

# Corrigés de la semaine 2

composite - Semaine 2 Séquence 3

```
1 # Pour calculer inconnue, on extrait une sous-chaine de composite
2 # qui commence a l'index len(connue)
3 # qui se termine a l'index len(composite)-len(connue)
4 # ce qui donne en utilisant une slice
5 inconnue = composite [ len(connue) : len(composite)-len(connue) ]
6 #
7 # on peut aussi faire encore plus simplement
8 inconnue = composite [ len(connue) : -len(connue) ]
```

divisible - Semaine 2 Séquence 7

```
1 def divisible(a, b):
2     "renvoie True si un des deux arguments divise l'autre"
3     # b divise a si et seulement si le reste
4     # de la division de a par b est nul
5     # et il faut regarder aussi si a divise b
6     return a%b==0 or b%a==0
```

```
1 def spam(l):
2     """
3     Prend en argument une liste, et retourne la liste modifiée:
4     * taille paire: on intervertit les deux premiers éléments
5     * taille impaire, on retire le dernier élément
6     """
7     # si la liste est vide il n'y a rien à faire
8     if not l:
9         pass
10    # si la liste est de taille paire
11    elif len(l)%2 == 0:
12        # on intervertit les deux premiers éléments
13        l[0], l[1] = l[1], l[0]
14    # si elle est de taille impaire
15    else:
16        # on retire le dernier élément
17        l.pop()
18    # et on n'oublie pas de retourner la liste dans tous les cas
19    return l
```

```
1 def multi_tri(listes):
2     "trie toutes les sous-listes, et retourne listes"
3     for liste in listes:
4         # sort fait un effet de bord
5         liste.sort()
6     # et on retourne la liste de départ
7     return listes
```

```

1 def multi_tri_reverse(listes, reverses):
2     """trie toutes les sous listes, dans une direction
3     précisée par le second argument"""
4     # zip() permet de faire correspondre les éléments
5     # de listes avec ceux de reverses
6     for liste, reverse in zip(listes, reverses):
7         # on appelle sort en précisant reverse=
8         liste.sort(reverse=reverse)
9     # on retourne la liste de départ
10    return listes

```

```

1 from math import e, pi
2
3 def liste_racines(p):
4     "retourne la liste des racines p-ièmes de l'unité"
5     # une simple compréhension fait l'affaire
6     # souvenez vous que 1j c'est notre 'i' complexe
7     return [e**((2*pi*1j*n)/p) for n in range(p)]
8
9 # Il est tout à fait possible aussi de construire les racines pas à pas
10 # C'est un peu moins élégant mais ça fonctionne très bien aussi
11 def liste_racines_bis(p):
12     "retourne la liste des racines p-ièmes de l'unité"
13     # on va construire le résultat petit à petit
14     # en partant d'une liste vide
15     resultat = []
16     # pour chaque n dans {0 .. p-1}
17     for n in range(p):
18         # on ajoute dans le résultat la racine d'ordre n
19         resultat.append(e**((2*pi*1j*n)/p))
20     # et on retourne le résultat
21     return resultat

```

```

1 def produit_scalaire(X,Y):
2     # initialisation du resultat
3     scalaire = 0
4     # ici encore avec zip() on peut faire correspondre
5     # les X avec les Y
6     for x,y in zip(X,Y):
7         scalaire += x*y
8     # on retourne le résultat
9     return scalaire
10
11 # Il y a plein d'autres solutions qui marchent aussi
12 # en voici notamment une qui utilise la fonction builtin sum
13 # (que nous n'avons pas encore vue, nous la verrons en semaine 4)
14 # en voici toutefois un avant-goût: la fonction sum est très pratique
15 # pour faire la somme de toute une liste de valeurs
16 def produit_scalaire_bis(X,Y):
17     """retourne le produit scalaire de deux listes de même taille"""
18     return sum([x*y for x, y in zip(X, Y)])

```

```
1  # un élève a remarqué très justement que ce code ne fait pas
2  # exactement ce qui est demandé, en ce sens qu'avec
3  # l'entrée correspondant à Ted Mosby on obtient A:><
4  # je préfère toutefois publier le code qui est en
5  # service pour la correction en ligne, et vous laisse
6  # le soin de l'améliorer si vous le souhaitez
7  def affichage(s):
8      # pour ignorer les espaces et les tabulations
9      # le plus simple est de les enlever
10     s=s.replace(' ', '').replace('\t','')
11     # la liste des mots séparés par une virgule
12     # nous est donnée par un simple appel à split
13     mots = s.split(',')
14     # si on n'a même pas deux mots, on retourne None
15     if len(mots) < 2:
16         return None
17     # maintenant qu'on sait qu'on a deux mots
18     # on peut extraire le prénom et le nom
19     prenom = mots.pop(0)
20     nom = mots.pop(0)
21     # on veut afficher "???" si l'âge est inconnu
22     age = "???"
23     # mais si l'âge est précisé dans la ligne
24     if len(mots) >= 2:
25         # alors on le prend
26         age = mots.pop(1)
27     # il ne reste plus qu'à formater
28     return "N:>{ }< P:>{ }< A:>{ }<".format(nom, prenom, age)
```

```
1 def carre(s):
2     # on enlève les espaces et les tabulations
3     s = s.replace(' ', '').replace('\t', '')
4     # la ligne suivante fait le plus gros du travail
5     # d'abord on appelle split() pour découper selon les ';'
6     # dans le cas où on a des ';' en trop, on obtient dans le
7     # résultat du split un 'token' vide, que l'on ignore
8     # ici avec le clause 'if token'
9     # enfin on convertit tous les tokens restants en entiers avec int()
10    entiers = [int(token) for token in s.split(";") if token]
11    # il n'y a plus qu'à mettre au carré, retraduire en strings,
12    # et à recoudre le tout avec join et ':'
13    return ":".join([str(entier**2) for entier in entiers])
```