MOOC Python 3

Session 2018

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
# on utilise ici un raw f-string avec le préfixe rf
# pour insérer la regexp <password> dans la regexp <user>
user        = rf"((?P<user>\w+){password}@)?"

# 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
```

url (bis) - Semaine 2 Séquence 2

```
# merci à sizeof qui a pointé l'utilisation de re.X
# https://docs.python.org/fr/3/library/re.html#re.X
# ce qui donne une présentation beaucoup plus compacte

protos_list = ['http', 'https', 'ftp', 'ssh', ]

url_bis = rf"""(?x)                     # verbose mode
    (?i)                                # ignore case
    (?P<proto>{"|".join(protos_list)}) # http|https|...
    ://                                 # separator
    ((?P<user>\w+){password}@)?         # optional user/password
    (?P<hostname>[\w\.]+)               # mandatory hostname
    (:(?P<port>\d+))?                   # optional port
    /(?P<path>.*)                       # mandatory 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}"
```

label (bis) - Semaine 2 Séquence 6

```
def label_bis(prenom, note):
    if note < 10:
        return f"{prenom} est recalé"
    # on n'en a pas vraiment besoin ici, mais
    # juste pour illustrer cette construction
    elif 10 <= note < 16:
        return f"{prenom} est reçu"
    else:
        return f"félicitations à {prenom}"
```

label (ter) - Semaine 2 Séquence 6

```
# on n'a pas encore vu l'expression conditionnelle
# et dans ce cas précis ce n'est pas forcément une
# idée géniale, mais pour votre curiosité on peut aussi
# faire comme ceci
def label_ter(prenom, note):
    return f"{prenom} est recalé" if note < 10 \
    else f"{prenom} est reçu" if 10 <= note < 16 \
    else 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
```

wc - Semaine 2 Séquence 6

```
def wc(string):
    """
    Compte les nombres de lignes, de mots et de caractères

    Retourne une liste de ces 3 nombres (notez qu'usuellement
    on renverrait plutôt un tuple, qu'on étudiera la semaine prochaine)
    """
    # on peut tout faire avec la bibliothèque standard
    nb_lines = string.count('\n')
    nb_words = len(string.split())
    nb_bytes = len(string)
    return [nb_lines, nb_words, nb_bytes]
```

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(line):
    # on enlève les espaces et les tabulations
    line = line.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 la clause 'if token'
    # enfin on convertit tous les tokens restants en entiers avec int()
    entiers = [int(token) for token in line.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])
```

carre (bis) - Semaine 2 Séquence 7

```
def carre_bis(line):
    # pareil mais avec, à la place des compréhensions
    # des expressions génératrices que - rassurez-vous -
    # l'on n'a pas vues encore, on en parlera en semaine 5
    # le point que je veux illustrer ici c'est que c'est
    # exactement le même code mais avec () au lieu de []
    line = line.replace(' ', '').replace('\t','')
    entiers = (int(token) for token in line.split(";")
               if token)
    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
    with open(in_filename, encoding='utf-8') as in_file:
        # on ouvre la sortie en écriture
        with open(out_filename, 'w', encoding='utf-8') as out_file:
            lineno = 1
            # pour toutes les lignes du fichier d'entrée
            # le numéro de ligne commence à 1
            for line in in_file:
                # 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
                out_file.write(f"{lineno}:{nb_words}:{nb_chars}:{line}")
                lineno += 1
```

comptage (bis) - Semaine 3 Séquence 2

```
def comptage_bis(in_filename, out_filename):
    """
    un peu plus pythonique avec enumerate
    """
    with open(in_filename, encoding='utf-8') as in_file:
        with open(out_filename, 'w', encoding='utf-8') as out_file:
            # enumerate(.., 1) pour commencer avec une ligne
            # numérotée 1 et pas 0
            for lineno, line in enumerate(in_file, 1):
                # une astuce : si on met deux chaines
                # collées comme ceci elle sont concaténées
                # et on n'a pas besoin de mettre de backslash
                # puisqu'on est dans des parenthèses
                out_file.write(f"{lineno}:{len(line.split())}:"
                               f"{len(line)}:{line}")
```

comptage (ter) - Semaine 3 Séquence 2

```
def comptage_ter(in_filename, out_filename):
    """
    pareil mais avec un seul with
    """
    with open(in_filename, encoding='utf-8') as in_file, \
         open(out_filename, 'w', encoding='utf-8') as out_file:
        for lineno, line in enumerate(in_file, 1):
            out_file.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

