1 Exercices de connaissance du cours

1.1 Listes

À quelles valeurs s'évaluent les expressions suivantes ?

```
[1, 1+1, 4-1]
len(['a', 'b'])
[1>0, True, 'acb' != 'abc']
[]
```

1.2 Ajout dans une séquence

Le but de cet exercice est de bien comprendre la méthode `append` et les opérations de concaténation et répétition. Si vous avez un doute, vous pouvez vous auto-corriger dans Thonny.

On souhaite ajouter à la fin d'un itérable (de type chaîne ou liste d'entiers) 2 fois un même élément (un caractère ou un entier). Par exemple :

```
$$$ ajout_doublon('coucou', '!')
'coucou!!'
```

Commenter la correction des fonctions suivantes :

```
def ajout_doublon1(s:str, x:str) -> str:
    return s + 2*x
```

```
def ajout_doublon2(l:list[int], x:int):
   res = []
   for elt in l:
       res.append(elt)
   res.append([x] * 2)
   return res
```

```
def ajout_doublon3(l:list[int], x:int):
    return l + 2 * [x]
```

```
def ajout_doublon4(s:str, x:str):
    s.append(2 * x)
    return s
```

```
def ajout_doublon5(l:list[int], x:int):
  res = []
  for elt in l:
      res.append(elt)
  res.append(x)
  return res.append(x)
```

1.3 Code mystère

On donne la fonction suivante :

```
def mystere(l:list[int]) -> int:
    """
    Precondition: l est non vide et contient uniquement des elts positifs ou nuls
    """
    res = 0
    for elt in l:
        if elt > res:
            res = elt
    return res
```

1. Que vaut `mystere([2])` ? `mystere([4, 3, 4, 6])` ?

2. Compléter la documentation de cette fonction.

2 Exécution d'une boucle `for` et tableau de la mémoire

3. Dérouler avec un tableau de la mémoire l'exécution de `mystere([4, 3, 4, 6])`, pour la fonction définie précédemment.

3 Écriture de tests et de code

Pour chacune des fonctions décrites ci-dessous, écrire la documentation complète, en particulier les tests, ainsi que le code.

4. Une fonction `nombre_occurrences` qui prend en paramètre un caractère `carac` et une chaîne `chaine` et qui renvoie le nombre d'occurrences de `carac` dans `chaine` :

```
>>> nombre_occurrences('a', 'abracadabra')
5
```

5. Une fonction `nombre_occurrences2` qui prend en paramètre une chaine `caracteres` non vide et une chaîne `chaine` et qui renvoie le nombre d'occurrences cumulé des caractères de `caracteres` dans `chaine` :

```
>>> nombre_occurrences2('afbcg', 'abracadabra')
8
```

6. Une fonction `produit` qui prend en paramètre une liste d'entiers et qui renvoie le produit des éléments qu'elle contient.

```
>>> produit([2, 3, -5])
-30
```

7. Une fonction `pairs_et_positifs` qui prend en paramètre une liste d'entiers `l` et renvoie une nouvelle liste contenant uniquement les entiers strictement positifs et pairs de `l`, dans le même ordre. Que faudrait-il changer pour obtenir à la fois les entiers pairs et les entiers strictement positifs ?

```
>>> pairs_positifs([2, 3, -6, 4, 5, 12])
[2, 4, 12]
```

8. Écrire une fonction `incremente_chiffres` qui prend en paramètre une chaîne de caractères `chaine` et renvoie une chaîne contenant dans le même ordre les caractères de `chaine`, mais tels que les chiffres ont été remplacés par leur valeur incrémentée. Par exemple :

```
>>> incremente_chiffres('3%$6')
'4%$7'
```

On utilisera `str.isdigit()`.

```
>>> 'a'.isdigit()
False
>>> '42'.isdigit()
True
```

On rappelle que $n!$ (factorielle n) est défini comme le produit des entiers de 1 à n :

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n = \prod_{i=1}^{n} i$$

9. Sans utiliser fonction `factorielle` intermédiaire, écrire une fonction `liste_factorielle` qui prend en paramètre un entier n positif ou nul et renvoie une liste d'entier contenant les valeurs successives de $i!$, pour i allant de 0 à n inclus. Par exemple :

```
>>> liste_factorielle(3)
[1, 1, 2, 6]
```