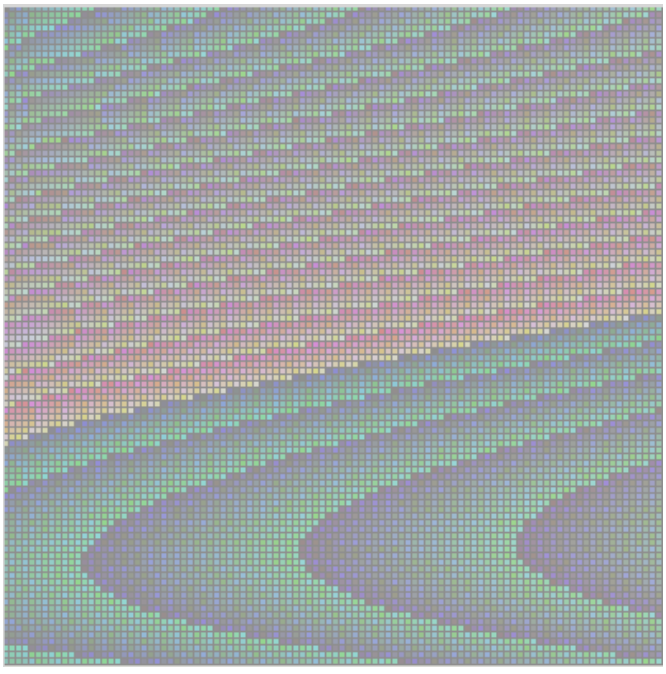
couleur = ['white', 'black', 'red', 'green', 'blue', 'cyan', 'yellow',
'magenta', 'navy' , 'purple', 'brown', 'violet', 'white']

et le tableau $\begin{pmatrix} 0 & 1 & 5 & 8 & 2 \\ 2 & 8 & 7 & 6 & 10 \\ 1 & 1 & 5 & 6 & 2 \\ 11 & 9 & 8 & 7 & 11 \\ 11 & 11 & 11 & 11 & 11 \\ 2 & 1 & 5 & 6 & 8 \end{pmatrix}$ on obtient l'image



En prenant une liste de 4096 couleurs (16 * 16 * 16 en rgb) on peut obtenir les images suivantes lorsque les cases (i,j) sont remplies par les fonctions suivantes
(4 i + 2 j**2) % 4096 sur un tableau 100 * 100 avec 4096 couleurs
puis (5*i + 8*j + floor(4100 *sin ((2*i**1.3+3*j**0.6)/400))) %4096





VI) Exercices

1) Ecrire une fonction **compte_espace** recevant comme argument une chaine de caractères **phrase** et qui détermine le nombre d'espaces inclus dans cette phrase.

2) Ecrire une fonction **compte_maj** recevant comme argument une chaine de caractères **phrase** et qui détermine le nombre de majuscules incluses dans cette phrase.

3) Ecrire une fonction **transpose** recevant comme argument un tuple **t** et deux entiers **a** et **b** et qui renvoie le tuple obtenu à partir de t en échangeant les éléments d'indice a et b.

4) Ecrire une fonction **remplacepar10** recevant comme argument une liste **L1** et une liste d'entiers strictement croissante **L2** et qui renvoie la liste obtenue à partir de L1 en changeant les éléments des L1 d'indice dans L2 par 10.

5) Ecrire une fonction **intercale** recevant comme argument une liste **L** et un nombre **a** et qui renvoie la liste obtenue à partir de L en intercalant la valeur a entre chaque élément de L

6) Ecrire une fonction **intercalebis** recevant comme argument une liste **L** et deux nombres **a** et **b** et qui renvoie la liste obtenue à partir de L en intercalant la valeur a ou la valeur b entre chaque élément de L (une fois sur deux c'est a et l'autre fois c'est b)

7) Ecrire une fonction **compte_voyelle** recevant comme argument une chaine de caractères **phrase** et qui détermine le nombre de voyelles incluses dans cette phrase.

8) Ecrire une fonction **somme_liste** qui reçoit comme argument une liste **L** (qu'on suppose constituée de nombres) et qui renvoie la somme des éléments de L. Même chose avec la fonction **moyenne_liste** qui calcule la moyenne. Même chose avec **plus_grand** pour le plus grand élément. Même chose avec **variance_liste** pour le calcul de la variance (moyenne des carrés des écarts à la moyenne).

9) Ecrire une fonction **max_consecutifs** qui reçoit comme argument une liste **L** (qu'on suppose constituée de nombres) et qui renvoie la longueur de la plus grande sous-liste constituée de termes identiques ainsi que la valeur de ce terme.

10) Ecrire une fonction **compte_zeros_finaux** qui reçoit comme argument une liste **L** et qui renvoie le nombre de zéros qui termine la liste. Ecrire une fonction **enleve_zeros_finaux** qui reçoit comme argument une liste **L** et qui renvoie la liste obtenue à partir de L en enlevant les zéros finaux.

11) Ecrire une fonction **inversion_phrase** qui reçoit comme argument une chaine de caractères **phrase** et qui renvoie la chaîne des caractères écrits dans l'autre sens.

12) Soit P un polynôme dont les coefficients sont donnés par une liste **Pol**. Pour simplifier, on supposera que la liste associée au polynôme nul est la liste [0].
   a) Ecrire une fonction **evaluer** qui prend comme argument la liste **Pol** (associée au polynôme P) et un nombre **x** et qui revoie la valeur P(x)
   b) Ecrire une fonction **derive** qui prend comme argument une liste **Pol** (associée au polynôme P) et qui renvoie la liste des coefficients de P'
   c) Ecrire une fonction **integre** qui prend comme argument une liste **Pol** (associée au polynôme P) et qui renvoie la liste des coefficients de la primitive de P s'annulant en 1
   d) Ecrire une fonction **integrebis** qui prend comme argument une liste **Pol** (associée au polynôme P) et qui renvoie la liste des coefficients de la primitive de P dont l'intégrale entre 0 et 1 vaut 1.