```
from collections import defaultdict

def graph_dict(filename):
    """
    construit une stucture de données de graphe
    à partir du nom du fichier d'entrée
    """
    # un dictionnaire vide normal
    graph = {}

    with open(filename) as feed:
        for line in feed:
            begin, value, end = line.split()
            # c'est cette partie qu'on économisera
            # dans la deuxième solution avec un defaultdict
            if begin not in graph:
                graph[begin] = []
            # remarquez les doubles parenthèses
            # car on appelle append avec un seul argument
            # qui est un tuple
            graph[begin].append((end, int(value)))
            # si on n'avait écrit qu'un seul niveau de parenthèses
            # graph[begin].append(end, int(value))
            # cela aurait signifié un appel à append avec deux arguments
            # ce qui n'aurait pas du tout fait ce qu'on veut
    return graph
```

graph_dict (bis) - Semaine 3 Séquence 4

```
def graph_dict_bis(filename):
    """
    pareil mais en utilisant un defaultdict
    """
    # on déclare le defaultdict de type list
    # de cette façon si une clé manque elle
    # sera initialisée avec un appel à list()
    graph = defaultdict(list)

    with open(filename) as feed:
        for line in feed:
            # 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
            graph[begin].append((end, int(value)))
    return graph
```

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, *_ 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 la mesure abrégée
        # dans l'entrée correspondante 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
    mais qui utilise les indices plutôt que l'unpacking
    """
    # 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 résultat
    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 {
        ext[0] : ext[4:6] + [ tuple(ext[1:4]), tuple(abb[1:4]) ]
        for (ext, abb) in zip (extended, abbreviated)
        }
```

read_set - Semaine 3 Séquence 5

```
# on suppose que le fichier existe
def read_set(filename):
    """
    crée un ensemble des mots-lignes trouvés dans le fichier
    """
    # on crée un ensemble vide
    result = set()

    # on parcourt le fichier
    with open(filename) as feed:
        for line in feed:
            # 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); ça se présente comme
# une compréhension de liste mais on remplace
# les [] par des {}
def read_set_bis(filename):
    with open(filename) as feed:
        return {line.strip() for line in feed}
```

search_in_set - Semaine 3 Séquence 5

```
# ici aussi on suppose que les fichiers existent
def search_in_set(filename_reference, filename):
    """
    cherche les mots-lignes de filename parmi ceux
    qui sont presents dans filename_reference
    """

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

    # on crée une liste vide
    result = []
    with open(filename) as feed:
        for line in feed:
            token = line.strip()
            # remarquez ici les doubles parenthèses
            # pour passer le tuple en argument
            result.append((token, token in reference_set))

    return result
```

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 un plus clair avec une compréhension
    # mais moins efficace car on calcule strip() deux fois
    with open(filename) as feed:
        return [(line.strip(), line.strip() in reference_set)
                for line in feed]
```

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
    # avec ce qu'on a vu jusqu'ici le moyen le plus naturel
    # consiste à calculer une compréhension de liste
    # et à la traduire en ensemble comme ceci
    extended_ids = set([ship[0] for ship in extended])

    # les ids de tous les bateaux dans abbreviated
    # je fais exprès de ne pas mettre les []
    # de la compréhension de liste, c'est pour vous introduire
    # les expressions génératrices - voir semaine 5
    abbreviated_ids = set(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 = \
        set([ship[4] for ship in extended if ship[0] in both_ids])
    extended_only_names = \
        set([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):
    """
    Même code mais qui utilise les compréhensions d'ensemble
    que l'on n'a pas encore vues - à nouveau, voir semaine 5
    mais vous allez voir que c'est assez intuitif
    """
    extended_ids = {ship[0] for ship in extended}
    abbreviated_ids = {ship[0] for ship in abbreviated}

    abbreviated_only_ids = abbreviated_ids - extended_ids
    both_ids = abbreviated_ids & extended_ids
    extended_only_ids = extended_ids - abbreviated_ids

    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}

    return extended_only_names, both_names, abbreviated_only_ids
```

diff (ter) - Semaine 3 Séquence 5

```
def diff_ter(extended, abbreviated):
    """
    Idem sans les calculs d'ensembles intermédiaires
    en utilisant les conditions dans les 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
```

diff (quater) - Semaine 3 Séquence 5

```
def diff_quater(extended, abbreviated):
    """
    Idem sans indices
    """
    extended_ids =     {id for id, *_ in extended}
    abbreviated_ids =  {id for id, *_ in abbreviated}
    abbreviated_only = {id for id, *_ in abbreviated
                        if id not in extended_ids}
    extended_only =    {name for id, _, _, _, name, *_ in extended
                        if id not in abbreviated_ids}
    both =             {name for id, _, _, _, name, *_ in extended
                        if id 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, item):
        # on insère au début de la liste
        self.queue.insert(0, item)

    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

