MOOC Python 3

Session 2018

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
```

tri_custom - Semaine 5 Séquence 2

```
def tri_custom(liste):
    """
    trie une liste en fonction du critère de l'énoncé
    """
    # pour le critère de tri on s'appuie sur l'ordre dans les tuples
    # c'est-à-dire
    # ((1, 2) <= (1, 2, 0) <= (1, 3) <= (2, 0)) == True
    # du coup il suffit que la fonction critère renvoie
    # selon la présence de p2, un tuple de 2 ou 3 éléments
    def custom_key(item):
        if 'p2' in item:
            return (item['p'], item['n'], item['p2'])
        return (item['p'], item['n'])
    liste.sort(key=custom_key)
    return liste
```

tri_custom_bis - Semaine 5 Séquence 2

```
def tri_custom_bis(liste):
    """
    tri avec une fonction lambda et une expression conditionnelle
    """
    # la même chose avec une lambda
    # l'expression conditionnelle est nécessaire ici, car
    # dans une lambda on est limité à des expressions
    liste.sort(key=lambda d: (d['p'], d['n'], d['p2'])
                             if 'p2' in d
                             else (d['p'], d['n']))
    return liste
```

tri_custom_ter - Semaine 5 Séquence 2

```
def tri_custom_ter(liste):
    """
    tri avec une fonction lambda et une compréhension de tuple
    """
    # sous cette forme, tout devient plus simple si on devait
    # avoir d'autres colonnes à prendre en compte
    keys = ('p', 'n', 'p2')
    liste.sort(key=lambda d: tuple(d[k] for k in keys if k in d))
    return liste
```

doubler_premier - Semaine 5 Séquence 2

```
def doubler_premier(func, first, *args):
    """
    renvoie le résultat de la fonction f appliquée sur
    func(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 func, en doublant first
    return func(2*first, *args)
```

doubler_premier_bis - Semaine 5 Séquence 2

```
def doubler_premier_bis(func, *args):
    """
    marche aussi mais moins élégant
    """
    first, *remains = args
    return func(2*first, *remains)
```

doubler_premier_ter - Semaine 5 Séquence 2

```
def doubler_premier_ter(func, *args):
    """
    ou encore comme ça, mais
    c'est carrément moche
    """
    first = args[0]
    remains = args[1:]
    return func(2*first, *remains)
```

doubler_premier_kwds - Semaine 5 Séquence 2

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

# Complément - niveau avancé
# ----
# Il y a un cas qui ne fonctionne pas avec cette implémentation,
# quand le premier argument de func a une valeur par défaut
# *et* on veut pouvoir appeler doubler_premier
# en nommant ce premier argument
#
# par exemple - avec func=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(fun1, fun2, entrees):
    """
    retourne une liste de booléens, un par entree dans entrees
    qui indique si fun1(entree) == fun2(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 [fun1(entree) == fun2(entree) for entree in entrees]
```

compare_args - Semaine 5 Séquence 2

```
def compare_args(fun1, fun2, arg_tuples):
    """
    retourne une liste de booléens, un par entree dans entrees
    qui indique si fun1(*tuple) == fun2(*tuple)
    """
    # c'est presque exactement comme compare_all, 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 [fun1(*arg) == fun2(*arg) for arg in arg_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(iter1, iter2):
    """
    renvoie une liste des éléments
    pris alternativement dans iter1 et dans iter2
    """
    # pour réaliser l'alternance on peut combiner zip avec aplatir
    # telle qu'on vient de la réaliser
    return aplatir(zip(iter1, iter2))
```

alternat_bis - Semaine 5 Séquence 3

```
def alternat_bis(iter1, iter2):
    """
    une deuxième version de alternat
    """
    # la même idée mais directement, sans utiliser aplatir
    return [element for conteneur in zip(iter1, iter2)
            for element in conteneur]
```

intersect - Semaine 5 Séquence 3

```
def intersect(tuples_a, tuples_b):
    """
    prend en entrée deux listes de tuples de la forme
    (entier, valeur)

    renvoie l'ensemble des valeurs associées, dans A ou B,
    aux entiers présents dans A et B

    il y a **plein** d'autres façons de faire, mais il faut
    juste se méfier de ne pas tout recalculer plusieurs fois
    si on veut faire trop court

    """

    # pour montrer un exemple de fonction locale:
    # une fonction qui renvoie l'ensemble des entiers
    # présents comme clé dans une liste d'entrée
    def keys(tuples):
        return {entier for entier, valeur in tuples}
    # on l'applique à A et B
    keys_a = keys(tuples_a)
    keys_b = keys(tuples_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 {val_a for key, val_a in tuples_a if key in common_keys} \
         | {val_b for key, val_b in tuples_b if key in common_keys}
```

