

# Tous les corrigés

## Table des matières

<code>composite</code> — Semaine 2 Séquence 3	3
<code>divisible</code> — Semaine 2 Séquence 7	3
<code>spam</code> — Semaine 2 Séquence 7	3
<code>multi_tri</code> — Semaine 2 Séquence 7	4
<code>multi_tri_reverse</code> — Semaine 2 Séquence 7	4
<code>liste_racines</code> — Semaine 2 Séquence 7	5
<code>produit_scalaire</code> — Semaine 2 Séquence 7	5
<code>affichage</code> — Semaine 2 Séquence 8	6
<code>carre</code> — Semaine 2 Séquence 8	7
<code>merge</code> — Semaine 3 Séquence 2	8
<code>merge</code> — Semaine 3 Séquence 2	9
<code>merge</code> — Semaine 3 Séquence 2	10
<code>diff</code> — Semaine 3 Séquence 3	11
<code>decode_zen</code> — Semaine 3 Séquence 5	12
<code>decode_zen</code> — Semaine 3 Séquence 5	13
<code>dispatch1</code> — Semaine 3 Séquence 7	13
<code>dispatch2</code> — Semaine 3 Séquence 7	14
<code>comptage</code> — Semaine 4 Séquence 1	14

pgcd — Semaine 4 Séquence 2	15
numbers — Semaine 4 Séquence 3	16
validation — Semaine 4 Séquence 3	17
aplatir — Semaine 4 Séquence 4	17
alternat — Semaine 4 Séquence 4	17
intersect — Semaine 4 Séquence 4	17
distance — Semaine 4 Séquence 8	18
doubler_premier — Semaine 4 Séquence 8	18
doubler_premier2 — Semaine 4 Séquence 8	19
validation2 — Semaine 4 Séquence 8	19
RPCProxy — Semaine 5 Séquence 6	20

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
```

multi\_tri\_reverse - Semaine 2 Séquence 7

```
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
```

liste\_racines - Semaine 2 Séquence 7

```
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)
```

carre - Semaine 2 Séquence 8

```
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])
```

index - Semaine 3 Séquence 2

```
1 def index(bateaux):
2     """
3     Calcule sous la forme d'un dictionnaire indexé par les ids
4     un index de tous les bateaux présents dans la liste en argument
5     Comme les données étendues et abrégées ont toutes leur id
6     en première position on peut en fait utiliser ce code
7     avec les deux types de données
8     """
9     # c'est une simple compréhension de dictionnaire
10    return { bateau[0]: bateau for bateau in bateaux}
```

index - Semaine 3 Séquence 2

```
1 def index2(bateaux):
2     """
3     La même chose mais ne manière itérative
4     """
5     # si on veut décortiquer
6     resultat = {}
7     for bateau in bateaux:
8         resultat [bateau[0]] = bateau
9     return resultat
```



```
1 def merge(extended, abbreviated):
2     """
3     Consolide des données étendues et des données abrégées
4     comme décrit dans l'énoncé
5     Le coût de cette fonction est linéaire dans la taille
6     des données (longueur des listes)
7     """
8     # on initialise le résultat avec un dictionnaire vide
9     result = {}
10    # pour les données étendues
11    for ship in extended:
12        # on affecte les 6 premiers champs
13        # et on ignore les champs de rang 6 et au delà
14        id, latitude, longitude, timestamp, name, country = ship[:6]
15        # on crée une entrée dans le résultat,
16        # avec la mesure correspondant aux données étendues
17        result[id] = [name, country, (latitude, longitude, timestamp)]
18    # maintenant on peut compléter le résultat avec les données abrégées
19    for id, latitude, longitude, timestamp in abbreviated:
20        # et avec les hypothèses on sait que le bateau a déjà été
21        # inscrit dans le résultat, donc on peut se contenter d'ajouter
22        # la mesure abrégée correspondant au bateau
23        result[id].append((latitude, longitude, timestamp))
24    # et retourner le résultat
25    return result
```

