CHAPITRE 2

Types construits

1 Listes python

1.1 Définition en extension

Définition 1. Un tableau (ou liste) est une séquence d'éléments de même type. Il est possible de connaitre ou de modifier la valeur de ces éléments grâce à leur position dans la séquence.

On peut définir une liste en énumérant tous les éléments qui la composent. On note ces éléments les uns à la suite des autres entre crochets `[` et `]`, séparés par une virgule `,`. On note la liste vide `[]`.

Code python

```
vide = []
tableau = [3, -1, 8, 11]
listeOS = ["GNU/linux", "windows", "macOS"]
lst = [True, True, False, True, True, True]
```

La **longueur** d'une liste est le nombre d'élément qui la composent. On utilise la fonction `len` pour connaître la longueur d'une liste en python. La longueur de la liste vide est `0`.

Code python

```
print(len(vide))
print(len(tableau))
print(len(listeOS))
print(len(lst))
```

Résultat

```
0
4
3
6
```

1.2 Indice d'un élément dans une liste

Si `liste` est une liste de `n` éléments, alors **l'indice** du *premier* élément est `0`, l'indice du second élément est `1`, etc. L'indice du *dernier* élément est `n-1`.

Code python

```
# liste    : [ _, _, _, ..., _]
# indice   :  0  1  2, ..., n-1
# avec n = len(list_exemple)
```

L'élément d'indice `i` de `liste` se note `liste[i]`.

Code python

```
print(tableau[1], listeOS[len(listeOS) - 1], lst[2])
```

Résultat

```
-1 macOS False
```

On peut modifier un élément d'une liste à l'aide de son indice.

Code python

```
lst = [1, 2, 3]
lst[0] = 10
print(lst)
```

Résultat

```
[10, 2, 3]
```

Soit `liste` une liste de taille n. On dit qu'un entier `i` est un indice **compatible avec la taille de `liste`** si on a $0 \leq i < n$. Si on accède ou modifie une liste avec un indice qui n'est pas compatible avec la taille de la liste, python soulève une erreur de type `IndexError`.

Code python

```
tableau = [3, -1, 8, 11]
# tableau a pour taille 4, les indices compatibles avec la taille de
↪ tableau sont les nombres i tels que 0 <= i < 4
tableau[4] = 15
```

Résultat

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call
↪ last)
Cell In[3045], line 3
      1 tableau = [3, -1, 8, 11]
      2 # tableau a pour taille 4, les indices compatibles avec la taille
      ↪ de tableau sont les nombres i tels que 0 <= i < 4
----> 3 tableau[4] = 15

IndexError: list assignment index out of range
```

1.3 Parcours d'une liste

On peut **parcourir une liste par valeur**. La variable de boucle prend successivement chaque valeur de la liste.

Code python

```
for os in listeOS:
    # variable de boucle : os
    print(os)
```

Résultat

```
GNU/linux
windows
macOS
```

On peut **parcourir une liste par indice**. La variable de boucle prend alors les valeurs des indices spécifiés dans le `range`.

Code python

```
for i in range(len(listeOS)):
    # variable de boucle : i
    os = listeOS[i]
    print(i, os)
```

Résultat