intersect_bis - Semaine 5 Séquence 3

```
def intersect_bis(A, B):
    A, B = dict(A), dict(B)
    keys = set(A) & set(B)
    return {A[k] for k in keys} | {B[k] for k in keys}
```

cesar - Semaine 5 Séquence 3

```
1
2   # pour passer des majuscules aux minuscules, il faut ajouter
3   # 97-65=32
4   import string
5
6   UPPER_TO_LOWER = ord('a') - ord('A')
7
8
9   def cesar(clear, key, encode=True):
10      """
11      retourne l'encryption du caractere <clear> par la clé <key>
12
13      le caractère <key> doit être un caractère alphabétique ASCII
14      c'est à dire que son ord() est entre ceux de 'a' et 'z' ou
15      entre ceux de 'A' et 'Z'
16      """
17
18      if clear not in string.ascii_letters:
19          return clear
20
21      # le codepoint de la clé
22      okey = ord(key)
23      # on normalise la clé pour être dans les minuscules
24      if key.isupper():
25          okey += UPPER_TO_LOWER
26
27      # la variable offset est un entier entre 1 et 26 qui indique
28      # de combien on doit décaler; dans le tout premier
29      # exemple, avec une clé qui vaut 'C' offset va valoir 3
30      offset = (okey - ord('a') + 1)
31
32      # si on encode, il faut ajouter l'offset,
33      # et si on décode, il faut le retrancher
34      if not encode:
35          offset = -offset
36
37      # ne reste plus qu'à faire le modulo
38      # sauf que les bornes ne sont pas les mêmes
39      # pour les majuscules ou pour les minuscules
40      bottom = ord('A') if clear.isupper() else ord('a')
41
42      return chr(bottom + (ord(clear) - bottom + offset) % 26)
```

cesar_bis - Semaine 5 Séquence 3

```
from itertools import chain

# une autre approche entièrement consiste à précalculer
# toutes les valeurs et les ranger dans un dictionnaire
# qui va être haché par le tuple
# (clear, key)
# ça ne demande que 4 * 26 * 26 entrées dans le dictionnaire
# c'est à dire environ 2500 entrées, ce n'est pas grand chose

# on commence par le cas où le texte et la clé sont minuscules
# on rappelle que ord('a')=97
# avec nos définitions, une clé implique un décalage
# de (ord(k)-96), car une clé A signifie un décalage de 1
# par contre pour faire les calculs modulo 26
# il faut faire (ord(c)-97) de façon à ce que A=0 et Z=25
ENCODED_LOWER_LOWER = {
    (c, k): chr((ord(c) - 97 + ord(k) - 96) % 26 + 97)
    for c in string.ascii_lowercase
    for k in string.ascii_lowercase
}

# maintenant on peut facilement en déduire la table
# pour un texte en minuscule et une clé en majuscule
# il suffit d'appliquer ENCODED_LOWER_LOWER avec la clé minuscule
ENCODED_LOWER_UPPER = {
    (c, k): ENCODED_LOWER_LOWER[(c, k.lower())]
    for c in string.ascii_lowercase
    for k in string.ascii_uppercase
}
```

