♦ TP2 – Rappels sur le langage Python (II)

I. — Chaînes de caractères

1) Définition

Une chaîne de caractères est une suite ordonnée de caractères quelconques (lettres, chiffres, symboles de ponctuation, etc) délimitée par des guillemets (`" "`) ou par des apostrophes (`' '`).

Par exemple, `"#TB2 'Le-Chesnoy'"` et `'@ !§a23 m*Et'` sont des chaînes de caractères.

La chaîne vide, qui ne contient aucun caractère, se définit par `""` ou `''`.

Une chaîne de caractères possède un type propre en Python : le type *string* (`str`). Il s'agit d'un objet *non mutable* ce qui signifie qu'**on ne peut pas modifier ou supprimer** une partie des caractères de la chaîne.

Remarque. Les guillemets (ou les apostrophes) sont indispensables pour différencier les chaînes de caractères des noms des variables. Ainsi, `ABC` est le nom d'une variable alors que `"ABC"` est une chaîne de caractères. Dès lors, l'instruction `print(ABC)` affichera la valeur de la variable `ABC` (ou un message d'erreur indiquant : `name 'ABC' is not defined` si cette variable n'a pas été définie précédemment) alors que `print("ABC")` affichera `ABC` (c'est-à-dire la chaîne de caractères formée par les trois lettres `A`, `B` et `C`).

Notons à ce propos que, parmi d'autres, `True` et `False` font exception à cette règle et ne désignent pas des variables mais des booléens. Ainsi, à l'affichage, `print("True")` ou `print(True)` donneront la même chose mais `return "True"` et `return True` ne renvoient pas le même type d'objet. D'ailleurs, dans une instruction conditionnelle, il vaut mieux éviter d'utiliser `return True`. Ainsi, plutôt que d'écrire

```
def est_positif(x):
  if (x >= 0):
    return True
  else:
    return False
```

il est préférable d'écrire

```
def est_positif(x):
  return x >= 0
```

2) Longueur et éléments d'une chaîne

Le nombre de caractères qui composent une chaîne est appelé la longueur de la chaîne. Tous les caractères, y compris les espaces, sont comptés.

La fonction `len` renvoie la longueur d'une chaîne de caractères (*lenght* signifiant longueur en anglais).

Par exemple, `len("#TB2 'Le-Chesnoy'")` renvoie 17.

Les caractères d'une chaîne de longueur n sont numérotés de 0 à $n-1$ de la gauche vers la droite et de -1 à $-n$ de la droite vers la gauche. Pour une chaîne de longueur n, on dit qu'un entier k est un indice valable (ou admissible) si $-n \leqslant k \leqslant n-1$.

Si k est un indice valable pour la chaîne `ch` alors `ch[k]` renvoie le caractère numéroté k dans la chaîne `ch`. Si k n'est pas un indice valable alors `ch[k]` renvoie un message d'erreur en indiquant : `string index out of range`.

Considérons par exemple la chaîne `ch = "chaine_1"`.

ch[0]	ch[1]	ch[2]	ch[3]	ch[4]	ch[5]	ch[6]	ch[7]
↓	↓	↓	↓	↓	↓	↓	↓
c	h	a	i	n	e	_	1
↑	↑	↑	↑	↑	↑	↑	↑
ch[-8]	ch[-7]	ch[-6]	ch[-5]	ch[-4]	ch[-3]	ch[-2]	ch[-1]

Ainsi, `ch[3]` renvoie `i`, `ch[-2]` renvoie `_` et `ch[8]` renvoie un message d'erreur alors que `ch[-8]` renvoie `c`.

3) Opérations sur les chaînes de caractères

Comme on l'a dit, une chaîne de caractères est un objet non mutable donc on ne peut pas supprimer ou modifier des caractères de la chaîne. En revanche, il est possible de faire certaines opérations sur une ou plusieurs listes :

1. **extraction** ou **slicing** : étant donné une chaîne `ch` et deux indices a et b valables tels que $a < b$ alors
   - `ch[a:b]` renvoie la sous-chaîne de `ch` comprise entre les caractères d'indices a et $b-1$;
   - si $p \in \mathbb{N}^*$, `ch[a:b:p]` renvoie la chaîne formée des caractères de `ch` d'indices compris entre a et $b-1$ avec un pas p, c'est-à-dire la chaîne dont les caractères sont `ch[a]`, `ch[a+p]`, `ch[a+2p]`, ..., `ch[a+kp]` où $a + kp < b \leqslant a + (k+1)p$.

