TP 1 : Corrigé partiel

Exercice 0.

1.
```
from math import factorial, e
s=0
for k in range(101): #de 0 à 100
    s+=(-1)**(k)/factorial(k) #s+= est un raccourci pour s=s+
```

2. On suppose N et b affectées. L'algorithme n'est qu'une variante de l'algorithme du cours qui extrait les chiffres dans la base b : ici il suffit d'incrémenter un compteur au lieu de calculer un nouveau chiffre.

```
c=0
while N>0:
    N=N//b
    c+=1
#ici c contient le nombre de chiffres de N dans la base b
```

Exercice 1. *Création de listes.* ***1.*** Liste constituée de 10 zéros, quelques possibilités :

```
L0 = [0] * 10
L0 = [0 for i in range(10)]

L0 = []
for i in range(10):
    L0.append(0)

L0 = []
while len(L0)<10:
    L0.append(0)
```

2. Liste des entiers de 0 à 99 :

```
L100 = list(range(100))
L100 = [i for i in range(100)]

L100 = []
for i in range(100):
    L100.append(i)

L100 = []
i=0
while len(L100)<100:
    L100.append(i)
    i+=1
```

Remarque : éviter les boucles `while` lorsque c'est possible. Une erreur peut mener à une boucle infinie !

3. Liste `L300` contenant `[0, 0, 0, 1, 1, 1, 2,...]` jusqu'à 99 répété trois fois :

```
L300 = []
for i in range(100):
    for j in range(3):
        L300.append(i)
```

On pouvait aussi écrire trois fois `L300.append(i)`, mais cette version est plus « générique ».

4. Liste contenant les entiers de 0 à 99 puis de 98 à 0 :

```
L = []
for i in range(100):
    L.append(i)
for i in range(99, -1, -1):
    L.append(i)
```

Autre possibilité avec la concaténation : `L = list(range(100))+list(range(99, -1, -1))`.

5. Liste des puissances de 2, de 2^0 à 2^{15} inclus :

```
[2**i for i in range(16)]
```

6. Liste des entiers positifs impairs inférieurs ou égaux à 100 divisibles par 5 ou par 7, mais pas par 35.

```
[i for i in range(1,100,2) if (i%5==0 or i%7==0) and i%35!=0]
```

Exercice 2. *Les deux algorithmes du cours pour passer d'une base à une autre.* On répète le cours ici. On passe d'un entier $N = \sum_{k=0}^{n-1} a_k b^k$ à la liste $[a_0, a_1, \dots, a_{n-1}]$ et réciproquement. Dans le premier algorithme, on suppose `N` et `b` affectés.

```
M = N
L = []
while M>0:
    L.append(M%b)
    M=M//b
```

Réciproquement, avec `L` et `b` affectés, voici comment stocker $\sum a_i b^i$ dans `s` :

```
s=0
for i in range(len(L)):
    s+=L[i]*b**i #s+= raccourci pour s=s+
```

Exercice 3. *Parcours de listes.* Les quatre codes suivants calculent la somme des éléments de `L`, le maximum, l'indice du maximum, et un booléen indiquant si `L` est croissante.

```
somme_liste=0
for i in range(len(L)):
    somme_liste+=L[i]
```

```
indice_max_liste=0
for i in range(1,len(L)):
    if L[i]>L[indice_max_liste]:
        indice_max_liste=i
```

```
max_liste=L[0]
for i in range(1,len(L)):
    if L[i]>max_liste:
        max_liste=L[i]
```

```
est_croissante=True
for i in range(len(L)-1):
    if L[i+1]<L[i]:
       est_croissante=False
```

Pour le maximum (ou son indice), l'idée est de stocker dans une variable le maximum temporaire vu pour l'instant (ou son indice). Lorsqu'on examine un nouvel élément, on le compare à ce maximum temporaire, qui change de valeur si nécessaire. Remarque : ces deux quantités peuvent être directement obtenues en Python comme suit.

```
max_liste=max(L)
indice_max_liste=L.index(max_liste)
```

Pour la croissance, on fait parcourir à un indice i tous les entiers entre 0 et $n-2$, où n est la taille de la liste, puisqu'on compare `L[i]` et `L[i+1]`. La croissance se traduit par le fait que `L[i]` $\leq$ `L[i+1]` pour tout $i \in [\![0, n-2]\!]$. Ainsi une liste n'est *pas* croissante s'il existe un indice i tel que `L[i]` $>$ `L[i+1]`. C'est ce que regarde le code.

Exercice 4. Test naïf de primalité. En supposant `p` affecté, le code suivant stocke dans la variable `premier` un booléen indiquant si `p` premier (on le suppose supérieur ou égal à 2).

```
est_premier = True
for i in range(2,p):
    if p%i == 0:
        est_premier = False
        break #permet d'arrêter tout de suite la boucle : inutile de continuer !
```