cesar_bis (continued) - Semaine 5 Séquence 3

```
1
2   # enfin pour le cas où le texte est en majuscule, on
3   # va considérer l'union des deux premières tables
4   # (que l'on va balayer avec itertools.chain sur leurs items())
5   # et dire que pour encoder un caractère majuscule, on
6   # n'a qu'à prendre encoder la minuscule et mettre le résultat en majuscule
7   ENCODED_UPPER = {
8       (c.upper(), k): value.upper()
9       for (c, k), value in chain(ENCODED_LOWER_LOWER.items(),
10                                 ENCODED_LOWER_UPPER.items())
11  }
12
13  # maintenant on n'a plus qu'à construire
14  # l'union de ces 3 dictionnaires
15  ENCODE_LOOKUP = {}
16  ENCODE_LOOKUP.update(ENCODED_LOWER_LOWER)
17  ENCODE_LOOKUP.update(ENCODED_LOWER_UPPER)
18  ENCODE_LOOKUP.update(ENCODED_UPPER)
19
20  # et alors pour calculer la table inverse,
21  # c'est extrêmement simple, on dit que
22  # decode(encoded, key) == clear
23  # ssi
24  # encode(clear, key) == encoded
25  DECODE_LOOKUP = {
26      (encoded, key): clear for (clear, key), encoded
27      in ENCODE_LOOKUP.items()
28  }
29
30  # et maintenant pour faire le travail il suffit de
31  # faire exactement **UNE** recherche dans la table qui va bien
32  # ce qui est plus efficace en principe que la première approche
33  # si le couple (texte, clé) n'est pas trouvé alors on renvoie texte tel quel
34  def cesar_bis(clear, key, encode=True):
35      lookup = ENCODE_LOOKUP if encode else DECODE_LOOKUP
36      return lookup.get((clear, key), clear)
```

vigenere - Semaine 5 Séquence 3

```
from itertools import cycle

# grâce à une combinaison de zip et de itertools.cycle
# on peut itérer sur
# d'une part, le message
# et d'autre part, sur la clé, en boucle
#
# notez que
# (*) cycle ne s'arrête jamais
# (*) mais zip, lui, s'arrête au plus court de ses (ici deux)
#     ingrédients
# ce qui fait que zip(message, cycle(cle))
# fait exactement ce dont on a besoin

def vigenere(clear, key, encode=True):
    return "".join(
        cesar(c, k, encode)
        for c, k in zip(clear, cycle(key))
    )
```

produit_scalaire - Semaine 5 Séquence 4

```
def produit_scalaire(vec1, vec2):
    """
    retourne le produit scalaire
    de deux listes de même taille
    """
    # avec zip() on peut faire correspondre les
    # valeurs de vec1 avec celles de vec2 de même rang
    #
    # et on utilise la fonction builtin sum sur une itération
    # des produits x1*x2
    #
    # 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 x1*x2
    #
    return sum(x1 * x2 for x1, x2 in zip(vec1, vec2))
```

produit_scalaire_bis - Semaine 5 Séquence 4

```
# Il y a plein d'autres solutions qui marchent aussi
#
def produit_scalaire_bis(vec1, vec2):
    """
    Une autre version, où on fait la somme à la main
    """
    scalaire = 0
    for x1, x2 in zip(vec1, vec2):
        scalaire += x1 * x2
    # on retourne le résultat
    return scalaire
```

produit_scalaire_ter - Semaine 5 Séquence 4

```
# Et encore une:
# celle-ci par contre est assez peu "pythonique"
#
# considérez-la comme un exemple de
# ce qu'il faut ÉVITER DE FAIRE:
#
def produit_scalaire_ter(vec1, vec2):
    """
    Lorsque vous vous trouvez en train d'écrire:

        for i in range(len(sequence)):
            x = iterable[sequence]
            # etc...

    vous pouvez toujours écrire à la place:

        for x in sequence:
            ...

    qui en plus d'être plus facile à lire,
    marchera sur tout itérable, et sera plus rapide
    """
    scalaire = 0
    # sachez reconnaitre ce vilain idiome:
    for i in range(len(vec1)):
        scalaire += vec1[i] * vec2[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
# énumérer les 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; mais ce n'est pas le cas, il faut
# laisser intacts les caractères de this.s qui ne sont pas
# dans this.d

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 dictionnaire qui implémente 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 est présente, et default sinon

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

decode_zen_ter - Semaine 5 Séquence 7

```
# une dernière version utilisant les fonctions ad hoc
# https://docs.python.org/3/library/stdtypes.html#str.translate
# et https://docs.python.org/3/library/stdtypes.html#str.maketrans

def decode_zen_ter(this_module):
    """
    cette version utilise les fonctions ad hoc de la classe str
    """
    # Le dictionnaire this_module.d n'est pas utilisable directement,
    # il faut faire la transformation fournie par str.maketrans
    # car la fonction translate attend comme clés des nombres
    # représentant la valeur Unicode des caractères.
    # Or this_module.d a comme clés les caractères à décoder
    # et non leur valeur Unicode.
    return this_module.s.translate(str.maketrans(this_module.d))
```