MOOC Python

Tous les corrigés

Table des matières

pythonid (regexp) - Semaine 2 Séquence 2

```
# un identificateur commence par une lettre ou un underscore
# et peut être suivi par n'importe quel nombre de
# lettre, chiffre ou underscore, ce qui se trouve être \w
# si on ne se met pas en mode unicode
pythonid = "[a-zA-Z_]\w*"
```

pythonid (bis) - Semaine 2 Séquence 2

```
# on peut aussi bien sûr l'écrire en clair
pythonid_bis = "[a-zA-Z_][a-zA-Z0-9_]*"
```

agenda (regexp) - Semaine 2 Séquence 2

```
# l'exercice est basé sur re.match, ce qui signifie que
# le match est cherché au début de la chaine
# MAIS il nous faut bien mettre \Z à la fin de notre regexp,
# sinon par exemple avec la cinquième entrée le nom 'Du Pré'
# sera reconnu partiellement comme simplement 'Du'
# au lieu d'être rejeté à cause de l'espace
#
# du coup pensez à bien toujours définir
# vos regexps avec des raw-strings
#
# remarquez sinon l'utilisation à la fin de :? pour signifier qu'on peut
# mettre ou non un deuxième séparateur ':'
#
agenda = r"\A(?P<prenom>[-\w]*):(?P<nom>[-\w]+):?\Z"
```

phone (regexp) - Semaine 2 Séquence 2

```
# idem concernant le \Z final
#
# il faut bien backslasher le + dans le +33
# car sinon cela veut dire 'un ou plusieurs'
#
phone = r"(\+33|0)(?P<number>[0-9]{9})\Z"
```

url (regexp) - Semaine 2 Séquence 2

```
# en ignorant la casse on pourra ne mentionner les noms de protocoles
# qu'en minuscules
i_flag = "(?i)"

# pour élaborer la chaine (proto1|proto2|...)
protos_list = ['http', 'https', 'ftp', 'ssh', ]
protos      = "(?P<proto>" + "|".join(protos_list) + ")"

# à l'intérieur de la zone 'user/password', la partie
# password est optionnelle - mais on ne veut pas le ':' dans
# le groupe 'password' - il nous faut deux groupes
password    = r"(:(?P<password>[^:]+))?"

# la partie user-password elle-même est optionnelle
user        = r"((?P<user>\w+){password}@)?".format(**locals())

# pour le hostname on accepte des lettres, chiffres, underscore et '.'
# attention à backslaher . car sinon ceci va matcher tout y compris /
hostname    = r"(?P<hostname>[\w\.]+)"

# le port est optionnel
port        = r"(:(?P<port>\d+))?"

# après le premier slash
path        = r"(?P<path>.*)"

# on assemble le tout
url = i_flag + protos + "://" + user + hostname + port + '/' + path
```

label - Semaine 2 Séquence 6

```
def label(prenom, note):
    if note < 10:
        return f"{prenom} est recalé"
    elif note < 16:
        return f"{prenom} est reçu"
    else:
        return f"félicitations à {prenom}"
```

inconnue - Semaine 2 Séquence 6

```
# pour enlever à gauche et à droite une chaine de longueur x
# on peut faire composite[ x : -x ]
# or ici x vaut len(connue)
def inconnue(composite, connue):
    return composite[ len(connue) : -len(connue) ]
```

inconnue (bis) - Semaine 2 Séquence 6

```
# ce qui peut aussi s'écrire comme ceci si on préfère
def inconnue_bis(composite, connue):
    return composite[ len(connue) : len(composite)-len(connue) ]
```

laccess - Semaine 2 Séquence 6

```
def laccess(liste):
    """
    retourne un élément de la liste selon la taille
    """
    # si la liste est vide il n'y a rien à faire
    if not liste:
        return
    # si la liste est de taille paire
    if len(liste) % 2 == 0:
        return liste[-1]
    else:
        return liste[len(liste)//2]
```

laccess (bis) - Semaine 2 Séquence 6

```
# une autre version qui utilise
# un trait qu'on n'a pas encore vu
def laccess(liste):
    # si la liste est vide il n'y a rien à faire
    if not liste:
        return
    # l'index à utiliser selon la taille
    index = -1 if len(liste) % 2 == 0 else len(liste) // 2
    return liste[index]
```

divisible - Semaine 2 Séquence 6

```
def divisible(a, b):
    "renvoie True si un des deux arguments divise l'autre"
    # b divise a si et seulement si le reste
    # de la division de a par b est nul
    if a % b == 0:
        return True
    # et il faut regarder aussi si a divise b
    if b % a == 0:
        return True
    return False
```