   Ainsi, si `ch = "chaine_1"` alors `ch[2:5]` renvoie la chaîne `"ain"` et `ch[1:5:2]` renvoie la chaîne `"hi"`.

   Remarque. Si $a \geqslant b$ alors `ch[a:b]` renvoie la chaîne vide (même si les indices a et b ne sont pas valables).
2. **concaténation** : étant donné deux chaînes de caractères `ch1` et `ch2`, l'instruction `ch1+ch2` concatène les deux chaînes, c'est-à-dire renvoie une nouvelle chaîne constituée des caractères de `ch1` suivi de ceux de `ch2`.

   Ainsi, si `ch1="tic"` et `ch2="tac"` alors `ch1+ch2` renvoie la chaîne `"tictac"`.

   En particulier, on peut utiliser ceci pour modifier une chaîne de caractères en lui ajoutant un caractère à la fin. Ainsi, `ch1 = ch1+"s"` modifie la chaîne `ch1` en `"tics"`. On peut également utiliser la syntaxe `ch1 += "s"`.
3. **répétition** : si on souhaite concaténer plusieurs fois la même chaîne, on peut utiliser l'opérateur `*` : plutôt que d'écrire `ch + ch + ch + ch`, on peut écrire `ch * 4`.

   Ainsi, `ch1 * 3` renvoie la chaîne `"tictictic"`.
4. **test d'appartenance** : on peut tester si un caractère `c` appartient à une chaine `ch` grâce à l'instruction `c in ch` qui renvoie `True` si le caractère `c` apparaît dans la chaîne `ch` et `False` sinon.

   Ceci fonctionne également pour tester si une chaîne `ch2` est une sous-chaîne d'une chaîne `ch1` en écrivant `ch2 in ch1`.

4) Itération sur les éléments d'une chaîne

Il est possible d'itérer une action sur les éléments d'une chaîne de caractères à l'aide d'une boucle `for`.

Par exemple, si on souhaite écrire une fonction `presence` qui prend en arguments une chaîne de caractères `ch` et un caractère `c` et qui renvoie `True` si le caractère `c` apparaît dans la chaîne `ch` et `False` sinon, plutôt que d'écrire

```
def presence(ch, c):
    n = len(ch)
    for k in range(n):
        if ch[k] == c:
            return True
    return False
```

il est préférable d'écrire

```
def presence(ch, c):
    for e in ch:
        if e == c:
            return True
    return False
```

5) Exercices

Exercice 1. Parmi les syntaxes suivantes, déterminer lesquelles sont correctes en Python (en supposant qu'aucune variable n'a été définie précédemment) et, le cas échéant, déterminer le résultat obtenu.

1) `'4' + 'a'`	2) `4 + 'a'`	3) `'2 - a'`	4) `4 * a`
5) `4 * 'a'`	6) `'a'**3`	7) `'ab' + 'bc'`	8) `'ab' * 'bc'`

Exercice 2. On considère la chaîne de caractères `a = "Bonjour"`. Dire si les instructions suivantes sont correctes et, le cas échéant, déterminer le résultat obtenu.

1) `a[0]`	2) `a[3]`	3) `len(a)`	4) `a[7]`
5) `a[0] + a[-1]`	6) `a[0] + 4*a[-1]`	7) `"B" in a`	8) `"b" in a`
9) `a[1] == 'o'`	10) `a[1] = 'B'`	11) `a + 'a'`	12) `a + a`

Exercice 3. Pour chacun des programmes suivants, déterminer la valeur des différentes variables à la fin de l'exécution.

1)
```
a = '2'
b = 3
a = a + '1'
b = a + 'b'
```

2)
```
A = 'a'
B = 'b'
A = A * 2
B = A + B
```

3)
```
a = 0
b = 1
a = a+b
b = 'a' + 'b'
```

4)
```
a = 'b'
b = 'a'
a = a * 2
b = a + b
```

Exercice 4. Écrire une fonction qui prend en arguments un caractère `c` et un entier $n > 0$ et qui renvoie la chaîne de caractère composée du caractère `c` répété n fois.

Exercice 5. Écrire une fonction qui prend en arguments deux caractères `c` et `d` et deux entiers $n > 0$ et $m > 0$ et qui renvoie la chaîne de caractères composée du caractère `c` répété n fois suivie du caractère `d` répété m fois. On pourra utiliser la fonction de l'exercice précédent.

Exercice 6.

