1 Les listes

Définition 1.1 (Liste) Une *liste* en Python est une collection ordonnée (ou séquence) d'éléments séparés par des virgules, le tout étant entre crochet. Le type de données associées aux listes est **`list`**.

Exemple 1.2 `[1,6,23]` est une liste composée de trois entiers.

Remarque 1.3 Les listes Python peuvent être composées d'éléments de types différents mais nous ne n'en manipulerons pas cette année.

Définition 1.4 (Longueur d'une liste) On appelle *longueur d'une liste*, le nombre d'éléments qui la compose.

Exemple 1.5 `M = ["a","b","c","d","e"]` a pour longueur 5.

En Python l'instruction `len(L)` renvoie le nombre d'éléments de la liste `L`, autrement dit sa longueur.

1.1 Définir une liste

1.1.1 Par extension

On indique entre crochets les éléments à mettre :

```
L = [x1,x2,...,xn]
```

Exemple 1.6 Les listes des exemples 1.2 et 1.5 sont des listes définies par extension.

1.1.2 Par compréhension

On indique entre crochets la façon dont on construit la liste à l'aide d'une boucle `for` :

```
L = [expression for element in itérable]
```

On peut également ajouter des conditions après la boucle `for` :

```
L = [expression for element in itérable if condition]
```

Exemple 1.7

- La liste `[k**2 for k in range(5)]` est une liste définie par compréhension. Elle correspond à la liste `[0,1,4,9,16]`
- La liste `[k**3 for k in range(6) if k%2 == 0]` est aussi une liste définie par compréhension. Elle correspond à la liste `[0,8,64]`.

1.1.3 Par transtypage sur des types de données dits *itérables*

Lorsque `x` est de type `range`, `tuple`, `set` ou `str`, l'instruction `list(x)` permet de définir une liste dont les éléments sont ceux de la structure `x` dans le même ordre.

Exemple 1.8

- L'instruction `list("NUM")` renvoie la liste `['N','U','M']`.
- L'instruction `list(range(10))` produit la liste `[0,1,2,3,4,5,6,7,8,9]`.

Remarque 1.9 On peut définit une liste vide de deux façons : soit par extension en ne mettant aucun élément entre les crochets ou bien par transtypage : `L = []` ou `L = list()`

1.2 Opérations

1.2.1 Avec l'opérateur +

`L + [...]` renvoie une liste résultant de la concaténation de la liste `L` avec la liste `[...]`.

Exemple 1.10 L'instruction `["bonjour","1ère"] + ["NSI"]` renvoie `["bonjour","1ère","NSI"]`.

1.2.2 Avec l'opérateur ⋆

`n*L` ou `L*n` renvoient une liste répétant `n` fois la liste `L`.

Cette opération est très intéressante pour initialiser une liste dont on connaît la longueur. Par exemple, si l'on sait que la liste contiendra cinq éléments, on peut l'initialiser avec la liste `5*[0]` qui correspond à la liste `[0,0,0,0,0]`.
Attention toutefois à manipuler cette opération avec précaution. En effet, si on répète une liste contenant une sous-liste, alors la modification d'un élément d'une des sous-listes répétées modifie toutes celles qui lui correspondent (voir section 1.6).

Exemple 1.11 L'instruction `L = [1,2] ; 2*L` renvoie `[1,2,1,2]`.

1.3 Tests

1.3.1 Appartenance

`x in L` renvoie `True` si `x` est un élément de la liste`L` et `False` sinon.

1.3.2 Non appartenance

`x not in L` renvoie `True` si `x` n'est pas un élément de la liste `L` et `False` sinon.

1.3.3 Egalité

Deux listes sont identiques quand elle sont composées des mêmes éléments dans le même ordre.

`L == M` renvoie `True` si `L` et `M` sont deux listes identiques et `False` sinon.

1.4 Extraire et modifier des données d'une liste

Définition 1.12 (Indice d'un élément d'une liste) On appelle *indice* (*index* en anglais) le numéro indiquant la position de l'élément dans la liste. En informatique, les indices commençant toujours à 0, le premier élément de la liste a donc pour indice 0, le deuxième élément a pour indice 1, ..., le n-ième élément (n nombre entier strictement positif) a pour indice $n-1$.

1.4.1 Accéder aux données

`L[i]` renvoie la `(i + 1)`ème donnée de `L` si `i` est un nombre entier tel que `0 ⩽ i < len(L)`.
Autrement dit `L[i]` est l'élément d'indice `i`.

`L[i:j]` renvoie une liste correspond à l'extraction des données comprises entre `L[i]` et `L[j-1]` incluses si `0 ⩽ i < j ⩽ len(L)`.

