MOOC Python

Corrigés de la semaine 5

multi_tri - Semaine 5 Séquence 2

```
def multi_tri(listes):
    """
    trie toutes les sous-listes
    et retourne listes
    """
    for liste in listes:
        # sort fait un effet de bord
        liste.sort()
    # et on retourne la liste de départ
    return listes
```

multi_tri_reverse - Semaine 5 Séquence 2

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

doubler_premier - Semaine 5 Séquence 2

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

doubler_premier (bis) - Semaine 5 Séquence 2

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

doubler_premier_kwds - Semaine 5 Séquence 2

```
def doubler_premier_kwds(f, first, *args, **keywords):
    """
    équivalent à doubler_premier
    mais on peut aussi passer des arguments nommés
    """
    # c'est exactement la même chose
    return f(2*first, *args, **keywords)

# Complément - niveau avancé
# ----
# Il y a un cas qui ne fonctionne pas avec cette implémentation,
# quand le premier argument de f a une valeur par défaut
# *et* on veut pouvoir appeler doubler_premier
# en nommant ce premier argument
#
# par exemple - avec f=muln telle que définie dans l'énoncé
#def muln(x=1, y=1): return x*y

# alors ceci
#doubler_premier_kwds(muln, x=1, y=2)
# ne marche pas car on n'a pas les deux arguments requis
# par doubler_premier_kwds
#
# et pour écrire, disons doubler_permier3, qui marcherait aussi comme cela
# il faudrait faire une hypothèse sur le nom du premier argument...
```

compare_all - Semaine 5 Séquence 2

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

compare_args - Semaine 5 Séquence 2

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

aplatir - Semaine 5 Séquence 3

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

alternat - Semaine 5 Séquence 3

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

alternat (bis) - Semaine 5 Séquence 3

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

intersect - Semaine 5 Séquence 3

```
def intersect(A, B):
    """
    prend en entrée deux listes de tuples de la forme
    (entier, valeur)
    renvoie la liste des valeurs associées dans A ou B
    aux entiers présents dans A et B
    """
    # pour montrer un exemple de fonction locale:
    # une fonction qui renvoie l'ensemble des entiers
    # présents dans une des deux listes d'entrée
    def keys(S):
        return {k for k, val in S}
    # on l'applique à A et B
    keys_A = keys(A)
    keys_B = keys(B)
    #
    # les entiers présents dans A et B
    # avec une intersection d'ensembles
    common_keys = keys_A & keys_B
    # et pour conclure on fait une union sur deux
    # compréhensions d'ensembles
    return {vala for k, vala in A if k in common_keys} \
         | {valb for k, valb in B if k in common_keys}
```

produit_scalaire - Semaine 5 Séquence 4

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

produit_scalaire (bis) - Semaine 5 Séquence 4

```
# Il y a plein d'autres solutions qui marchent aussi
def produit_scalaire_bis(X, Y):
    # initialisation du résultat
    scalaire = 0
    for x, y in zip(X, Y):
        scalaire += x * y
    # on retourne le résultat
    return scalaire
```

produit_scalaire (ter) - Semaine 5 Séquence 4

```
# et encore une: celle-ci par contre est assez peu "pythonique"
# je la donne plutôt comme un exemple de ce qu'il faut éviter
# on aime bien en général éviter les boucles du genre
# for i in range(len(iterable)):
#     ... iterable[i]
def produit_scalaire_ter(X, Y):
    scalaire = 0
    n = len(X)
    for i in range(n):
        scalaire += X[i] * Y[i]
    return scalaire
```

decode_zen - Semaine 5 Séquence 7

```
# le module this est implémenté comme une petite énigme
# comme le laissent entrevoir les indices, on y trouve
# (*) dans l'attribut 's' une version encodée du manifeste
# (*) dans l'attribut 'd' le code à utiliser pour décoder
#
# ce qui veut dire qu'en première approximation on pourrait
# obtenir une liste des caractères du manifeste en faisant
#
# [ this.d[c] for c in this.s ]
#
# mais ce serait le cas seulement si le code agissait sur
# tous les caractères; comme ce n'est pas le cas il faut
# laisser intacts les caractères de this.s qui ne sont pas
# dans this.d (dans le sens "c in this.d")
#
# je fais exprès de ne pas appeler l'argument this pour
# illustrer le fait qu'un module est un objet comme un autre

def decode_zen(this_module):
    "décode le zen de python à partir du module this"
    # la version encodée du manifeste
    encoded = this_module.s
    # le 'code'
    code = this_module.d
    # si un caractère est dans le code, on applique le code
    # sinon on garde le caractère tel quel
    # aussi, on appelle 'join' pour refaire une chaîne à partir
    # de la liste des caractères décodés
    return ''.join([code[c] if c in code else c for c in encoded])
```

decode_zen (bis) - Semaine 5 Séquence 7

```
# une autre version un peu plus courte
#
# on utilise la méthode get d'un dictionnaire, qui permet de spécifier
# (en second argument) quelle valeur on veut utiliser dans les cas où la
# clé n'est pas présente dans le dictionnaire
#
# dict.get(key, default)
# retourne dict[key] si elle eset présente, et default sinon

def decode_zen_bis(this_module):
    "une autre version plus courte"
    return "".join([this_module.d.get(c, c) for c in this_module.s])
```

decode_zen (ter) - Semaine 5 Séquence 7

```
# presque la même chose, mais en utilisant une expression génératrice
# à la place de la compréhension; la seule différence avec la version bis
# est l'absence des crochets carrés []
# ici je triche, nous n'avons pas encore vu ces expressions-là,
# nous les verrons en semaine 6, mais ça me permet de les introduire
# pour les curieux donc:
# avec ce code, **on ne crée pas la liste** qui est passée au join(),
# c'est comme si cette liste était cette fois
# parcourue à travers **un itérateur**
#
# on est donc un peu plus efficace - même si ça n'est évidemment
# pas très sensible dans ce cas précis

def decode_zen_ter(this_module):
    "une version avec une expression génératrice plutôt qu'une compréhension"
    return "".join(this_module.d.get(c, c) for c in this_module.s)
```