Chapitre 3 : Séquences et type de données

Instructions introduites dans ce cours : `len,[],append,pop,break,type`.

La notion de *séquence* désigne en Python différents types d'objets qui partagent le même genre de structure (à peu de choses près il s'agit d'une structure de vecteur, au sens mathématique du terme), et sur lesquels on peut appliquer un certain nombre de fonctions communes. Nous allons introduire dans ce cours trois types de séquences : les listes, les tuples et les chaînes de caractères.

1 Les listes

1 A) Notions élémentaires

Sous Python une *liste* est une suite d'éléments séparés par des virgules, cette suite étant délimitée par des crochets. Intuitivement, on peut considérer qu'il s'agit de la représentation d'un vecteur en langage Python. Le nombre d'éléments d'une liste est appelé sa *longueur*, et on peut calculer celle-ci grâce à la primitive `len` (abréviation de l'anglais *length*), qui renvoie alors un entier. La liste `L` définie par `L=[ ]` ne contient aucun élément (sa longueur est donc nulle), et est appelée la *liste vide*.

Étant donnée une liste L de longueur n, on appelle *indice d'un élément de* L sa place (où son rang) dans la liste, en considérant que le rang du premier élément de L est 0 (et donc le rang du dernier est $n-1$). En Python, l'instruction `L[i]` permet d'accéder à l'élément d'indice i de L.

Exemple 1.1.

```
>>> L=[1,2,3,4,5]
>>> L
[1, 2, 3, 4, 5]
>>> len(L)
5
>>> L[0]
1
>>> L[1]
2
>>> L[4]
5
>>> L[5]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: list index out of range
```

Exemple 1.2. Écrire une fonction Python `remplace(L)` qui, à partir d'une liste L renvoie une liste vide si L est vide et remplace le dernier élément de la liste par 0 sinon.

Remarque 1.3.

La commande `in` peut s'utiliser sous la forme `a in L` et renvoie le booléen `True` si `a` est un élément de la liste et `False` sinon.

1 B) Parcours des termes d'une liste via leur indice

De nombreux algorithmes consistent à parcourir une liste. L'exemple suivant décrit l'un d'entre eux.

Exemple 1.4.
Cherchons à implémenter la fonction `nb_zero(L)` qui prend en entrée une liste `L`, et renvoie en sortie le nombre de termes nuls présents dans cette liste. Il suffit en effet :

- d'initialiser une variable `compteur` à `0`, qui contiendra au final le nombre de termes nuls de `L` ;
- pour calculer la valeur finale de `compteur`, on *parcourt* la liste `L` (c'est-à-dire que l'on regarde tous ses termes les uns après les autres, par exemple de gauche à droite), et dès que l'on rencontre un terme nul, on incrémente la valeur de `compteur` (c'est-à-dire que l'on utilise le code `compteur=compteur+1`) ;

Le programme ci-dessous est une implémentation de cet algorithme.

On obtient ainsi :

```
>>> nb_zero([1,0,0,2,3,0])
3
```

On voit sur cet exemple que pour parcourir la liste `L` on effectue une boucle for, en posant `n=len(L)` puis en faisant `for k in range(n)` : la variable `k` prendra successivement les valeurs `0,1,2,...,n-1` et donc `L[k]` prendra successivement les valeurs `L[0],L[1],L[2],...,L[n-1]`.

1 C) Ajout et suppression d'un élément dans une liste

En Python, l'instruction `L.append(x)` permet d'ajouter l'élément `x` à la fin de la liste `L`. Cette instruction modifie la liste `L`. En particulier, sa longueur est augmentée de 1. De manière analogue, l'instruction `L.pop()` permet de supprimer le dernier élément de la liste `L`, si la liste `L` n'est pas vide.

Exemple 1.5.

```
>>> L=[1,3,4]
>>> L
[1,3,4]
>>> L.append(2)
>>> L
[1,3,4,2]
```

```
>>> L.pop()
2
>>> L
[1,3,4]
```

On constate sur cet exemple que l'instruction `L.pop()` renvoie un résultat : l'élément qui vient d'être supprimé dans la liste. On peut donc affecter ce résultat à une variable `a`. L'instruction `a=L.pop()` effectue donc deux choses en même temps : elle modifie la liste `L` en lui enlevant son dernier élément, et elle affecte à la variable `a` la valeur de cet élément. Attention, l'instruction `L.append(x)` ne renvoie aucun résultat, elle se contente de modifier la liste.