```
1 def merge2(extended, abbreviated):
2     """
3     Une deuxième version
4     """
5     # on initialise le résultat avec un dictionnaire vide
6     result = {}
7     # on remplit d'abord à partir des données étendues
8     for ship in extended:
9         id = ship[0]
10        # on crée la liste avec le nom et le pays
11        result[id] = ship[4:6]
12        # on ajoute un tuple correspondant à la position
13        result[id].append(tuple(ship[1:4]))
14    # pareil que pour la première solution,
15    # on sait d'après les hypothèses
16    # que les id trouvées dans abbreviated
17    # sont déjà présentes dans le resultat
18    for ship in abbreviated:
19        id = ship[0]
20        # on ajoute un tuple correspondant à la position
21        result[id].append(tuple(ship[1:4]))
22    return result
```

```

1 def merge3(extended, abbreviated):
2     """
3     Une troisième solution
4     """
5     # ici on va tirer profit du fait que les id sont
6     # en première position dans les deux tableaux
7     # aussi si on les trie on va mettre les deux tableaux 'en phase'
8     #
9     # c'est une technique qui marche dans ce cas précis
10    # parce qu'on sait que les deux tableaux contiennent des données
11    # pour exactement le même ensemble de bateaux
12    #
13    # ici on a deux choix, selon qu'on peut se permettre ou non de
14    # modifier les données en entrée. Supposons que oui:
15    extended.sort()
16    abbreviated.sort()
17    # si ça n'avait pas été le cas on aurait fait plutôt
18    # extended = extended.sorted() et idem pour l'autre
19    #
20    # il ne reste plus qu'à assembler le résultat
21    # en découpant des tranches et en les transformant en tuples
22    # lorsque c'est ce qui est demandé
23    return {
24        e[0] : e[4:6] + [ tuple(e[1:4]), tuple(a[1:4]) ]
25        for (e,a) in zip (extended, abbreviated)
26    }

```

```

1 def diff(extended, abbreviated):
2     """Calcule comme demandé dans l'exercice, et sous formes d'ensembles
3     (*) les noms des bateaux seulement dans extended
4     (*) les noms des bateaux présents dans les deux listes
5     (*) les ids des bateaux seulement dans abbreviated
6     """
7     # on n'utilise que des ensembles dans tous l'exercice
8     # les ids de tous les bateaux dans extended
9     # en utilisant une compréhension d'ensemble
10    extended_ids = {ship[0] for ship in extended}
11    # les ids de tous les bateaux dans abbreviated
12    # en utilisant une compréhension d'ensemble
13    abbreviated_ids = {ship[0] for ship in abbreviated}
14    # les ids des bateaux seulement dans abbreviated
15    # en utilisant la difference des ensembles
16    abbreviated_only_ids = abbreviated_ids - extended_ids
17    # les ids des bateaux dans les deux listes
18    # en utilisant l'intersection des ensembles
19    both_ids = abbreviated_ids & extended_ids
20    # les ids des bateaux seulement dans extended
21    # en utilisant la difference des ensembles
22    extended_only_ids = extended_ids - abbreviated_ids
23    # on recalcule les noms pour les deux catégories où c'est possible
24    # par une compréhension d'ensemble
25    both_names = \
26        {ship[4] for ship in extended if ship[0] in both_ids}
27    extended_only_names = \
28        {ship[4] for ship in extended if ship[0] in extended_only_ids}
29    # enfin on retourne les 3 ensembles sous forme d'un tuple
30    return extended_only_names, both_names, abbreviated_only_ids

```

