MOOC Python

Corrigés de la semaine 4

comptage - Semaine 4 Séquence 1

```
def comptage(in_filename, out_filename):
    """
    retranscrit le fichier in_filename dans le fichier out_filename
    en ajoutant des annotations sur les nombres de lignes, de mots
    et de caractères
    """
    # on ouvre le fichier d'entrée en lecture
    # on aurait pu mettre open (in_filename, 'r')
    with open(in_filename) as input:
        # on ouvre la sortie en écriture
        with open(out_filename, "w") as output:
            # initialisations
            lineno = 0
            total_words = 0
            total_chars = 0
            # pour toutes les lignes du fichier d'entrée
            for line in input:
                # on maintient le nombre de lignes
                # qui est aussi la ligne courante
                lineno += 1
                # autant de mots que d'éléments dans split()
                nb_words = len(line.split())
                total_words += nb_words
                # autant de caractères que d'éléments dans la ligne
                nb_chars = len(line)
                total_chars += nb_chars
                # on écrit la ligne de sortie; pas besoin
                # de newline (\n) car line en a déjà un
                output.write("{}:{}:{}:{}"
                             .format(lineno, nb_words, nb_chars, line))
            # on écrit la ligne de synthèse
            output.write("{}:{}:{}\n".\
                         format(lineno, total_words, total_chars))
```

pgcd - Semaine 4 Séquence 2

```
def pgcd(a, b):
    "le pgcd de a et b par l'algorithme d'Euclide"
    # l'algorithme suppose que a >= b
    # donc si ce n'est pas le cas
    # il faut inverser les deux entrées
    if b > a :
        a, b = b, a
    # boucle sans fin
    while True:
        # on calcule le reste
        r = a % b
        # si le reste est nul, on a terminé
        if r == 0:
            return b
        # sinon on passe à l'itération suivante
        a, b = b, r
```

pgcd (v2) - Semaine 4 Séquence 2

```
# il se trouve qu'en fait la première inversion n'est
# pas nécessaire
# en effet si a <= b, la première itération de la boucle
# while va faire
# r = a % b = a
# et ensuite
# a, b = b, r = b, a
# ce qui provoque l'inversion
def pgcd_bis(a, b):
    while True:
        # on calcule le reste
        r = a % b
        # si le reste est nul, on a terminé
        if r == 0:
            return b
        # sinon on passe à l'itération suivante
        a, b = b, r
```

numbers - Semaine 4 Séquence 3

```
from operator import mul

def numbers(liste):
    """
    retourne un tuple contenant
    (*) la somme
    (*) le produit
    (*) le minimum
    (*) le maximum
    des éléments de la liste
    """

    return (
        # la builtin 'sum' renvoie la somme
        sum(liste),
        # pour la multiplication, reduce est nécessaire
        reduce(mul, liste, 1),
        # les builtin 'min' et 'max' font ce qu'on veut aussi
        min(liste),
        max(liste),
    )
```

compare - Semaine 4 Séquence 3

```
def compare(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]
```

aplatir - Semaine 4 Séquence 4

```
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 4 Séquence 4

```
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 (v2) - Semaine 4 Séquence 4

```
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 4 Séquence 4

```
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 values(S):
        return {i for i, val in S}
    # on l'applique à A et B
    val_A = values(A)
    val_B = values(B)
    #
    # bien sûr on aurait pu écrire directement
    # val_A = {i for i, val in A}
    # val_B = {i for i, val in B}
    #
    # les entiers présents dans A et B
    # avec une intersection d'ensembles
    common_keys = val_A & val_B
    # et pour conclure on fait une union sur deux
    # compréhensions d'ensembles
    return {vala for a, vala in A if a in common_keys} \
         | {valb for b, valb in B if b in common_keys}
```

distance - Semaine 4 Séquence 8

```
import math

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

doubler_premier - Semaine 4 Séquence 8

```
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 (v2) - Semaine 4 Séquence 8

```
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 4 Séquence 8

```
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_args - Semaine 4 Séquence 8

```
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]
```