1. Écrire une fonction qui prend en arguments une chaîne de caractères `ch` et un entier $n > 0$ et qui affiche n fois la chaîne `ch` sur n lignes successives.
2. Modifier la fonction précédente de telle sorte que les lignes soient de plus numérotées de 1 à n (la première sera donc de la forme : `1. ch`, la seconde de la forme : `2. ch` et ainsi de suite, le numéro de la ligne étant suivi d'un point et le point suivi d'un espace puis de la chaîne `ch`).

Exercice 7. Écrire une fonction qui prend en arguments un caractère `c` et un entier $n > 0$ et qui affiche n lignes successives de telle sorte que la première ligne contient `c`, la seconde `cc`, la troisième `ccc` et, de manière générale, la $i-$ème ligne contient la chaîne formée du caractère `c` répété i fois.

Exercice 8. En utilisant l'itération sur les chaînes de caractères, écrire une fonction `premiere_occ` qui prend en arguments une chaîne de caractères `ch` et un caractère `c` et qui renvoie le plus petit indice k tel que `ch[k]` soit égal à `c` si `c` apparaît au moins une fois dans `ch` et qui renvoie `None` sinon. Ainsi, `premiere_occ('aabcada', 'a')` renvoie 0 et `premiere_occ('abcd', 'e')` renvoie `None`.

Exercice 9. En utilisant l'itération sur les chaînes de caractères, écrire une fonction `nb_occurrence` qui prend en arguments une chaîne de caractères `ch` et un caractère `c` et qui renvoie le nombre de fois où le caractère `c` apparaît dans la chaîne `ch`. Ainsi, `nb_occurrence('aabcada', 'a')` renvoie 4 et `nb_occurrence('abcd', 'e')` renvoie 0.

Exercice 10. Écrire une fonction `sous_chaine` prenant en arguments deux chaînes `ch1` et `ch2` et qui renvoie `True` si `ch1` est une sous-chaîne de `ch2` ou si `ch2` est une sous-chaîne de `ch1` et `False` sinon.

Exercice 11. Déterminer l'affichage obtenu lors de l'exécution des deux programmes suivants.

```
# programme 1
ch = ''
for k in range(1,5):
  if k % 2 == 1:
    ch += 'a'
  else:
    ch += 'b'
print(ch)
```

```
# programme 2
ch = ''
for k in range(4):
  if k % 2 == 1:
    ch += 'a'
  else:
    ch += 'b'
  print(ch)
```

Exercice 12.

1. Écrire une fonction `triple_six` prenant en argument une chaîne `ch` et qui renvoie `True` si le caractère `'6'` apparaît au moins 3 fois consécutivement dans la chaîne `ch` et qui renvoie `False` sinon.
2. On veut écrire une fonction `triple_six_exact` prenant en argument une chaîne `ch` et qui renvoie `True` si `ch` contient la sous-chaîne `'666'` mais pas la sous-chaîne de `'6666'` et qui renvoie `False` sinon.

   Compléter la fonction suivante, qui distingue les cas où les trois 6 consécutifs sont :
   - au début de la chaîne `ch` ;
   - à la fin de la chaîne `ch` ;
   - ni au début ni à la fin de la chaîne `ch`.

```
def triple_six_exact(ch):
  n = len(ch)
  if ch[0:3] == "666" and ..........:
    return True
  if ch[n-3:n] == "666" and ..........:
    return True
  for k in range(1, ......):
    if ch[k:k+3] == "666" and .......... and ..........:
      return True
  return False
```

Exercice 13.

1. On souhaite écrire une fonction `miroir` qui prend en argument une chaîne `ch` et qui renvoie la même chaîne mais écrite à l'envers. Ainsi, `miroir('abcd')` renvoie `'dcba'`.
   Parmi les quatre fonctions suivantes, une seule convient. Déterminer laquelle.

```
def miroir1(ch):
  inv = ''
  for e in ch:
    inv = inv + e
  return inv
```

```
def miroir2(ch):
  inv = ''
  for e in ch:
    inv = e + inv
  return inv
```

```
def miroir3(ch):
  inv = ''
  n = len(ch)
  for k in range(1,n):
    inv = ch[k] + inv
  return inv
```

```
def miroir4(ch):
  inv = ''
  n = len(ch)
  for k in range(1,n):
    inv += ch[k]
  return inv
```