1 D) Construction de listes à l'aide d'une boucle for

Supposons maintenant que l'on veuille construire une liste de la forme `[f(0),...,f(n-1)]` où f est une fonction déclarée au préalable, et n un entier. Bien entendu, on peut toujours entrer chacun des éléments de la liste un-à-un au clavier, mais si l'entier n est grand cela n'est pas possible. On peut accélérer la définition d'une telle liste en utilisant une boucle for. Nous allons pour cela donner deux méthodes différentes.

L'idée est de construire la liste terme par terme : on part d'une liste vide, et lors du k-ième passage dans la boucle on construit le k-ième terme de la liste (on dit que l'on effectue une construction par *couches* ou par *strates*).

Par exemple, la fonction déclarée ci-dessous prend en entrée un entier n et une fonction f, puis renvoie la liste `[f(0),...,f(n-1)]`.

```
def construction_liste(f,n):
    L=[]
    for k in range(n):
        L.append(f(k))
    return(L)
```

Exemple 1.6.

```
def g(x):
    return(x**2)
```

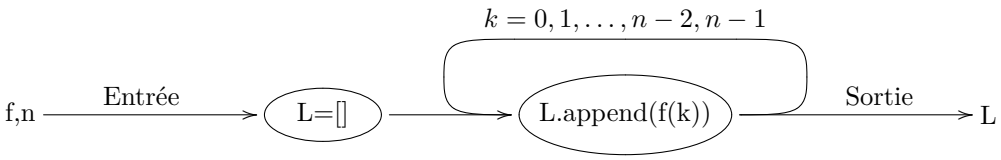
```
>>> construction_liste(g,5)
[0, 1, 4, 9, 16]
```

Il est important de comprendre ici que la ligne de code `L.append(f(k))` rajoute l'élément `f(k)` derrière tous les autres éléments de `L`. Les différentes étapes de l'évaluation de ce programme par la machine sont les suivantes :

- à l'étape 0 (initialisation), la liste est vide et donc de longueur 0 ;
- à l'étape 1, on a $k = 0$ et on rajoute `f(0)` à la fin de la liste, qui est alors de longueur 1 ;
- à l'étape 2, on a $k = 1$ et on rajoute `f(1)` à la fin de la liste, qui est alors de longueur 2 ;

  $\vdots$
- à l'étape n, on a $k = n-1$ et on rajoute `f(n-1)` à la fin de la liste, qui devient de longueur n ;

On peut également représenter l'évaluation par le graphe suivant :



Exemple 1.7.
Pour une liste `L`, l'instruction `sum(L)` renvoie la somme des éléments d'une liste `L`. Construire une fonction `somme_geom(q,n)` qui renvoie

$$\sum_{k=0}^{n} q^k$$

en utilisant l'instruction `sum`.

La seconde méthode consiste en fait à utiliser une syntaxe dédiée à ce problème, qui est la suivante :

```
L=[f(i) for i in range(n)]
```

Remarque 1.8.

Exemple 1.9.

```
def g(x):
    return(x**2)
L=[g(i) for i in range(5)]
```

```
>>> L
[0, 1, 4, 9, 16]
```

Sur cet exemple, la définition par compréhension est donc nettement plus rapide à implémenter. Néanmoins, dans des cas plus compliquées on ne peut pas utiliser la compréhension, et il est finalement nécessaire d'appliquer la première méthode.

1 E) Affectations parallèles

Grâce à la structure de liste, on peut effectuer plusieurs affectations en *parallèle* (encore qualifiées de *simultanées*). En essayant d'appliquer cette opération au problème de l'interversion (voir le TP n° 1), on voit que ces affectations ont bien lieu simultanément.

