

# Corrigés de la semaine 4

comptage - Semaine 4 Séquence 1

```
1 def comptage(in_filename, out_filename):
2     """
3     retranscrit le fichier in_filename dans le fichier out_filename
4     en ajoutant des annotations sur les nombres de lignes, de mots
5     et de caractères
6     """
7     # on ouvre le fichier d'entrée en lecture
8     # on aurait pu mettre open (in_filename, 'r')
9     with open(in_filename) as input:
10         # on ouvre la sortie en écriture
11         with open(out_filename, "w") as output:
12             # initialisations
13             lineno = 0
14             total_words = 0
15             total_chars = 0
16             # pour toutes les lignes du fichier d'entrée
17             for line in input:
18                 # on maintient le nombre de lignes
19                 # qui est aussi la ligne courante
20                 lineno += 1
21                 # autant de mots que d'éléments dans split()
22                 nb_words = len(line.split())
23                 total_words += nb_words
24                 # autant de caractères que d'éléments dans la ligne
25                 nb_chars = len(line)
26                 total_chars += nb_chars
27                 # on écrit la ligne de sortie; pas besoin
28                 # de newline (\n) car line en a déjà un
29                 output.write("{}: {}: {}:".format(lineno, nb_words, nb_chars).rstrip() + "\n")
30             # on écrit la ligne de synthèse
31             output.write("{}: {}:".format(lineno, total_words).rstrip() + "\n")
32             output.write("{}: {}".format(lineno, total_chars).rstrip() + "\n")
33
```

```

1 def pgcd(a, b):
2     "le pgcd de a et b par l'algorithme d'Euclide"
3     # l'algorithme suppose que a >= b
4     # donc si ce n'est pas le cas
5     # il faut inverser les deux entrées
6     if b > a :
7         a, b = b, a
8     # boucle sans fin
9     while True:
10        # on calcule le reste
11        r = a % b
12        # si le reste est nul, on a terminé
13        if r == 0:
14            return b
15        # sinon on passe à l'itération suivante
16        a, b = b, r

```

```

1 from operator import mul
2
3 def numbers(liste):
4     """
5     retourne un tuple contenant
6     (*) la somme
7     (*) le produit
8     (*) le minimum
9     (*) le maximum
10    des éléments de la liste
11    """
12
13    return (
14        # la builtin 'sum' renvoie la somme
15        sum(liste),
16        # pour la multiplication, reduce est nécessaire
17        reduce(mul, liste, 1),
18        # les builtin 'min' et 'max' font ce qu'on veut aussi
19        min(liste),
20        max(liste)
21    )

```

validation - Semaine 4 Séquence 3

```
1 def validation(f, g, entrees):
2     """
3     retourne une liste de booléens, un par entree dans entrees
4     qui indique si f(entree) == g(entree)
5     """
6     # on vérifie pour chaque entrée si f et g retournent
7     # des résultats égaux avec ==
8     # et on assemble le tout avec une comprehension de liste
9     return [f(entree) == g(entree) for entree in entrees]
```

aplatir - Semaine 4 Séquence 4

```
1 def aplatir(conteneurs):
2     "retourne une liste des éléments des éléments de conteneurs"
3     # on peut concaténer les éléments de deuxième niveau
4     # par une simple imbrication de deux compréhensions de liste
5     return [element for conteneur in conteneurs for element in conteneur]
```

alternat - Semaine 4 Séquence 4

```
1 def alternat(l1, l2):
2     "renvoie une liste des éléments pris un sur deux dans l1 et dans l2"
3     # pour réaliser l'alternance on peut combiner zip avec aplatir
4     # telle qu'on vient de la réaliser
5     return aplatir(zip(l1, l2))
```

alternat - Semaine 4 Séquence 4

```
1 def alternat2(l1, l2):
2     "une deuxième version de alternat"
3     # la même idée mais directement, sans utiliser aplatir
4     return [element for conteneur in zip(l1, l2) for element in conteneur]
```

```

1 def intersect(A, B):
2     """
3     prend en entrée deux listes de tuples de la forme
4     (entier, valeur)
5     renvoie la liste des valeurs associées dans A ou B
6     aux entiers présents dans A et B
7     """
8     # pour montrer un exemple de fonction locale:
9     # une fonction qui renvoie l'ensemble des entiers
10    # présents dans une des deux listes d'entrée
11    def values(S):
12        return {i for i, val in S}
13    # on l'applique à A et B
14    val_A = values(A)
15    val_B = values(B)
16    #
17    # bien sûr on aurait pu écrire directement
18    # val_A = {i for i, val in A}
19    # val_B = {i for i, val in B}
20    #
21    # les entiers présents dans A et B
22    # avec une intersection d'ensembles
23    common_keys = val_A & val_B
24    # et pour conclure on fait une union sur deux
25    # compréhensions d'ensembles
26    return {vala for a, vala in A if a in common_keys} \
27        | {valb for b, valb in B if b in common_keys}

```

```

1 import math
2
3 def distance(*args):
4     "la racine de la somme des carrés des arguments"
5     # avec une compréhension on calcule la liste des carrés des arguments
6     # on applique ensuite sum pour en faire la somme
7     # vous pourrez d'ailleurs vérifier que sum ([]) = 0
8     # enfin on extrait la racine avec math.sqrt
9     return math.sqrt(sum([x**2 for x in args]))

```

```

1 def doubler_premier(f, first, *args):
2     """
3     renvoie le résultat de la fonction f appliquée sur
4     f(2 * first, *args)
5     """
6     # une fois qu'on a écrit la signature on a presque fini le travail
7     # en effet on a isolé la fonction, son premier argument, et le reste
8     # des arguments
9     # il ne reste qu'à appeler f, après avoir doublé first
10    return f(2*first, *args)

```

```

1 def doubler_premier2(f, first, *args, **keywords):
2     """
3     équivalent à doubler_premier
4     mais on peut aussi passer des arguments nommés
5     """
6     # c'est exactement la même chose
7     return f(2*first, *args, **keywords)
8
9 # Complément - niveau avancé
10 # ----
11 # Il y a un cas qui ne fonctionne pas avec cette implémentation,
12 # quand le premier argument de f a une valeur par défaut
13 # *et* on veut pouvoir appeler doubler_premier
14 # en nommant ce premier argument
15 #
16 # par exemple - avec f=muln telle que définie dans l'énoncé
17 #def muln(x=1, y=1): return x*y
18
19 # alors ceci
20 #doubler_premier2(muln, x=1, y=2)
21 # ne marche pas car on n'a pas les deux arguments requis
22 # par doubler_premier2
23 #
24 # et pour écrire, disons doubler_permier3, qui marcherait aussi comme cela
25 # il faudrait faire une hypothèse sur le nom du premier argument...

```

```
1 def validation2(f, g, argument_tuples):
2     """
3     retourne une liste de booléens, un par entree dans entrees
4     qui indique si f(*tuple) == g(*tuple)
5     """
6     # c'est presque exactement comme validation, sauf qu'on s'attend
7     # à recevoir une liste de tuples d'arguments, qu'on applique
8     # aux deux fonctions avec la forme * au lieu de les passer directement
9     return [f(*tuple) == g(*tuple) for tuple in argument_tuples]
```