divisible (bis) - Semaine 2 Séquence 6

```
def divisible_bis(a, b):
    "renvoie True si un des deux arguments divise l'autre"
    # on n'a pas encore vu les opérateurs logiques, mais
    # on peut aussi faire tout simplement comme ça
    # sans faire de if du tout
    return a % b == 0 or b % a == 0
```

morceaux - Semaine 2 Séquence 6

```
def morceaux(x):
    if x <= -5:
        return -x - 5
    elif x <= 5:
        return 0
    else:
        return x / 5 - 1
```

morceaux (bis) - Semaine 2 Séquence 6

```
def morceaux_bis(x):
    if x <= -5:
        return -x - 5
    if x <= 5:
        return 0
    return x / 5 - 1
```

morceaux (ter) - Semaine 2 Séquence 6

```
# on peut aussi faire des tests d'intervalle
# comme ceci  0 <= x <= 10
def morceaux_ter(x):
    if x <= -5:
        return -x - 5
    elif -5 <= x <= 5:
        return 0
    else:
        return x / 5 - 1
```

liste_P - Semaine 2 Séquence 7

```
def P(x):
    return 2 * x**2 - 3 * x - 2

def liste_P(liste_x):
    """
    retourne la liste des valeurs de P
    sur les entrées figurant dans liste_x
    """
    return [P(x) for x in liste_x]
```

liste_P (bis) - Semaine 2 Séquence 7

```
# On peut bien entendu faire aussi de manière pédestre
def liste_P_bis(liste_x):
    liste_y = []
    for x in liste_x:
        liste_y.append(P(x))
    return liste_y
```

carre - Semaine 2 Séquence 7

```
def carre(s):
    # on enlève les espaces et les tabulations
    s = s.replace(' ', '').replace('\t','')
    # la ligne suivante fait le plus gros du travail
    # d'abord on appelle split() pour découper selon les ';'
    # dans le cas où on a des ';' en trop, on obtient dans le
    #    résultat du split un 'token' vide, que l'on ignore
    #    ici avec le clause 'if token'
    # enfin on convertit tous les tokens restants en entiers avec int()
    entiers = [int(token) for token in s.split(";")
               # en éliminant les entrées vides qui correspondent
               # à des point-virgules en trop
               if token]
    # il n'y a plus qu'à mettre au carré, retraduire en strings,
    # et à recoudre le tout avec join et ':'
    return ":".join([str(entier**2) for entier in entiers])
```

comptage - Semaine 3 Séquence 2

```
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, encoding='utf-8') as input:
        # on ouvre la sortie en écriture
        with open(out_filename, "w", encoding='utf-8') as output:
            lineno = 1
            # pour toutes les lignes du fichier d'entrée
            # le numéro de ligne commence à 1
            for line in input:
                # autant de mots que d'éléments dans split()
                nb_words = len(line.split())
                # autant de caractères que d'éléments dans la ligne
                nb_chars = len(line)
                # on écrit la ligne de sortie; pas besoin
                # de newline (\n) car line en a déjà un
                output.write(f"{lineno}:{nb_words}:{nb_chars}:{line}")
                lineno += 1
```

comptage (bis) - Semaine 3 Séquence 2

```
# un peu plus pythonique avec enumerate
def comptage_bis(in_filename, out_filename):
    with open(in_filename, encoding='utf-8') as input:
        with open(out_filename, "w", encoding='utf-8') as output:
            # enumerate(.., 1) pour commencer avec une ligne
            # numérotée 1 et pas 0
            for lineno, line in enumerate(input, 1):
                output.write(f"{lineno}:{len(line.split())}:"
                             f"{len(line)}:{line}")
```

surgery - Semaine 3 Séquence 2

```
def surgery(liste):
    """
    Prend en argument une liste, et retourne la liste modifiée:
    * taille paire: on intervertit les deux premiers éléments
    * taille impaire >= 3: on fait tourner les 3 premiers éléments
    """
    # si la liste est de taille 0 ou 1, il n'y a rien à faire
    if len(liste) < 2:
        pass
    # si la liste est de taille paire
    elif len(liste) % 2 == 0:
        # on intervertit les deux premiers éléments
        liste[0], liste[1] = liste[1], liste[0]
    # si elle est de taille impaire
    else:
        liste[-2], liste[-1] = liste[-1], liste[-2]
    # et on n'oublie pas de retourner la liste dans tous les cas
    return liste
```