2. Un palindrome est une chaîne de caractère qui donne le même résultat qu'on la lise de la gauche vers le droite ou de la droite vers la gauche. Par exemple, `'non'`, `'kayak'` ou `'etlamarinevaveniramalte'` sont des palindromes.
   Écrire une fonction `palindrome` prenant en argument une chaîne de caractère `ch` et qui renvoie `True` si `ch` est un palindrome et `False` sinon.
3. On souhaite modifier la fonction précédente afin qu'elle ne tienne pas compte des espaces et qu'elle considère que la chaîne `'et la marine va venir a malte'` est aussi un palindrome.
   a. Écrire une fonction `sans_espace` prenant en argument une chaîne de caractères `ch`, qui supprime tous les espaces de `ch` et qui renvoie la chaîne obtenue.
   b. Écrire une fonction `palindrome2` répondant à la question.

II. — Listes

1) Définition

Une liste est une suite ordonnée d'objets, qui peuvent être de types différents, qui est délimitée par des crochets et dont les éléments sont séparés par des virgules.

Par exemple, `[3.14, 'pi', 3]` est une liste contenant le flottant `3.14`, la chaîne de caractères `'pi'` et l'entier `3`.

La liste vide, qui ne contient aucun élément, se définit par `[]`.

Une liste possède un type propre en Python : le type *list* (`list`). Il s'agit d'un objet *mutable* ce qui signifie qu'on peut modifier ou supprimer des éléments d'une liste.

Les éléments d'un liste peuvent eux-même être des listes : on parle dans ce cas de liste de listes. Par exemple, `[[1,2,-1],[3,2,1],[6,3]]` est une liste de listes d'entiers.

Il existe différentes façons de définir une liste en Python :

- **par énumération**, comme nous l'avons fait précédemment, en écrivant explicitement les éléments de la liste : `L = [2,4,'a',2.4,3]` ;
- **en compréhension**, c'est-à-dire à l'aide d'une instruction conditionnelle (`if`) ou d'une boucle (`for` ou `while`) permettant de décrire tous les éléments de la liste dans l'ordre. Par exemple, la liste des entiers pairs de 0 à 10 peut s'obtenir de la manière suivante :

  `L = [2*k for k in range(6)]`.

- **par répétition** si on souhaite créer une liste dont tous les éléments sont identiques. Par exemple, `[1]*10` crée une liste contenant 10 éléments, tous égaux à `1`.

2) Longueur et éléments d'une liste

Comme pour les chaînes de caractères, la longueur d'une liste est son nombre d'éléments et on l'obtient à l'aide de la fonction `len`.

La notion d'indice valable et la gestion des éléments d'une liste sont identiques à celles des chaînes.

Ainsi, si `L = [2, "abc", -1.1, [3,1]]` alors `len(L)` renvoie 4, `L[1]` renvoie la chaîne de caractères `"abc"` et `L[-1]` renvoie la liste `[3,1]`. Dès lors, `L[1][2]` renvoie le caractère `"c"` et `L[-1][0]` renvoie l'entier `3`.

3) Opérations sur les listes

La syntaxe et le fonctionnement de l'extraction de sous-liste, de la concaténation, de la répétition et du test d'appartenance sont les mêmes pour les listes que pour les chaînes de caractères. **Attention**, cependant, le test d'appartenance ne fonctionne que pour des éléments et pas pour des sous-listes. Ainsi, `[1,2] in [1,2,3]` ou `[2] in [1,2,3]` renvoie `False` alors que `2 in [1,2,3]` renvoie `True`.

Le type `list` étant mutable, on dispose, de plus, d'opérations qui modifient la liste initiale :

1. **modification d'un élément** : on peut remplacer l'élément d'indice i d'une liste `L` par la valeur `v` grâce à la syntaxe `L[i] = v`.

   Par exemple, si `L = [3, "c", 1.1]` alors la syntaxe `L[1] = -2` modifie la liste `L` qui devient `[3, -2, 1.1]`.

2. **ajout d'un élément à une liste** : on peut modifier une liste `L` en lui ajoutant un élément `e` en utilisant

- `L.append(e)` qui ajoute l'élément `e` à la fin de la liste `L`. Par exemple, si `L = [2,4,6]` alors `L.append(8)` ajoute la valeur 8 à la liste `L` et ainsi la nouvelle valeur de `L` est `[2,4,6,8]`.

  Une autre syntaxe possible est `L = L + e` ou `L += e` mais il est préférable d'utiliser la méthode `append` car l'opération `+` en Python désigne déjà la concaténation des listes et cela peut être source d'erreur si l'élément `e` est lui-même une liste.
- `L.insert(i,e)` qui ajoute l'élément `e` à l'indice i dans la liste, ce qui a pour effet de décaler les éléments suivants.