```
0 GNU/linux
1 windows
2 macOS
```

1.4 Définition en compréhension d'une liste

On peut définir une liste **en compréhension**, c'est à dire en décrivant les éléments qui la composent. On utilise pour cela la syntaxe `[<elem> for <var> in <seq>]` avec :

- `<var>` est une variable de boucle ;
- `<elem>` est une expression qui se calcule en fonction de la variable de boucle `<var>` ;
- `<seq>` est un itérable (une liste, une `range`, une chaîne de caractères…).

Exemple 1.

- Initialiser un tableau de taille arbitraire avec une valeur.

Résultat
```
>>> [0 for i in range(6)]
[0, 0, 0, 0, 0, 0]
```

- Calculer les images d'une fonction.

Résultat
```
>>> [ 3*i - 2 for i in range(5)]
[-2, 1, 4, 7, 10]
```

Remarque. On peut aussi rajouter des conditions pour **filtrer** certains éléments avec la syntaxe : `[<elem> for <var> in <seq> if <booleen>]`.

Exemple 2.

Code python
```
1 tab = [7, 2, -3, -5, 4]
2 tab_pos = [elem for elem in tab if elem > 0]
3 print(tab_pos)
4 tab_ipair = [tab[i] for i in range(len(tab)) if i%2 == 0]
5 print(tab_ipair)
```

Résultat
```
[7, 2, 4]
[7, -3, 4]
```

Exercice 1. Écrire une fonction `concetane` qui prend en entrée deux listes `lst1` et `lst2` et qui renvoie une liste constituée des éléments de `lst1` suivis des éléments de `lst2` dans cet ordre.

Code python
```
1 def concatene(lst1, lst2):
2     """ [int], [int] -> [int]
3     Renvoie la concaténation de lst1 et lst2 """
4     ..........................................................
5     ..........................................................
6     ..........................................................
7     ..........................................................
8     ..........................................................
9     ..........................................................
```

Code python
```
1 lst1, lst2 = [1, 2, 3], [4, 5, 6]
2 print(concatene(lst1, lst2))
3 print(concatene([], lst2))
4 print(concatene(lst1, []))
```

Résultat
```
[1, 2, 3, 4, 5, 6]
[4, 5, 6]
[1, 2, 3]
```

1.5 Type mutable et phénomène d'aliasing

On considère la séquenced d'instructions ci-desssous.

Résultat
```
>>> a = [1, 2]
>>> b = a
>>> b[0] = 3
>>> a
[3, 2]
```

Ce résultat s'explique par le fait que lorsque l'on exécute l'instruction `b = a`, `a` et `b` font référence **au même objet** dans la mémoire de l'ordinateur. On parle de phénomène d'**aliasing**.

Exemple 3. En raison de ce phénomène, il faut être très prudent lorsque l'on écrit des fonctions qui manipulent des listes. Par exemple, considérons le problème suivant : écrire une fonction `plus_un_a_tout` prend en argument une liste d'entiers `lst` et qui ajoute 1 à tous les éléments. On peut écrire deux fonctions similaires qui répondent à ce problème :

Code python
```python
def plus_un_a_tout_prc(lst):
    """ [int] -> None """
    for i in range(len(lst)):
        lst[i] = lst[i] + 1
```

Code python
```python
def plus_un_a_tout_cmp(lst):
    """ [int] -> [int] """
    return [e + 1 for e in lst]
```

Cependant, ces fonctions sont ***très*** différentes.

- La fonction `plus_un_a_tout_prc` **modifie en place** la liste `lst` passée en argument. On dit qu'elle a un **effet de bord** (on parle de fonction **impure**). Elle renvoie toujours `None`.

Code python
```python
lst = [1, 2, 3]
print(plus_un_a_tout_prc(lst))
print(lst)
```

Résultat
```
None
[2, 3, 4]
```

- La fonction `plus_un_a_tout_cmp` **construit et renvoie une nouvelle liste**, indépendante de la liste initiale. La liste initiale `lst` **n'est pas** modifiée (on parle de fonction **pure**).

Code python
```python
lst = [1, 2, 3]
print(plus_un_a_tout_cmp(lst))
print(lst)
```

Résultat
```
[2, 3, 4]
[1, 2, 3]
```

Remarque. Vous avez écrit les fonctions **pures** `ajoute_fin`, `supprimer_pos`, et `concatene`. Celles-ci construisent et renvoie de nouvelles listes, indépendantes des listes initiales, qui ne sont pas modifiées. En python, il existe un analogue **impur** de ces fonctions : il s'agit des méthodes `append`, `pop`, et `extend`.

- `lst.append(e)` : ajoute à la fin de la liste `lst` l'élément `e`.

Code python
```python
lst = [1, 2, 3]
lst.append(4) # renvoie None
print(lst)
```

Résultat
```
[1, 2, 3, 4]
```

- `lst.pop()` : supprime et renvoie le dernier élément de la liste `lst`.

Code python
```python
lst = [1, 2, 3]
lst.pop() # renvoie 3
print(lst)
```

Résultat
```
[1, 2]
```

- `lst1.extend(lst2)` : ajoute à la fin de `lst1` les éléments de `lst2`.

Code python
```python
lst = [1, 2, 3]
lst.extend([4, 5]) # renvoie None
print(lst)
```

Résultat
```
[1, 2, 3, 4, 5]
```