graph_dict - Semaine 3 Séquence 4

```
# on va utiliser un defaultdict

from collections import defaultdict

def graph_dict(filename):
    # on le déclare de type list
    g = defaultdict(list)
    with open(filename) as f:
        for line in f:
            # on coupe la ligne en trois parties
            begin, value, end = line.split()
            # comme c'est un defaultdict on n'a
            # pas besoin de l'initialiser
            g[begin].append( (end, int(value)))
    return g
```

graph_dict (bis) - Semaine 3 Séquence 4

```
# la même chose mais sans defaultdict

def graph_dict_bis(filename):
    # un dictionnaire vide
    g = {}
    with open(filename) as f:
        for line in f:
            begin, value, end = line.split()
            # c'est ça qu'on économise avec un defaultdict
            if begin not in g:
                g[begin] = []
            # sinon c'est tout pareil
            g[begin].append( (end, int(value)))
    return g
```

index - Semaine 3 Séquence 4

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

index (bis) - Semaine 3 Séquence 4

```
def index_bis(bateaux):
    """
    La même chose mais de manière itérative
    """
    # si on veut décortiquer
    resultat = {}
    for bateau in bateaux:
        resultat [bateau[0]] = bateau
    return resultat
```

index (ter) - Semaine 3 Séquence 4

```
def index_ter(bateaux):
    """
    Encore une autre, avec un extended unpacking
    """
    # si on veut décortiquer
    resultat = {}
    for bateau in bateaux:
        # avec un extended unpacking on peut extraire
        # le premier champ; en appelant le reste _
        # on indique qu'on n'en fera en fait rien
        id, *_ = bateau
        resultat [id] = bateau
    return resultat
```

merge - Semaine 3 Séquence 4

```
def merge(extended, abbreviated):
    """
    Consolide des données étendues et des données abrégées
    comme décrit dans l'énoncé
    Le coût de cette fonction est linéaire dans la taille
    des données (longueur commune des deux listes)
    """
    # on initialise le résultat avec un dictionnaire vide
    result = {}
    # pour les données étendues
    # on affecte les 6 premiers champs
    # et on ignore les champs de rang 6 et au delà
    for id, latitude, longitude, timestamp, name, country, *ignore in extended:
        # on crée une entrée dans le résultat,
        # avec la mesure correspondant aux données étendues
        result[id] = [name, country, (latitude, longitude, timestamp)]
    # maintenant on peut compléter le résultat avec les données abrégées
    for id, latitude, longitude, timestamp in abbreviated:
        # et avec les hypothèses on sait que le bateau a déjà été
        # inscrit dans le résultat, donc result[id] doit déjà exister
        # et on peut se contenter d'ajouter ls mesure abrégée
        # dans l'entrée correspondant dans result
        result[id].append((latitude, longitude, timestamp))
    # et retourner le résultat
    return result
```

merge (bis) - Semaine 3 Séquence 4

```
def merge_bis(extended, abbreviated):
    """
    Une deuxième version, linéaire également
    """
    # on initialise le résultat avec un dictionnaire vide
    result = {}
    # on remplit d'abord à partir des données étendues
    for ship in extended:
        id = ship[0]
        # on crée la liste avec le nom et le pays
        result[id] = ship[4:6]
        # on ajoute un tuple correspondant à la position
        result[id].append(tuple(ship[1:4]))
    # pareil que pour la première solution,
    # on sait d'après les hypothèses
    # que les id trouvées dans abbreviated
    # sont déja présentes dans le resultat
    for ship in abbreviated:
        id = ship[0]
        # on ajoute un tuple correspondant à la position
        result[id].append(tuple(ship[1:4]))
    return result
```

merge (ter) - Semaine 3 Séquence 4

```
def merge_ter(extended, abbreviated):
    """
    Une troisième solution
    à cause du tri que l'on fait au départ, cette
    solution n'est plus linéaire mais en O(n.log(n))
    """
    # ici on va tirer profit du fait que les id sont
    # en première position dans les deux tableaux
    # si bien que si on les trie,
    # on va mettre les deux tableaux 'en phase'
    #
    # c'est une technique qui marche dans ce cas précis
    # parce qu'on sait que les deux tableaux contiennent des données
    # pour exactement le même ensemble de bateaux
    #
    # on a deux choix, selon qu'on peut se permettre ou non de
    # modifier les données en entrée. Supposons que oui:
    extended.sort()
    abbreviated.sort()
    # si ça n'avait pas été le cas on aurait fait plutôt
    # extended = extended.sorted() et idem pour l'autre
    #
    # il ne reste plus qu'à assembler le résultat
    # en découpant des tranches
    # et en les transformant en tuples pour les positions
    # puisque c'est ce qui est demandé
    return {
        e[0] : e[4:6] + [ tuple(e[1:4]), tuple(a[1:4]) ]
        for (e,a) in zip (extended, abbreviated)
        }
```