  Par exemple, si `L = [2,4,6]` alors `L.insert(2,8)` ajoute la valeur 8 à la liste `L` de telle sorte que 8 devienne l'élément d'indice 2 de `L`. Ainsi, la nouvelle valeur de `L` est `[2,4,8,6]`.

3. **ajout simultané d'éléments à la fin d'une liste** : si `L1` et `L2` sont deux listes, on peut modifier `L1` en lui ajoutant les éléments de `L2` à la fin grâce à la méthode `.extend` : `L1.extend(L2)`.

   Ainsi, si `L1 = [2,4,6]` et `L2 = [1,3,5]` alors `L1.extend(L2)` modifie la liste `L1` qui devient `[2,4,6,1,3,5]`.
4. **suppression d'un élément d'une liste** : on peut supprimer un élément dans une liste `L` en utilisant :
   - `L.pop()` pour supprimer le dernier élément de la liste `L`.
   - `L.pop(i)` pour supprimer l'élément d'indice i de la liste `L`.

   Par exemple, si `L = [2,4,6]` alors `L.pop()` supprime la valeur 6 de la liste `L` et ainsi la nouvelle valeur de `L` est `[2,4]` et `L.pop(1)` supprime la valeur 4 de la liste `L` et ainsi la nouvelle valeur de `L` est `[2,6]`.

4) Itération sur les éléments d'une liste

Comme pour les chaînes de caractères, il est possible d'itérer une action sur les éléments d'une liste à l'aide d'une instruction conditionnelle et/ou d'une boucle.

Par exemple, si on souhaite écrire une fonction `somme_liste` qui prend en argument une liste `L` dont les éléments sont des nombres et qui renvoie la somme des éléments de `L`, plutôt que d'écrire

```
def somme_liste(L):
    n = len(L)
    S = 0
    for k in range(n):
        S += L[k]
    return S
```

il est préférable d'écrire

```
def somme_liste(L):
    S = 0
    for e in L:
        S += e
    return S
```

5) Exercices

Exercice 14. On considère la liste suivante :

```
L = [1,2.3,4.1,4,2.55,5,23]
```

Les instructions suivantes sont-elles correctes et, si oui, que renvoient-elles ?

1. `L[1]`	**2.** `L[-2]`	**3.** `L[7]`	**4.** `len(L)`
5. `L[1.5]`	**6.** `L[1,5]`	**7.** `L[1:5]`	**8.** `L[0:4:2]`
9. `L[0]+L[3]`	**10.** `[L[0]]+[L[3]]`	**11.** `3*L[0]`	**12.** `3*[L[0]]`
13. `L[2] == 2.3`	**14.** `L[-3] == 2.55`	**15.** `4 in L`	**16.** `[4.1,4] in L`

Exercice 15. On considère la liste suivante :

```
M = [[1,-2], [0.5,6], [-1,3.45]].
```

Les instructions suivantes sont-elles correctes et, si oui, que renvoient-elles ?

1. `M[0]` **2.** `M[0][1]` **3.** `M[0][1][2]` **4.** `len(M)` **5.** `len(M[0])` **6.** `M[0]+M[2]`.

Exercice 16. On souhaite écrire une fonction `affiche` qui affiche en colonne les entiers de 0 à 3. Parmi les quatre fonctions suivantes, laquelle ou lesquelles convien(nen)t ?

```
def affiche1():
  for i in [0, 3]:
    print(i)
```

```
def affiche2():
  for j in [0, 1, 2, 3]:
    print(j)
```

```
def affiche3():
  for k in range(3):
    print(k)
```

```
def affiche4():
  for m in range(4):
    print(m)
```

Exercice 17.

1. Compléter les programmes suivants pour qu'ils affichent la liste `L` de tous les entiers de 1 à 100.

```
# Programme 1
L = []
for k in range(...):
  L += ...
print(L)
```

```
# Programme 2
L = []
for k in range(...):
  L.append(...)
print(L)
```

```
# Programme 3
L = []
L = 100*[0]
for k in range(...):
  L[k] = ...
print(L)
```

2. On considère la liste `L` de la question précédente. Écrire un programme qui multiplie par 2 tous les éléments de la liste `L` et affiche la nouvelle liste obtenue.
3. Écrire une fonction `carre` prenant en argument un entier $n > 0$ et qui renvoie la liste des carrés des entiers de 1 à n. Par exemple, `carre(5)` doit renvoyer `[1, 4, 9, 16, 25]`.
4. En utilisant la définition en compréhension, écrire un programme qui affiche une liste dont les éléments sont les doubles de tous les entiers de 1 à 100 puis une fonction `carre_compr` prenant en argument un entier $n > 0$ et qui renvoie la liste des carrés des entiers de 1 à n.