```

1  # le module this est implémenté comme une petite énigme
2  # comme le laissent entrevoir les indices, on y trouve
3  # (*) dans l'attribut 's' une version encodée du manifeste
4  # (*) dans l'attribut 'd' le code à utiliser pour décoder
5  #
6  # ce qui veut dire qu'en première approximation on pourrait
7  # obtenir une liste des caractères du manifeste en faisant
8  #
9  # [ this.d [c] for c in this.s ]
10 #
11 # mais ce serait le cas seulement si le code agissait sur
12 # tous les caractères; comme ce n'est pas le cas il faut
13 # laisser intacts les caractères dans this.s qui ne sont pas
14 # dans this.d (dans le sens "c in this.d")
15 #
16 # je fais exprès de ne pas appeler l'argument this pour
17 # illustrer le fait qu'un module est un objet comme un autre
18 #
19
20 def decode_zen(this_module):
21     "décode le zen de python à partir du module this"
22     # la version encodée du manifeste
23     encoded = this_module.s
24     # le 'code'
25     code = this_module.d
26     # si un caractère est dans le code, on applique le code
27     # sinon on garde le caractère tel quel
28     # aussi, on appelle 'join' pour refaire une chaîne à partir
29     # de la liste des caractères décodés
30     return ''.join([code[c] if c in code else c for c in encoded])

```

```

1  # une autre version qui marche aussi, en utilisant
2  # dict.get(key, default)
3  def decode_zen2(this):
4      return "".join([this.d.get(c, c) for c in this.s])

```

```

1 def dispatch1(a, b):
2     """dispatch1 comme spécifié"""
3     # si les deux arguments sont pairs
4     if a%2 == 0 and b%2 == 0:
5         return a*a + b*b
6     # si a est pair et b est impair
7     elif a%2 == 0 and b%2 != 0:
8         return a*(b-1)
9     # si a est impair et b est pair
10    elif a%2 != 0 and b%2 == 0:
11        return (a-1)*b
12    # sinon - c'est que a et b sont impairs
13    else:
14        return a*a - b*b

```

```

1 def dispatch2(a, b, A, B):
2     """dispatch2 comme spécifié"""
3     # les deux cas de la diagonale \
4     if (a in A and b in B) or (a not in A and b not in B):
5         return a*a + b*b
6     # sinon si b n'est pas dans B
7     # ce qui alors implique que a est dans A
8     elif b not in B:
9         return a*(b-1)
10    # le dernier cas, on sait forcément que
11    # b est dans B et a n'est pas dans A
12    else:
13        return (a-1)*b

```

```

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
28                output.write("{}: {}: {}:{}\n".format(
29                    lineno, nb_words, nb_chars, line))
30            # on écrit la ligne de synthèse
31            output.write("{}: {}: {}{}\n".format(lineno, total_words, total_chars))

```

```

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

```

1 def intersect(A, B):
2     """
3     avec 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     # une fonction qui renvoie l'ensemble des entiers
9     # présent dans une des deux listes d'entrée
10    def values(S):
11        return {i for i, val in S}
12    # on l'applique à A et B
13    val_A = values(A)
14    val_B = values(B)
15    # les entiers présents dans A et B
16    # avec une intersection d'ensembles
17    common_keys = val_A & val_B
18    # et pour conclure on fait une union sur deux
19    # compréhensions d'ensembles
20    return {vala for a, vala in A if a in common_keys} \
21        | {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]))

```

doubler\_premier - Semaine 4 Séquence 8

```
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)
```

doubler\_premier2 - Semaine 4 Séquence 8

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

```

1 def validation2(f, g, argument_tuples):
2     """
3     retourne une liste de booleens, 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]

```

```

1 # une troisième implémentation de RPCProxy
2
3 class Forwarder(object):
4     def __init__(self, rpc_proxy, function):
5         self.function = function
6         self.rpc_proxy = rpc_proxy
7         # en rendant cet objet callable, on peut l'utiliser
8         # comme méthode dans RPCProxy
9     def __call__(self, *args):
10        print "Envoi à {} \nde la fonction {} -- args= {}".\
11            format(self.rpc_proxy.url, self.function, args)
12        return "retour de la fonction " + self.function
13
14 class RPCProxy(object):
15
16     def __init__(self, url, login, password):
17         self.url = url
18         self.login = login
19         self.password = password
20
21     def __getattr__(self, function):
22         """
23         Crée à la volée une instance de Forwarder
24         correspondant à 'function'
25         """
26         return Forwarder(self, function)
27

```