Exemple 1.10.

```
>>> [a,b,c]=[1,2,3]
>>> [a,b,c]
[1, 2, 3]
```

```
>>> [a,b]=[b,a]
>>> a
2
>>> b
1
```

1 F) Extraction d'une sous-liste

Il est possible d'extraire une sous-liste d'une liste (on parle de slicing en anglais), en précisant les indices des éléments que l'on souhaite extraire : la commande `L[i:j]` renvoie la liste `[L[i],...,L[j-1]]` (observez bien que le dernier terme est `L[j-1]` et non pas `L[j]` : on obtient donc une liste de longueur $j-i$). Notons que l'accès aux sous-listes se fait également en écriture : étant donné une liste `M`, l'instruction `L[i:j]=M` remplace la sous-liste de `L` par la liste `M`. Par conséquent, la liste `L` est modifiée. Remarquons enfin que si la liste `M` n'a pas pour longueur $j-i$, Python ne produit pas d'erreur : la longueur de la liste `L` est donc modifiée !

Exemple 1.11.

```
>>> L=[1,2,3,4,5]
>>> L[1:4]
[2, 3, 4]
>>> L[1:5]
[2, 3, 4, 5]
```

```
>>> L[1:4]=[7,0,-2]
>>> L
[1, 7, 0, -2, 5]
>>> L[2:4]=[8,12,-5]
>>> L
[1, 7, 8, 12, -5, 5]
```

1 G) Comparaison de deux listes

On peut tester l'égalité ou la différence de deux listes grâce aux primitives `==` et `!=`.

Exemple 1.12.

```
>>> [1,2]==[2,1]
False
>>> [1,2]==[1,2]
True
```

1 H) Concaténation

La *concaténation* de deux listes est l'opération consistant à mettre bout-à-bout les éléments de deux listes pour en créer une troisième. En Python, elle s'effectue grâce au symbole `+`.

Exemple 1.13.

```
>>> L=[1,4]
>>> M=[5,7]
>>> L+M
[1, 4, 5, 7]
```

1 I) Affectation et listes

L'affectation d'une liste à une variable ne crée pas une copie de la liste. Voici un exemple de code qui illustre ce phénomène :

```
>>> a =[1,2,4]
>>> b=a
>>> a[1] = 0
>>> b
[1,0,4]
```

On voit que la modification de la liste `a` affecte aussi la liste `b`. Pour faire une nouvelle copie de la liste, il est nécessaire d'utiliser une expression comme ci-dessous :

```
>>> a =[1,2,4]
>>> b=a[:]
>>> a[1] = 0
>>> b
[1,2,4]
```

Remarque 1.14.

2 Les tuples

Sous Python un *tuple* est une suite d'éléments séparés par des virgules, cette suite étant délimitée par des parenthèses. En termes syntaxiques, la seule chose qui différencie une liste d'un tuple c'est le fait que l'on utilise des parenthèses plutôt que des crochets (par exemple le tuple vide est donné par `()`).

La plupart des opérations valables sur les listes peuvent également s'effectuer sur les tuples, en utilisant les mêmes primitives : calcul de longueur, concaténation, affectation parallèle...

Exemple 2.1.

```
>>> s=(1,2)
>>> t=(3,4)
>>> len(s)
2
>>> s+t
(1, 2, 3, 4)
```

```
>>> (a,b)=s
>>> (a,b)
(1, 2)
>>> a
1
>>> b
2
```

Ce qui différencie principalement les tuples des listes d'un point de vue fonctionnel, c'est que l'accès aux termes d'un tuple peut se faire en lecture mais pas en écriture : concrètement, si `t` est un tuple, la commande

`t[0]` renvoie bien le premier terme de `t`, mais par contre la commande `t[0]=1` renvoie un message d'erreur, de même que `t.append(0)`. On peut donc lire les termes d'un tuple, mais pas les modifier. Pour cette raison, on qualifie les tuples d'*objets immutables*. On peut donc voir les tuples comme des objets « sécurisés ».

Exemple 2.2.

```
>>> t[0]                    >>> t[0]=2
3                           Traceback (most recent call last):
                              File "<console>", line 1, in <module>
                            TypeError: 'tuple' object does not support item assignment
```

Remarque 2.3.