`L[i:j:k]` renvoie une liste correspond à l'extraction de `k` en `k` des données comprises entre `L[i]` et `L[j-1]` incluses si `0 ⩽ i < j ⩽ len(L)` et si `k` est un entier strictement positif.

Remarque 1.13 En utilisant les deux dernières instructions on dit que l'on fait du *slicing* (on récupère des sous-listes).

Exemple 1.14 On considère les instructions suivantes :

```
L = [1,3,5,7,9,11]
len(L); L[2]; L[1:4]; L[1:3:2]
```

La seconde ligne renvoie dans l'ordre :

`6`, `5`, `[3,5,7]` et `[3]`.

1.4.2 Modifier les données

`L[i] = x` remplace la `(i + 1)`ème donnée de `L` par `x` si `i` est un nombre entier tel que `0 ⩽ i < len(L)`.

`del(L[i])` supprime la `(i + 1)`ème donnée de `L` si `i` est un nombre entier tel que `0 ⩽ i < len(L)`.

Exemple 1.15 Soit la liste `L = [1,"essai",False]`

```
L[1]="valeur remplacée"; L
del(L[1]); L
```

La 1ère ligne renvoie la liste `[1,"valeur remplacée",False]`.
La 2ème ligne renvoie la liste `[1,False]`.

Remarque 1.16 On peut également utiliser des méthodes agissant directement sur la liste `L` (voir la section 1.7).

1.5 Parcourir les données d'une liste

La liste `L` peut être parcourue, élément par élément, à l'aide d'une boucle. On peut utiliser l'une ou l'autre des syntaxes suivantes :

```
for element in L       # parcours sur les éléments
for i in range(len(L)) # parcours sur les index des éléments
```

Exemple 1.17 Soit la liste `L = [1, 2.5, "bonjour"]`.
Les instructions ci-dessous produisent le même affichage dans la console :

```
for elem in L:
    print(elem)
```

```
for k in range(len(L)):
    print(L[k])
```

```
1
2.5
bonjour
```

1.6 Copier et initialiser

Si l'on souhaite copier une liste dans une seconde liste, quelques précautions sont à prendre.

Exemple 1.18 Les instructions suivantes auront pour effet de modifier les deux listes `L1` et `L2` à la fois en `['a',1,4.5,'nsi']`.

```
L1 = ['a',1,2.0,'nsi']
L2 = L1
L2[2] = 4.5
```

En fait, les variables `L1` et `L2` font référence à la même liste, seule l'adresse de la liste `L1` a été enregistrée, ainsi toute modification de l'une des deux listes affectera l'autre.

1.6.1 Copie d'une liste non composée de listes

Pour réaliser une copie de liste en `Python`, il faut copier les éléments de la première liste `L1` dans la deuxième liste `L2`. On pourra procéder des trois manières suivantes :

Par *slicing* :

```
L2 = L1[:]
```

Avec la méthode `copy()` :

```
L2 = L1.copy()
```

Par compréhension :

```
L2 = [elem for elem in L1]
```

Attention, la méthode `copy` ne fonctionne pas avec les listes de listes.

1.6.2 Initialisation de listes

Quand on utilise une structure de liste dans un programme, on est souvent amené à initialiser sa liste de longueur `n` en une liste ne contenant que des 0. Dans ce cas on pourra utiliser au choix :

```
L = n*[0]
```
ou
```
L = [0]*n
```

Remarque 1.19 Attention cependant au cas particulier des listes de listes :
L'instruction `L = 3*[[0,0]]` produit à première vue la liste `[[0,0],[0,0],[0,0]` mais `L` contient en fait trois fois la même liste `[0,0]` (même adresse mémoire) et toute modification se répercutera plusieurs fois. En effet l'instruction `L[1][1]=1` ne modifie pas seulement le 2ème élément de la 2ème liste mais également le 2ème élément de la première et de la dernière liste : la liste `L` est modifiée en `[[0,1],[0,1],[0,1]]`

1.6.3 Initialisation de listes de listes

Pour initialiser une liste de listes on procède généralement par compréhension :

```
L = [[e1,e2,...,ek] for i in range(n)]
```

Cette instruction produit une liste `L` composée de `n` listes de la forme `[e1,e2,...,ek]` où `e1,e2`, ... et `ek` désignent des données.

Exemple 1.20 L'instruction `L=[[0,0] for i in range(3)]` produit la liste `[[0,0],[0,0],[0,0]]`.
Contrairement à l'exemple de la remarque 1.19, cette fois l'instruction `L[1][1]=1` modifie uniquement le 2ème élément de la 2ème liste et modifie ainsi `L` en la liste `[[0,0],[0,1],[0,0]]`.

