Le cas particulier des listes

La semaine dernière, nous avons rappelé quelques rudiments du langage Python, et parmi les premiers exercices, vous avez pu remarquer que les listes de données jouent un rôle particulier. En effet, c'est un objet incontournable qu'il faudra apprendre à manipuler rapidement.

1 Trois présentations pratiques et des méthodes déjà implémentées

On a rappelé que les **listes** pouvaient être définies de trois façons :

1. en convertissant une variable de type *tuple* grâce à la commande `list`,
2. en complétant la liste au fur et à mesure dans un programme itératif avec une boucle `for` ou `while`,
3. en décrivant la séquence contenue dans la liste. On parle alors de **liste par compréhension** pour laquelle les éléments sont exprimés en fonction de l'indice associé :

```
In : L = [k**2 for k in range(0,11)]; L
```
$$[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$$
python

Et en plus, on pourra même y ajouter des **instructions conditionnelles** :

```
In : L = [k**2 for k in range(0,11) if k%2 == 0]; L
```
$$[0, 4, 16, 36, 64, 100]$$
python

Contrairement aux n-uplets, on pourra cette fois-ci en modifier le contenu et on fera bien entendu attention à l'indexation car les éléments sont toujours numérotés de 0 jusqu'à la longueur de la liste obtenue par la commande `len` -1 :

```
In : L = list(range(11)); L[5] = 0; print(L); len(L)
```
[0, 1, 2, 3, 4, 0, 6, 7, 8, 9, 10]
11
python

Si de plus, on cherche à supprimer un élément, on pourra toujours le faire au moyen de la commande `del` :

```
In : del(L[2:5]); L
```
$$[0, 1, 0, 6, 7, 8, 9, 10]$$
python

En fait, la donnée de deux indices sous la forme `i:j` nous permet d'extraire les éléments d'une liste, mais on veillera à bien comprendre que l'élément d'indice i est toujours inclus, alors que l'élément d'indice j est exclus.

De la même façon, on retiendra quelques **méthodes** pratiques qui vous ont déjà été présentées. Ce sont des fonctions qui opèrent sur la liste donnée :

commande Python	interprétation
`L.append(x)`	ajoute l'élément x à la fin de la liste L
`L1.extend(L2)`	ajoute à la fin de $L1$ les éléments de $L2$
`L.insert(i,x)`	insère au rang i l'élément x
`L.remove(x)`	supprime la première occurrence de x dans L
`L.reverse()`	permet de retourner la liste L en inversant les éléments
`L.sort()`	permet d'ordonner la liste L

Remarque Ces commandes existent, mais encore une fois, on ne vous demande pas de toutes les connaître mais plutôt de les reconstruire.

Exercice 1 (utilisation des listes pour la gestion des polynômes). [ ]
On considère le polynôme :

$$P(X) = a_0 + a_1 X + \ldots + a_n X^n$$

celui-ci pourra être représenter à l'aide d'une liste $L = [a_0, a_1, \ldots, a_n]$ de sorte que pour tout $z_0 \in \mathbb{C}$:

$$P(z_0) = \sum_{k=0}^{n} L[k] z_0^k$$

1. Dans le langage Python, construire le programme $evaluation(L : list, z_0 : complex) \longrightarrow complex$ qui, pour tout polynôme L donné sous la forme d'une liste et tout complexe z_0, renvoie l'image de z_0 par P.
2. On définit alors l'**algorithme de Hörner** par : $\begin{cases} u_0 = a_n \\ \forall\, k \in [\![1, n]\!], u_k = u_{k-1} z_0 + a_{n-k} \end{cases}$.
   (a) Justifier rapidement qu'à la n-ième étape, $u_n = P(z_0)$.
   (b) Dans le langage Python, construire le programme $horner(L : list, z_0 : complex) \longrightarrow complex$ qui, pour tout polynôme L donné sous la forme d'une liste et tout complexe z_0, renvoie l'image de z_0 en utilisant l'algorithme précédent.