3 Compléments sur les chaînes de caractères

Nous avions vu que sous Python une *chaîne de caractères* (ou *string* en anglais) est une suite de symboles du clavier délimitée par des guillemets " ou des apostrophes '.

Les chaînes de caractères se manipulent exactement comme les tuples d'un point de vue fonctionnel : on peut calculer leur longueur, effectuer des concaténations, des affectations parallèles, accéder aux termes d'une chaîne (c'est-à-dire à ses lettres) en lecture mais pas en écriture.

```
>>> x='bonjour'             >>> x[0]
>>> len(x)                  'b'
7                           >>> x[0]='a'
>>> y=' numero 6'           Traceback (most recent call last):
>>> x+y                       File "<console>", line 1, in <module>
'bonjour numero 6'          TypeError: 'str' object does not support item assignment
```

Il faut enfin noter que si l'on cherche à définir une chaîne de caractères contenant une apostrophe ou des guillemets, on rencontre des problèmes... Pour résoudre ceux-ci on peut utiliser le caractère \ (appelé *backslash* en anglais et *contre-oblique* en français), qui permet de définir un certain nombre de caractères spéciaux.

Codage du symbole	Interprétation dans une chaîne de caractères
\\	\
\'	'
\"	"
\n	Saut de ligne
\t	Tabulation
\r	Retour chariot

Exemple 3.1.

```
>>>x="Un \"Python\" est soit : \n-un serpent dangereux \n-un langage de programma
tion."
>>> print(x)
Un "Python" est soit:
-un serpent dangereux
-un langage de programmation.
```

Notons enfin que l'utilisation de chaîne de caractères permet d'implémenter des fonctions offrant un affichage plus ergonomique : si l'on implémente une fonction qui est censée afficher une valeur `x` à l'écran, on peut lui demander d'afficher également une phrase accompagnant le résultat.

Exemple 3.2. Écrire une fonction Python `sol_eq(a,b)` qui renvoie la solution de l'équation $ax + b = 0$ sous la forme

```
>>> sol_eq(2,1)
'la solution de 2x+1 = 0 est x = -0.5'
```

On renverra une autre phrase lorsqu'il n'y a pas de solution.

4 Optimisation sur un problème classique

On considère le problème suivant : écrire un programme qui prend en entrée une liste `L`, puis renvoie le booléen `False` si celle-ci ne possède aucun terme nul, et l'indice du premier terme nul sinon. La solution algorithmique à ce problème est assez simple : *a priori* on décide de renvoyer `False` (cela correspond à l'initialisation), puis on parcourt les termes de `L` de gauche à droite, et lorsque l'on rencontre un terme nul pour la première fois on renvoie l'indice de celui-ci.

Exercice 4.1. On propose deux programmes :

```
def fonction1(L):
    n=len(L)
    y=False
    for k in range(n):
        if L[k]==0:
            y=k
    return(y)
```

```
def fonction2(L):
    n=len(L)
    y=False
    for k in range(n):
        if L[k]==0 and y==False:
            y=k
    return(y)
```

Expliquer le fonctionnement de chacun de ces programmes :

Optimisation

On peut encore améliorer le programme précédent, en remarquant qu'une fois que l'on a rencontré un terme nul, il est inutile de poursuivre la boucle : la valeur que possède `y` à ce moment là ne changera plus. Et il est d'ailleurs souhaitable de terminer cette boucle, car le temps qui est ensuite pris pour parcourir le reste de la liste est du temps perdu, et pour peu que la liste soit longue cela nuira véritablement à l'efficacité du programme.

Pour forcer l'arrêt du programme lorsque la valeur finale de `y` a été trouvée, il suffit d'utiliser l'instruction `break` : lorsque la machine rencontre celle-ci alors qu'elle est en train d'évaluer une boucle for, elle sort immédiatement de la boucle (on dit que la boucle est *brisée*) et passe directement à la suite du programme qu'elle est en train d'évaluer.