read_set - Semaine 3 Séquence 5

```
# on suppose que le fichier existe
def read_set(filename):
    # on crée un ensemble vide
    result = set()
    # on parcourt le fichier
    with open(filename) as f:
        for line in f:
            # avec strip() on enlève la fin de ligne,
            # et les espaces au début et à la fin
            result.add(line.strip())
    return result
```

read_set (bis) - Semaine 3 Séquence 5

```
# on peut aussi utiliser une compréhension d'ensemble
# (voir semaine 5)
# comme une compréhension de liste mais on remplace
# les [] par des {}
def read_set_bis(filename):
    with open(filename) as f:
        return {line.strip() for line in f}
```

search_in_set (bis) - Semaine 3 Séquence 5

```
def search_in_set_bis(filename_reference, filename):

    # on tire profit de la fonction précédente
    reference_set = read_set(filename_reference)

    # c'est plus clair avec une compréhension
    with open(filename) as f:
        return [ (line.strip(), line.strip() in reference_set)
                 for line in f ]
```

diff - Semaine 3 Séquence 5

```
def diff(extended, abbreviated):
    """Calcule comme demandé dans l'exercice, et sous formes d'ensembles
    (*) les noms des bateaux seulement dans extended
    (*) les noms des bateaux présents dans les deux listes
    (*) les ids des bateaux seulement dans abbreviated
    """
    ### on n'utilise que des ensembles dans tous l'exercice
    # les ids de tous les bateaux dans extended
    # une compréhension d'ensemble
    extended_ids = {ship[0] for ship in extended}
    # les ids de tous les bateaux dans abbreviated
    # idem
    abbreviated_ids = {ship[0] for ship in abbreviated}
    # les ids des bateaux seulement dans abbreviated
    # une difference d'ensembles
    abbreviated_only_ids = abbreviated_ids - extended_ids
    # les ids des bateaux dans les deux listes
    # une intersection d'ensembles
    both_ids = abbreviated_ids & extended_ids
    # les ids des bateaux seulement dans extended
    # ditto
    extended_only_ids = extended_ids - abbreviated_ids
    # pour les deux catégories où c'est possible
    # on recalcule les noms des bateaux
    # par une compréhension d'ensemble
    both_names = \
          {ship[4] for ship in extended if ship[0] in both_ids}
    extended_only_names = \
          {ship[4] for ship in extended if ship[0] in extended_only_ids}
    # enfin on retourne les 3 ensembles sous forme d'un tuple
    return extended_only_names, both_names, abbreviated_only_ids
```

diff (bis) - Semaine 3 Séquence 5

```
def diff_bis(extended, abbreviated):
    """
    Idem avec seulement des compréhensions
    """
    extended_ids =     {ship[0] for ship in extended}
    abbreviated_ids =  {ship[0] for ship in abbreviated}
    abbreviated_only = {ship[0] for ship in abbreviated
                        if ship[0] not in extended_ids}
    extended_only =    {ship[4] for ship in extended
                        if ship[0] not in abbreviated_ids}
    both =             {ship[4] for ship in extended
                        if ship[0] in abbreviated_ids}
    return extended_only, both, abbreviated_only
```

fifo - Semaine 3 Séquence 8

```
class Fifo:
    """
    Une classe FIFO implémentée avec une simple liste
    """

    def __init__(self):
        # l'attribut queue est un objet liste
        self.queue = []

    def incoming(self, x):
        # on insère au début de la liste
        self.queue.insert(0, x)

    def outgoing(self):
        # une première façon de faire consiste à
        # utiliser un try/except
        try:
            return self.queue.pop()
        except IndexError:
            return None
```

fifo (bis) - Semaine 3 Séquence 8

```
# une autre implémentation pourrait faire comme ceci
class FifoBis(Fifo):
    def __init__(self):
        self.queue = []

    def incoming(self, x):
        self.queue.insert(0, x)

    def outgoing(self):
        # plus concis mais peut-être moins lisible
        if len(self.queue):
            return self.queue.pop()
        # en fait on n'a même pas besoin du else..