1.6.4 Copier des listes de listes

Pour copier dans une seconde liste `L2` le contenu d'une liste `L1`, même s'il est possible de procéder par compréhension, il est généralement plus simple d'importer la fonction `deepcopy(L)` de la bibliothèque `copy` :

```
from copy import deepcopy
L2 = deepcopy(L1)
```

1.7 Des méthodes agissant sur les listes

Il existe un grand nombre d'autres méthodes agissant sur les listes. On ne les présente pas toutes ici. La liste complète peut être obtenue avec l'instruction `help(list)`.

1.7.1 La méthode `append`

`L.append(x)` ajoute à la fin de la liste `L` la donnée `x`.

Exemple 1.21 Soit la liste `L = [1,2,3]`. L'instruction `L.append(7)` modifie `L` en `[1,2,3,7]`.

1.7.2 La méthode `extend`

`L.extend(M)` ajoute à la fin de la liste `L` les éléments de la liste `M`.

Exemple 1.22 Soit la liste `L = [1,7]`. L'instruction `L.extend([32,9])` modifie `L` en `[1,7,32,9]`.

Remarque 1.23 Si `L = ["A","B"]`, l'instruction `L.extend("abc")` modifie `L` en `['A','B','a','b','c']`.

1.7.3 La méthode `remove`

`L.remove(x)` supprime de la liste `L` le premier élément dont la valeur est `x`
ou une erreur si `x` n'existe pas dans `L`.

Exemple 1.24 Soit la liste `L = [1,3,5,7]`. L'instruction `L.remove(3)` modifie `L` en `[1,5,7]`.

1.7.4 La méthode `pop`

`x = L.pop(i)` supprime la `(i + 1)`ème donnée de `L` et l'affecte à la variable `x`
si `i` est un nombre entier tel que $0 \leqslant$ `i < len(L)`
et renvoie une erreur sinon.

Exemple 1.25 Soit la liste `L = [1,3,5,7]`. L'instruction `x = L.pop(2)` affecte à `x` la valeur `5` et modifie `L` en `[1,3,7]`.

1.7.5 La méthode `index`

`L.index(elem)` renvoie l'indice de la première occurrence de l'élément `elem` dans la liste `L`
ou une erreur si `elem` n'est pas présent.

Exemple 1.26 Soit la liste `L = [4,15,2,7]`. L'instruction `L.index(15)` renvoie 1.

1.7.6 La méthode `count`

`x = L.count(elem)` renvoie le nombre d'occurrences de l'élément `elem` dans la liste `L`

Exemple 1.27 Soit la liste `L = [0,1,2,1,1,1]`. `L.count(1)` renvoie 4 et `L.count(7)` renvoie 0.

1.7.7 La méthode `insert`

`L.insert(i, elem)` insère l'élément `elem` à l'indice `i` dans la liste `L`

Exemple 1.28 Soit la liste `L = list(range(6))`. L'instruction `L.insert(3,27)` insère `27` à la quatrième position dans `L`, autrement dit `L` est modifiée en `[0,1,2,27,3,4,5]`.

1.7.8 La méthode `sort`

`L.sort()` trie la liste `L` par ordre croissant si elle est composée d'éléments comparables et renvoie une erreur sinon.

Exemple 1.29

- Soit la liste `L = [15,27,3,12,5].` `L.sort()` modifie `L` en `[3,5,12,15,27].`
- Soit la liste `L = ['bac','num','nsi','b'].` `L.sort()` modifie `L` en `['b','bac','nsi','num']`
- Soit la liste `L = [15,27,'nsi'].` `L.sort()` renvoie une erreur ($<$ non compatible entre `str` et `int`)

Remarque 1.30 Il est possible de trier la liste dans l'ordre décroissant. Il suffira d'utiliser l'instruction `L.sort(reverse=True)`.

2 Les tuples ou n-uplets

Définition 2.1 (Tuple ou n-uplet) Les données de type **`tuple`** sont des séquences ordonnées d'objets séparés par des virgules et encadrés par des parenthèses. Ce sont des données **non modifiables**.

Exemple 2.2 `(1,True,[3,4])` est un tuple composé d'un entier, d'un booléen et d'une liste.

Remarque 2.3 Les tuples `Python` peuvent être composées d'éléments de types différents, et même de listes ou de tuples.

Définition 2.4 (Longueur d'un tuple) On appelle *longueur d'un tuple*, le nombre d'éléments qui le compose.

Exemple 2.5 `(1,True,[3,4])` est une tuple composée de 3 éléments.