On obtient ainsi le programme suivant.

```
def premier_zero(L):
    n=len(L)
    y=False
    for k in range(n):
        if L[k]==0:
            y=k
            break
    return(y)
```

```
>>> premier_zero([1,0,3,0])
1
```

Notez au passage que le code `y==False` est devenu inutile et a donc été supprimé. En effet, lorsque l'on rencontre le premier terme nul du vecteur, la condition `y==False` est nécessairement réalisée. On effectue alors l'opération `y=k` et on sort directement de la boucle : le programme se termine ensuite, sans qu'il y ait de risque que la condition `L[k]==0` soit satisfaite de nouveau, et donc que l'on modifie ensuite la valeur de `y`.

Remarque 4.2.
Il convient d'être prudent lorsque l'on manipule l'instruction `break`, car celle-ci peut rapidement mener à des erreurs de programmation.

Exercice 4.3. Écrire une fonction `premiers_zero_un(L)` qui prend en entrée une liste `L`, et qui renvoie un tuple (n, p) tel que n (resp. p) soit le l'indice du premier zero (resp. 1) apparaissant dans `L`. On écrira une fonction optimisée avec la commande `break`.

5 Introduction à la notion de type

Intuitivement, il est clair que si on pose `x=2` et `y=[1,2,3]`, alors `x` et `y` n'ont pas la même « nature. » En informatique on dira qu'ils n'ont pas le même *type*. On dispose sous Python d'une primitive `type` qui permet de calculer le type d'un objet. Les différents types que nous avons (implicitement) vu jusqu'à présent sont les suivants :

- le type `int` qui regroupe les entiers ;
- le type `float` qui regroupe les nombres décimaux (que l'on qualifie de *nombres flottants*, voir un cours ultérieur) ;
- les types `list`, `tuple` et `str` pour respectivement les listes, tuples et chaînes de caractères ;
- le type `range` qui est spécifique aux objets de la forme `range(i,j,k)` ;
- le type `function` pour les fonctions.

On dispose sous Python d'une primitive `type` qui permet de calculer le type d'un objet.

Exemple 5.1.

```
>>> type(x)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'x' is not defined
>>> x=2
>>> type(x)
<class 'int'>
>>> type(2.33)
<class 'float'>
```

```
>>> type((1,2))
<class 'tuple'>
>>> type('abc')
<class 'str'>
>>> type(range(5))
<class 'range'>
>>>def succ(n):
...    return(n+1)
>>> type(succ)
<class 'function'>
```

La notion de type est fondamentale en informatique. Il est donc important de bien retenir le nom des types définis ci-dessus, car cela permettra de bien comprendre les messages d'erreur que l'on rencontrera. À titre d'exemple, observons les messages d'erreurs suivants :

```
>>> succ([1,2,3])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    def succ(n):
  File "<tmp 1>", line 2, in succ
TypeError: can only concatenate list (not "int") to list

>>> range(5)+[1,2]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'range' and 'list'
```

Commentaire :

Nous pouvons encore donner deux autres types que l'on rencontrera parfois.

```
>>> type(print)
<class 'builtin_function_or_method'>
```

```
>>>def succ2(n):
...    print(n+1)
>>> type(succ2)
<class 'function'>
>>> type(succ2(2))
3
<class 'NoneType'>
```

Le message de gauche indique que, contrairement à ce qu'on pourrait imaginer, la fonction `print` n'a pas le type `function`. Son type est sensiblement différent, il s'agit de `builtin_fonction`, aussi appelé `method`. Cela résulte du caractère « orienté objet » de Python, qui ne sera pas véritablement abordé en classes préparatoires (nous en reparlerons néanmoins brièvement en TP).

Le message de droite peut également paraître surprenant. Bien entendu, `succ2` est une fonction, donc pas de problème concernant son type. Mais il faut remarquer que cette fonction ne renvoie rien (c'est ce que nous avions appelé une procédure dans le cours n° 2), donc `succ2(2)` n'est pas véritablement un objet... c'est juste une instruction qui n'a pas de contenu (intuitivement, on peut considérer que c'est un objet vide). Et il y a justement un type prévu pour cela, le type `Nonetype`.

Pour finir, signalons que les séquences ne sont pas nécessairement homogènes : une séquence peut contenir à la fois des objets de type `int`, `str` et `list`. Au final ce sera toujours une séquence.

Exemple 5.2.

```
>>> L=[2,'abc',[1,2,3]]
>>> type(L)
<class 'list'>
```