Remarque On vient donc de présenter deux façons de calculer l'image d'un complexe par un polynôme. Mais si on détermine soigneusement le nombre d'opérations pour obtenir chacune de ces images, on montre que la méthode de Hörner est plus efficace. Ainsi, on retiendra que la **complexité en nombre d'opérations** pour un polynôme de degré n est donnée par :

$$C_{evaluation}(n) = O(n^2) \text{ et } C_{horner}(n) = O(n)$$

2 Applications

Exercice 2 (suite de Fibonacci). [ ]
On rappelle que la **suite de Fibonacci** est définie par :

$$\begin{cases} u_0 = u_1 = 1 \\ \forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n \end{cases}$$

1. Dans le langage Python, construire la fonction récursive $fibonacci(n : int) \to int$ qui renvoie la valeur de u_n.
2. On souhaite travailler à l'aide des listes. Dans le langage Python, construire la fonction itérative $fibonacci2(n : int) \to list$ qui renvoie la liste des valeurs de $[u_0, \ldots, u_n]$.

Exercice 3 (le groupe symétrique). [ ]
Pour $n \in \mathbb{N}^*$, on note S_n le groupe des permutations de l'ensemble $[\![0, n-1]\!]$. Une permutation de S_n sera représentée en Python par une liste, dont l'élément d'indice i est l'image de i par cette permutation.

Par exemple, si on note $\sigma \in S_4$ définie par :

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 3 & 1 & 0 & 2 \end{pmatrix}$$

alors on a $\sigma(0) = 3$, $\sigma(1) = 1$, $\sigma(2) = 0$ et $\sigma(3) = 2$. Ainsi, en Python, cette permutation sera décrite par la liste des images obtenues par σ : $[3, 1, 0, 2]$.

Dans tout l'exercice, on pourra utiliser librement les tests Python du type `x in L` (respectivement `x not in L`) permettant de vérifier si x est présent dans la liste L (respectivement de vérifier si x n'est pas présent dans la liste L).

1. Si s est une liste Python représentant une permutation de S_4, quelle instruction Python permet de trouver l'image de 1 par cette permutation ? Quelle liste Python représente la transposition $(2\ 3) \in S_4$?
2. Écrire une fonction Python $composition(s1 : list, s2 : list) \longrightarrow list$prenant en entrée deux listes représentant des permutations σ_1 et σ_2 du même groupe de permutations et renvoyant la liste représentant la permutation $\sigma_1 \circ \sigma_2$. *On fera attention à longueur des permutations données en entrée.*
3. Écrire une fonction Python $inv(s : list) \longrightarrow list$ prenant en entrée une liste représentant une permutation σ et renvoyant la liste représentant σ^{-1}.

Exercice 4 (recherche dichotomique dans une liste déjà triée). [ ]
On considère une liste L constituée de n nombres réels et on cherche à savoir si un nombre réel x appartient à la liste L.
Pour cela, on introduit deux curseurs représentant les indices qui encadreront ma recherche :

$$a = 0 \text{ et } b = \texttt{len(L)-1}$$

puis, on coupe la liste en deux avant de comparer x avec le terme du milieu $L[(a+b)//2]$.
La liste étant triée, on peut donc savoir si x est atteint, ou s'il se trouve éventuellement dans une des deux parties de la liste. On peut alors déplacer les curseurs et itérer le procédé tant que les bornes d'encadrement sont dans le bon sens.

Dans le langage Python, construire la fonction $chercherdicoto(L : list, x : float) \to bool$ qui renvoie `True` ou `False` en fonction de l'appartenance de x à la liste L.

Exercice 5 (des listes aux chaines de caractères). [ ]
On étend le travail effectué sur les listes aux chaines de caractère, car il s'agit encore d'un type de données structurées.
Par exemple, on peut encore identifier les caractères par leur place :

```
In : nom='chaimaa'; nom[0]; len(nom)
                                        'c'
                                        7
                                                                    python
```

Ici, on considère un texte donné et on souhaite relever les positions d'un mot dans ce texte.

Pour cela, on parcourt les caractères du texte et lorsque l'un d'entre eux coïncident avec la première lettre de notre mot, on teste les lettres suivantes afin de noter l'emplacement du mot si celui-ci appartient au texte. Concrètement, où est Charly ? on cherchera d'abord à identifier la position de la lettre c avant de comparer les suivantes au mot "charly" :

t	c	a	c	h	r	l	e	n	a	c	h	a	r	l	y	d	p	o

1. Dans le le langage Python, construire la fonction booléenne $presence(text : str, mot : str) \to bool$ qui renvoie si oui ou non le mot est présent dans la chaîne de caractère.
2. En déduire le programme $position(text : str, mot : str) \longrightarrow list$ qui teste si le mot est présent dans la chaîne, puis dans ce cas renvoie la liste des positions du mot *mot* dans le texte *text*.