```

dispatch1 - Semaine 4 Séquence 2

```
def dispatch1(a, b):
    """dispatch1 comme spécifié"""
    # si les deux arguments sont pairs
    if a%2 == 0 and b%2 == 0:
        return a*a + b*b
    # si a est pair et b est impair
    elif a%2 == 0 and b%2 != 0:
        return a*(b-1)
    # si a est impair et b est pair
    elif a%2 != 0 and b%2 == 0:
        return (a-1)*b
    # sinon - c'est que a et b sont impairs
    else:
        return a*a - b*b
```

dispatch2 - Semaine 4 Séquence 2

```
def dispatch2(a, b, A, B):
    """dispatch2 comme spécifié"""
    # les deux cas de la diagonale \
    if (a in A and b in B) or (a not in A and b not in B):
        return a*a + b*b
    # sinon si b n'est pas dans B
    # ce qui alors implique que a est dans A
    elif b not in B:
        return a*(b-1)
    # le dernier cas, on sait forcément que
    # b est dans B et a n'est pas dans A
    else:
        return (a-1)*b
```

libelle - Semaine 4 Séquence 2

```
def libelle(ligne):
    # on enlève les espaces et les tabulations
    ligne = ligne.replace(' ', '').replace('\t','')
    # on cherche les 3 champs
    mots = ligne.split(',')
    # si on n'a pas le bon nombre de champs
    # rappelez-vous que 'return' tout court
    # est équivalent à 'return None'
    if len(mots) != 3:
        return
    # maintenant on a les trois valeurs
    nom, prenom, rang = mots
    # comment présenter le rang
    rang_ieme = "1er" if rang == "1" \
                else "2nd" if rang == "2" \
                else f"{rang}-ème"
    return f"{prenom}.{nom} ({rang_ieme})"
```

pgcd - Semaine 4 Séquence 3

```
def pgcd(a, b):
    # le cas pathologique
    if a * b == 0:
        return 0
    "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 (bis) - Semaine 4 Séquence 3

```
# 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):
    # le cas pathologique
    if a == 0 or b == 0:
        return 0
    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 (ter) - Semaine 4 Séquence 3

```
# une autre alternative, qui fonctionne aussi
# plus court, mais on passe du temps à se convaincre
# que ça fonctionne bien comme demandé
def pgcd_ter(a, b):
    # le cas pathologique
    if a * b == 0:
        return 0
    # si on n'aime pas les boucles sans fin
    # on peut faire aussi comme ceci
    while b:
        a, b = b, a % b
    return a
```

distance - Semaine 4 Séquence 6

```
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]))
```

numbers - Semaine 4 Séquence 6

```
from operator import mul

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

    if not liste:
        return 0, 0, 0

    return (
        # la builtin 'sum' renvoie la somme
        sum(liste),
        # les builtin 'min' et 'max' font ce qu'on veut aussi
        min(liste),
        max(liste),
    )
```

numbers (bis) - Semaine 4 Séquence 6

```
# en regardant bien la documentation de sum, max et min,
# on voit qu'on peut aussi traiter le cas singulier
# (pas d'argument) en passant
#   start à sum
#   et default à min ou max
# comme ceci
def numbers_bis(*liste):
    return (
        # attention:
        # la signature de sum est: sum(iterable[, start])
        # du coup on ne peut pas passer à sum start=0
        # parce que start n'a pas de valeur par défaut
        sum(liste, 0),
        # par contre avec min c'est min(iterable, *[, key, default])
        # du coup on doit appeler min avec default=0 qui est plus clair
        # l'étoile qui apparaît dans la signature
        # rend le paramètre default keyword-only
        min(liste, default=0),
        max(liste, default=0),
    )
```

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)
```

RPCProxy - Semaine 6 Séquence 2

```
# une troisième implémentation de RPCProxy

class Forwarder:
    """
    Une instance de la classe Forwarder est un callable
    qui peut être utilisée comme une méthode sur l
    class RPCProxy
    """
    def __init__(self, rpc_proxy, methodname):
        """
        le constructeur  mémorise l'instance de RPCProxy
        et le nom de la méthode qui a été appelée
        """
        self.methodname = methodname
        self.rpc_proxy = rpc_proxy

    def __call__(self, *args):
        """
        en rendant cet objet callable, on peut l'utiliser
        comme une méthode de RPCProxy
        """
        print "Envoi à {}\nde la fonction {} -- args= {}".\
            format(self.rpc_proxy.url, self.methodname, args)
        return "retour de la fonction " + self.methodname

class RPCProxy:
    """
    Une troisième implémentation de RPCProxy qui sous-traite
    à une classe annexe 'Forwarder' qui se comporte comme
    une *factory* de méthodes
    """
    def __init__(self, url, login, password):
        self.url = url
        self.login = login
        self.password = password

    def __getattr__ (self, methodname):
        """
        Crée à la volée une instance de Forwarder
        correspondant à 'methodname'
        """
        return Forwarder(self, methodname)
```