Bonnes pratiques de programmation **Examen du 31 mai 2017** GB3, 2^{nd} semestre 2016-2017

Durée = 2h15. Langage Python.
Inscrivez **lisiblement** vos Nom et Prénom en tête de la copie double qui vous a été distribuée ;
ensuite vous pouvez cacheter le coin de la copie double ;
enfin inscrivez EN GROS CHIFFRES sur la copie double le numéro suivant : nUmErO

Vous devez inscrire vos réponses directement sur cette feuille d'énoncé, que vous placerez à l'intérieur de la copie double avant de la rendre.

Ordinateurs, téléphones et autres moyens de communication sont interdits.

Exercice 1 : En supposant avoir défini au préalable la liste `S = ["lun","mar","mer","jeu","ven","sam","dim"]` (de longueur 7), chacun des 4 morceaux de programme suivants contient une erreur. Corrigez-la directement sur la feuille.

```
def voir(n):
  if (n<0) == "False":
    print("Pas glop!")
  else:
    print(S[n % 7])
```

```
def valJours(d):
  r=0
  for e in d :
    if d[e] in S:
      r = r + [e]
  return(sorted(r))
```

```
def leJour(n):
  if (n < 1) or (n > 7) :
    print("incorrect")
  else:
    return(S[n])
```

```
def jourSemaine(n):
  if n < 0 :
    print("Nombre positif SVP!")
  else :
    print( n % 7 )

jour = S[jourSemaine(n)]
```

▷ `True sans ""...` `r=[] de type list...` `return(S[n-1]) ...` `return(n % 7) au lieu de print`

Exercice 2 : Écrivez une fonction `estRectangle` qui prend en entrées 3 nombres flottants `a`, `b` et `c` et qui dit s'ils forment un triangle rectangle ayant `c` pour hypoténuse. Indiquez quel est le type du résultat avant de programmer la fonction.

```
▷ ♯ type du résultat: bool
▷ def estRectangle(a,b,c):
▷     return ( a*a + b*b == c*c )
```

Exercice 3 : Écrivez une fonction `difference` qui prend en entrée une chaîne de caractères `b` représentant un brin d'ADN et qui calcule combien `b` possède de `G` ou `C` en plus que de `A` ou `T` : les brins riches en `G/C` donneront un résultat positif alors que ceux pauvres en `G/C` donneront un résultat négatif. Par exemple `difference("AAATTTGGCC")` retourne `-2`.

```
▷ def difference(b):
▷     r = 0
▷     for n in b :
▷       if n in "GC":
▷         r = r + 1
▷       else :
▷         r = r - 1
▷     return(r)
```

Exercice 4 : Écrivez une fonction `approxCompl` qui prend en entrée deux brins d'ADN `x` et `y` de 3' en 5' et retourne un booléen qui dit s'ils sont complémentaires, en autorisant jusqu'à 5% de mésappariement. On n'admet pas de décalage dans cet exercice, donc `x` et `y` doivent être exactement de même longueur (retournez `False` sinon). Par exemple `"AGAAATTTTTGGGGGCCCCC"` et `"GGGGGCCCCCAAAAATTTTT"` sont approximativement complémentaires : seul le second nucléotide du premier brin n'est pas complémentaire de l'avant dernier du second, et ces brins faisant 20 nucléotides on n'a que 5% d'erreur. On utilisera (sans le redéfinir) le dictionnaire `C = {"A":"T" , "T":"A" , "G":"C" , "C":"G"}`.

```
▷ def approxCompl(x,y) :
▷     n = len(x)
▷     if len(y) != n :
▷       return(False)
▷     m = 0
▷     i = 0
▷     while i < len(x) :
▷       if C[x[i]] != y[ n - i - 1 ] :
▷         m = m + 1
▷       i = i + 1
▷     return ( m <= (n / 20) )
```

T.S.V.P. ⟶

Dans les exercices suivants, on considère des dictionnaires dont les éléments sont des noms et la donnée associée à chaque nom est un chaîne de caractères qui représente un brin d'ADN. On peut ainsi par exemple avoir un dictionnaire qui représente des brins d'ADN issus d'une espèce donnée de bactéries :

```
exemple = { "ex01":"AGAAATTTTTGGGGGCCCCC" , "ex02":"GGGGGCCCCCAAAAATTTTT" , "ex03": "GATGCTGTCATGAAA" , ... }
```

On souhaite *in fine* extraire d'un dictionnaire les couples de ses brins qui sont approximativement complémentaires au sens de l'exercice précédent.

Exercice 5 : Dans un premier temps on doit charger le dictionnaire à partir d'un fichier généré par un robot. Un tel fichier contient deux lignes pour chaque brin : la première avec le nom du brin et la seconde avec le brin lui-même.
L'exemple ci-contre correspondrait au dictionnaire donné plus haut.

```
ex01
AGAAATTTTTGGGGGCCCCC
ex02
GGGGGCCCCCAAAAATTTTT
ex03
GATGCTGTCATGAAA
...
```

Écrivez une fonction `charger` qui prend en entrée une adresse de fichier à lire et retourne le dictionnaire correspondant.

```
▷ def charger(f) :
▷     d = {}
▷     r = open(f)
▷     nom = f.readline()
▷     brin = f.readline()
▷     while nom != "" and brin != "" :
▷        d[nom[:-1]] = brin[:-1]    ♯ supprime dernier \n
▷        nom = f.readline()
▷        brin = f.readline()
▷     f.close()
▷     return(d)
```

Exercice 6 : Écrivez une fonction `listNoms` qui prend en entrée un dictionnaire de brins et fournit en sortie la liste triée des noms des brins qu'il contient.
Rappel : `sorted` prend en entrée une liste et fournit une liste triée contenant exactement les mêmes éléments.

```
▷ def listNoms(d) :
▷    l = []
▷    for n in d :
▷      l = l + [n]
▷    return(sorted(l))
```

Exercice 7 : Écrivez une fonction `listCompl` qui prend en entrée un dictionnaire de brins et fournit en sortie la liste des couples de noms de ce dictionnaire dont les brins associés sont approximativement complémentaires. On pourra naturellement utiliser toutes les fonctions des exercices précédents (même si l'on n'a pas réussi à les résoudre).
Attention : il ne faut pas écrire deux fois les couples complémentaires. Pour notre dictionnaire `exemple` fourni plus haut, la liste devra naturellement contenir le couple `(ex01,ex02)` puisqu'on a vu que leurs brins sont approximativement complémentaires mais il ne faudra pas y inclure le couple `(ex02,ex01)` ! En revanche, rien n'interdit qu'un brin soit approximativement complémentaire de lui-même.

```
▷ def listCompl(d):
▷    l = listNoms(d)
▷    r = []
▷    i = 0
▷    while i < len(l) :
▷      for n in l[i:] :
▷        if approxCompl ( d[l[i]] , d[n] ) :
▷          r = r + [ (l[i],n) ]
▷      i = i + 1
▷    return(r)
```