En `Python` l'instruction `len(T)` renvoie le nombre d'éléments du tuple `T`, autrement dit sa longueur.

2.1 Définir un tuple

Comme pour les listes, on peut procéder par extension, par compréhension ou par transtypage.

2.1.1 Par extension

On indique les éléments à mettre, éventuellement entre parenthèses :

Avec un seul élément : `T = (x,)` ou `T = x,`

Avec $n \geqslant 2$ éléments : `T = (x1,x2,...,xn)` ou `T = x1,x2,...,xn`

Exemple 2.6 Le tuple `T1 = (2,)` est un tuple défini par extension à un seul élément.
Le tuple `T2 = (1,True,[3,4])` est défini par extension avec 3 éléments.

Remarque 2.7 Lors d'une affectation parallèle (instruction de la forme `var1,var2, ...,varn = x1,x2,...,xn` qui permet d'affecter respectivement aux variables `var1,var2,...,varn` les valeurs `x1,x2,...,xn`), ce sont en fait des tuples qui interviennent.

2.1.2 Par compréhension

On indique entre parenthèses la façon dont on construit le tuple à l'aide d'une boucle `for` :

```
T = tuple(expression for element in itérable)
```

On peut également ajouter des conditions après la boucle `for` :

```
T = tuple(expression for element in itérable if condition)
```

Remarque 2.8 Attention, contrairement aux listes, ici la fonction de transtypage associée `tuple` est obligatoire pour définir un tuple par compréhension.

Exemple 2.9 Le tuple `tuple(k for k in range(10) if k%3 == 0)` est un tuple défini par compréhension. Il correspond au tuple `(0,3,6,9)`.

2.1.3 Par transtypage sur des types de données dits *itérables*

Lorsque `x` est de type `range`, `list`, `set` ou `str`, l'instruction `tuple(x)` permet de définir un tuple dont les éléments sont ceux de la structure `x` dans le même ordre.

Exemple 2.10

- L'instruction `tuple("NSI")` renvoie le tuple `('N','S','I')`.
- L'instruction `tuple(range(6))` produit le tuple `(0,1,2,3,4)`.

Remarque 2.11 On peut définit un tuple vide de deux façons : soit par extension en ne mettant aucun élément entre les parenthèses ou bien en utilisant la fonction associées de transtypage : `T = ()` ou `T = tuple()`

2.2 Opérations

Les opérations + et $\star$ sont compatibles entre des tuples, comme pour les listes.

2.3 Tests

Ce sont les mêmes tests que pour les listes (même syntaxe).

2.4 Extraire les données d'un tuple

Définition 2.12 (Indice d'un élément d'un tuple) On appelle *indice* (*index* en anglais) le numéro indiquant la position de l'élément dans le tuple.

Pour accéder aux données d'un tuple, on procède comme pour listes.
Par contre, les tuples ne sont pas modifiables donc des instructions du type `T = tuple(range(4)) ; T[1] = 'a'` renverront des erreurs (`'tuple' object does not support item assignment`).

2.5 Parcourir les données

Comme pour les listes, un tuple `T` peut être parcouru, élément par élément, à l'aide d'une boucle. On peut utiliser l'une ou l'autre des syntaxes suivantes :

```
for element in T       # parcours sur les éléments
for i in range(len(T)) # parcours sur les index des éléments
```

.

2.6 Copier des tuples

On procède comme pour les listes. Attention, un tuple n'étant pas modifiable, il n'est pas possible de changer la valeur d'un élément d'un tuple à moins de le transtyper en liste le temps de le modifier.

2.7 Méthodes agissant sur les tuples

Comme les tuples sont non modifiables, seules les méthodes `count` et `index` sont utilisables. Elles s'utilisent de la même façon qu'avec les listes.

2.8 Lien avec les fonctions

Lorsque dans une fonction on renvoie n valeurs ($n \geqslant 2$) en utilisant une instruction de la forme `return var1, var2,..,varn`, la fonction renvoie en fait un tuple. Les données renvoyées peuvent être ensuite récupérées soit à l'aide d'une affectation parallèle, soit dans une variable de type tuple.

Exemple 2.13 La fonction `somme_produit(a,b)` ci-dessous. renvoie deux résultats : la somme et le produit. On peut les récupérer sous la forme d'un tuple ou bien dans deux variables distinctes `somme` et `produit`.

```
def somme_produit(a,b) :
    return a+b,a*b
res = somme_produit(5,6)
somme, produit = somme_produit(5,6)
```

- Ici la variable `res` contient le tuple `(11,30)`,
- `somme` contient l'entier 11 et
- `produit` contient l'entier 30.