Exercice 18. Pour chacune des listes suivantes, donner une syntaxe Python permettant de la définir en compréhension.

1. `[1,3,5,7,9,11,13,15,17,19]`
2. `[0,1,4,9,16,25,36,49,64,81,100]`
3. `[1,2,4,8,16,32,64,128,256,512,1024]`
4. `[3,6,12,15,21,24,30,33,39,42,48,51]`

Exercice 19. Écrire une fonction `indices` qui prend en argument une chaîne de caractères `ch` et un caractère `c` et qui renvoie la liste, éventuellement vide, des indices de la chaîne `ch` où apparaît le caractère `c`. Ainsi, `indices('abbacccaa','a')` renvoie `[0,3,7,8]` et `indices('abbacccaa','d')` renvoie `[]`.

Exercice 20. Écrire une fonction `sous_liste` prenant en argument deux listes `L1` et `L2` et qui renvoie `True` si `L2` est une sous-liste de `L1` (c'est-à-dire si tous les éléments de `L2` apparaissent consécutivement et dans le même ordre dans `L1`) et qui renvoie `False` sinon.

Exercice 21. Écrire une fonction `tous_positifs` qui prend en argument une liste non vide de nombres `L` et qui renvoie `True` si tous les nombres de la liste sont positifs ou nuls et `False` sinon.

Exercice 22.

1. Écrire une fonction `positifs` prenant en argument une liste non vide d'entiers `L` et qui renvoie le nombre d'éléments de `L` qui sont positifs ou nuls.
   Par exemple, `positifs([2, -1, 3, 4, -6, 0, 6])` doit renvoyer 5.
2. Écrire une fonction `rangs_negatifs` prenant en argument une liste non vide d'entiers `L` et qui renvoie la liste des indices des éléments strictement négatifs de `L`.
   Par exemple, `rangs_negatifs([2, -1, 3, 4, -6, 0, 6])` doit renvoyer `[1, 4]`.
3. Écrire une fonction `motif` prenant en argument une liste non vide d'entiers `L` et qui renvoie `True` si `[0, 0]` est une sous-liste de `L` et `False` sinon.
4. Écrire une fonction `dix_vingt` prenant en argument une liste non vide d'entiers `L` et qui renvoie `True` si tous les éléments de `L` sont compris entre 10 et 20 (avec égalité possible) et `False` sinon.

Exercice 23.

1. On veut écrire une fonction `croissante` prenant en argument une liste de réels `L` et qui renvoie `True` si chaque élément de la liste est inférieur ou égal au suivant et `False` sinon.
   Recopier, en la corrigeant, la fonction proposée ci-dessous qui comporte des erreurs.

```
def croissante(L):
  n = len(L)
  for k in range(0, n):
    if L[k] > L[k+1]:
      return False
    else:
      return True
```

2. Écrire une fonction `monotone` prenant en argument une liste de réels `L` et qui renvoie `True` si la suite des termes de `L` est monotone (c'est-à-dire croissante ou décroissante) et `False` sinon.

Exercice 24. Expliquer le rôle de la fonction `nb6` suivante :

```
from random import *

def mystere(n):
  c=0
  L=[randint(1,6) for k in range(n)]
  for j in L:
    if (j==2):
      c=c+1
  return c
```

Exercice 25. Écrire une fonction `suite` qui prend en argument un entier naturel n et qui renvoie la liste des n premières valeurs de la suite (u_n) définie par $u_0 = 1$ et, pour tout $n \in \mathbb{N}$, $u_{n+1} = u_n^2 + n$.

Exercice 26. Écrire une fonction `valeur_absolue_liste` qui prend en argument une liste d'entiers `L` et remplace chaque élément de la liste par sa valeur absolue.

Exercice 27. Écrire une fonction `E_et_sigma` qui prend en arguments deux listes : la liste des valeurs d'une variable aléatoire X et la liste des probabilités associées (dans le même ordre, évidemment) et qui renvoie l'espérance et l'écart-type de X.