

MOOC Python

Corrigés de la semaine 5

multi_tri - Semaine 5 Séquence 2

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

multi_tri_reverse - Semaine 5 Séquence 2

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

doubler_premier - Semaine 5 Séquence 2

```
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_premier (bis) - Semaine 5 Séquence 2

```
1 def doubler_premier_bis(f, *args):
2     """
3     marche aussi mais moins élégant
4     """
5     first = args[0]
6     remains = args[1:]
7     return f(2*first, *remains)
```

doubler_premier_kwds - Semaine 5 Séquence 2

```
1 def doubler_premier_kwds(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_premier_kwds(muln, x=1, y=2)
21 # ne marche pas car on n'a pas les deux arguments requis
22 # par doubler_premier_kwds
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...
```

compare_all - Semaine 5 Séquence 2

```
1 def compare_all(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]
```

compare_args - Semaine 5 Séquence 2

```
1 def compare_args(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 compare, 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]
```

aplatir - Semaine 5 Séquence 3

```
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 5 Séquence 3

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

alternat (bis) - Semaine 5 Séquence 3

```
1 def alternat_bis(l1, l2):
2     """
3     une deuxième version de alternat
4     """
5     # la même idée mais directement, sans utiliser aplatir
6     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 keys(S):
12        return {k for k, val in S}
13    # on l'applique à A et B
14    keys_A = keys(A)
15    keys_B = keys(B)
16    #
17    # les entiers présents dans A et B
18    # avec une intersection d'ensembles
19    common_keys = keys_A & keys_B
20    # et pour conclure on fait une union sur deux
21    # compréhensions d'ensembles
22    return {vala for k, vala in A if k in common_keys} \
23           | {valb for k, valb in B if k in common_keys}

```

```

1 def produit_scalaire(X, Y):
2     """
3     retourne le produit scalaire
4     de deux listes de même taille
5     """
6     # on utilise la fonction builtin sum sur une itération
7     # des produits x*y
8     # avec zip() on peut faire correspondre les X avec les Y
9     # remarquez bien qu'on utilise ici une expression génératrice
10    # et PAS une compréhension car on n'a pas du tout besoin de
11    # créer la liste des produits x*y
12    return sum(x * y for x, y in zip(X, Y))

```

produit_scalaire (bis) - Semaine 5 Séquence 4

```
1 # Il y a plein d'autres solutions qui marchent aussi
2 #
3 def produit_scalaire_bis(X, Y):
4     """
5     Une autre version
6     """
7     scalaire = 0
8     for x, y in zip(X, Y):
9         scalaire += x * y
10    # on retourne le résultat
11    return scalaire
```

produit_scalaire (ter) - Semaine 5 Séquence 4

```
1 # Et encore une:
2 # celle-ci par contre est assez peu "pythonique"
3 # considérez-la comme un exemple de ce qu'il faut éviter
4 #
5 def produit_scalaire_ter(X, Y):
6     """
7     Un exemple de ce qu'il faut éviter de faire:
8     for i in range(len(iterable)):
9         x = iterable[i]
10    peut le plus souvent se remplacer par un
11    for x in iterable:
12        ...
13    """
14    scalaire = 0
15    # on calcule la taille
16    n = len(X)
17    # uniquement pour faire ce vilain idiome
18    for i in range(n):
19        scalaire += X[i] * Y[i]
20    return scalaire
```

```
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  # énumérer les caractères du manifeste en faisant
8  # (this.d[c] for c in this.s)
9  #
10 # mais ce serait le cas seulement si le code agissait sur
11 # tous les caractères; comme ce n'est pas le cas il faut
12 # laisser intacts les caractères de this.s qui ne sont pas
13 # dans this.d
14
15 def decode_zen(this_module):
16     """
17     décode le zen de python à partir du module this
18     """
19     # la version encodée du manifeste
20     encoded = this_module.s
21     # le dictionnaire qui implémente le code
22     code = this_module.d
23     # si un caractère est dans le code, on applique le code
24     # sinon on garde le caractère tel quel
25     # aussi, on appelle 'join' pour refaire une chaîne à partir
26     # de la liste des caractères décodés
27     return ''.join(code[c] if c in code else c for c in encoded)
```

```
1  # une autre version un peu plus courte
2  #
3  # on utilise la méthode get d'un dictionnaire, qui permet de spécifier
4  # (en second argument) quelle valeur on veut utiliser dans les cas où la
5  # clé n'est pas présente dans le dictionnaire
6  #
7  # dict.get(key, default)
8  # retourne dict[key] si elle est présente, et default sinon
9
10 def decode_zen_bis(this_module):
11     """
12     une autre version un peu plus courte
13     """
14     return "".join(this_module.d.get(c, c) for c in this_module.s)
```