Considérons une **chaîne de caractères** contenant un texte formé de phrases se terminant par un point ordinaire, un point d'exclamation, ou un point d'interrogation ; exemple :

```
s = "Il fait très beau ! Vais-je sortir ? J'hésite encore."
```

1. Sachant que le texte ne contient pas le caractère '*', quel est l'intérêt du code suivant ?

```
ponc = ['.', '!', '?']
for p in ponc:
    s = s.replace(p+' ', p+'*')
```

2. Construire la liste des phrases du texte puis faire afficher ces phrases ainsi :

```
Il fait très beau !
Vais-je sortir ?
J'hésite encore.
```

```
sl = s.split('*')
for phrase in sl:
    print(phrase)
```

3. Étant donnée une liste de phrases, construire la chaîne de caractères contenant le texte formé par la suite de ces phrases (comme celui donné en exemple au début dans s).

```
s2 = ''
for phrase in sl:
    s2 = s2 + phrase + ' '  espace
s2 = s2[:-1]  # pour enlever espace en trop à la fin.
```

Mélange d'une liste. La bibliothèque *random* de Python contient entre autres les méthodes :
- `random.randint(start, stop)` qui renvoie aléatoirement un entier entre *start* et *stop*, ces deux valeurs étant incluses dans les tirages possibles ;
- `random.shuffle(list)` qui mélange les éléments de la liste donnée en argument ; NB : la méthode modifie la liste d'origine, elle ne renvoie pas une copie.

exemple :

```
import random as rd
L = list(range(10))
rd.shuffle(L) ; print(L)
```

peut renvoyer :

```
[6, 9, 8, 4, 1, 0, 7, 2, 3, 5]
```

4. **Copie** : soit L une liste de nombres ; parmi les instructions suivantes, entourer celle ou celles qui crée(nt) une copie de L indépendante, pouvant être ensuite modifiée sans affecter L :

(1) ~~L2 = L~~ (copie d'étiquette) (2) `L2 = L[:]` (3) `L2 = [e for e in L]`

5. Plutôt que d'utiliser la méthode *shuffle*, on définit une fonction *melange* qui effectue n fois l'échange de deux éléments $L[i]$ et $L[j]$ pris au hasard dans une "vraie copie" de la liste donnée en argument (la fonction renvoie la copie mélangée).
Compléter pour ce faire le script suivant (on rappelle la possibilité en Python d'affectation en parallèle : `a,b = new_a,new_b`.

```
def melange(L:list , n:int):
    L2 = L[:]
    for k in range(n):
        i = rd.randint(0,len(L)-1)
        j = rd.randint(0,len(L)-1)
        L2[i],L2[j] = L2[j],L2[i]
    return L2
```

6. **Appels récursifs.** On souhaite compter le nombre de fois qu'une fonction récursive (non précisée) s'appelle elle-même. Expliquez pourquoi l'un de ces scripts convient, et pas l'autre :

```
def foncrec(...):
    c = 0
    if (fin de récursivité):
        return ...
    else:
        c += 1
        return foncrec(...)
```

remise à zéro à chaque appel de la fonction, donc ne permet pas de compter ces appels !

```
c = 0  une seule initialisation du compteur
def foncrec(...):
    global c  pour que la fonction utilise cette variable définie globalement dans la session de travail.
    if (fin de récursivité):
        return ...
    else:
        c += 1  incrémentation à chaque appel
        return foncrec(...)
```

↳ cette méthode convient.

La **suite de Fibonacci** (F_n) est définie par $F_0 = 0, F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$; elle commence donc par 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144...

7. Coder récursivement ci-contre la fonction `fibo1(n)`, renvoyant F_n (sans contrôle sur la valeur de n).

```
def fibo1(n:int):
    if n==0:
        return 0
    elif n==1:
        return 1
    else
        return fibo1(n-1)+fibo2(n-2)
```

8. Justifier que la complexité temporelle soit plutôt polynomiale ou plutôt exponentielle.

F(0) et F(1) → un seul appel de la fon, sinon deux appels à chaque "tour" donc le nb d'appel augmente en 2^n et la complexité est exponentielle.

9. Coder ci-dessous une instruction supplémentaire, de type `assert`, qui vérifiera que la valeur de n envoyée à la fonction est de type entier (type 'int'), et de valeur positive ou nulle; un message d'erreur explicite sera prévu si ce n'est pas le cas.

```
assert type(n)==int and n>=0, "n : entier naturel"
```

10. Coder non récursivement, et sans appeler `fibo1`, la fonction `fibo2(p)` renvoyant la liste des nombres F_n tels que $F_n \leq p$, avec $p \geq 1$.

```
def fibo2(p:int):
    assert type(p)==int and p>=1, "p:entier naturel>0"
    lf=[0]
    a=0; b=1  # ou a,b = 0,1
    while b<=p:
        lf.append(b)
        a,b = b,a+b
    return lf
```

```
# code + simple mais un peu moins clair
b=1
while b<=p:
    lf.append(b)
    b += lf[-2]  # l'avant-dernier élt
return lf
```