```
class FifoBis:
    """
    une alternative en testant directement
    plutôt que d'attraper l'exception
    """
    def __init__(self):
        self.queue = []

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

    def outgoing(self):
        # plus concis mais peut-être moins lisible
        if self.queue:
            return self.queue.pop()
        # pour que pylint soit content on *peut* retourner None explicitement
        return None

```

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):
    """
    n'oubliez pas votre docstring
    """
    # 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 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
    if b == 0:
        return a
    # boucle sans fin
    while True:
        # on calcule le reste
        reste = a % b
        # si le reste est nul, on a terminé
        if reste == 0:
            return b
        # sinon on passe à l'itération suivante
        a, b = b, reste
```

pgcd (bis) - Semaine 4 Séquence 3

```
def pgcd_bis(a, b):
    """
    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:
    reste = a % b  c'est-à-dire a
    et ensuite
    a, b = b, reste = b, a
    provoque l'inversion
    """
    # si l'on des deux est nul on retourne l'autre
    if a * b == 0:
        return a or b
    # sinon on fait une boucle sans fin
    while True:
        # on calcule le reste
        reste = a % b
        # si le reste est nul, on a terminé
        if reste == 0:
            return b
        # sinon on passe à l'itération suivante
        a, b = b, reste
```

pgcd (ter) - Semaine 4 Séquence 3

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

taxes - Semaine 4 Séquence 3

```
# une solution très élégante proposée par adrienollier

# les tranches en ordre décroissant
TaxRate = (
    (150_000, 45),
    (45_000, 40),
    (11_500, 20),
    (0, 0),
)

def taxes(income):
    """
    U.K. income taxes calculator
    https://www.gov.uk/income-tax-rates
    """
    due = 0
    for floor, rate in TaxRate:
        if income > floor:
            due += (income - floor) * rate / 100
            income = floor
    return int(due)
```

taxes (bis) - Semaine 4 Séquence 3

```
1
2   # cette solution est plus pataude; je la retiens
3   # parce qu'elle montre un cas de for .. else ..
4   # qui ne soit pas trop tiré par les cheveux
5   # quoique
6
7   bands = [
8       # à partir de 0. le taux est nul
9       (0, 0.),
10      # jusqu'à 11 500 où il devient de 20%
11      (11_500, 20/100),
12      # etc.
13      (45_000, 40/100),
14      (150_000, 45/100),
15  ]
16
17  def taxes_bis(income):
18      """
19      Utilise un for avec un else
20      """
21      amount = 0
22
23      # en faisant ce zip un peu étrange, on va
24      # considérer les couples de tuples consécutifs dans
25      # la liste bands
26      for (band1, rate1), (band2, _) in zip(bands, bands[1:]):
27          # le salaire est au-delà de cette tranche
28          if income >= band2:
29              amount += (band2-band1) * rate1
30          # le salaire est dans cette tranche
31          else:
32              amount += (income-band1) * rate1
33              # du coup on peut sortir du for par un break
34              # et on ne passera pas par le else du for
35              break
36      # on ne passe ici qu'avec les salaires dans la dernière tranche
37      # en effet pour les autres on est sorti du for par un break
38      else:
39          band_top, rate_top = bands[-1]
40          amount += (income - band_top) * rate_top
41      return int(amount)
```

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]))
```

distance (bis) - Semaine 4 Séquence 6

```
def distance_bis(*args):
    """
    Idem mais avec une expression génératrice
    """
    # on n'a pas encore vu cette forme - cf Semaine 5
    # mais pour vous donner un avant-goût d'une expression
    # génératrice:
    # on peut faire aussi comme ceci
    # observez l'absence de crochets []
    # la différence c'est juste qu'on ne
    # construit pas la liste des carrés,
    # car on n'en a pas besoin
    # et donc un itérateur nous suffit
    return math.sqrt(sum(x**2 for x in args))
```

numbers - Semaine 4 Séquence 6

```
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
# (où il n'y 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
        # on pourrait par contre faire juste sum(liste)
        # car le défaut pour start c'est 0
        # dit autrement, sum([]) retourne bien 0
        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(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}
```

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)
```

shipdict - Semaine 6 Séquence 4

```
1
2   # helpers - used for verbose mode only
3   # could have been implemented as static methods in Position
4   # but we had not seen that at the time
5
6
7   def d_m_s(f):
8       """
9       make a float readable; e.g. transform 2.5 into 2.30'00''
10      we avoid using the degree sign to keep things simple
11      input is assumed positive
12      """
13      d = int(f)
14      m = int((f - d) * 60)
15      s = int((f - d) * 3600 - 60 * m)
16      return "{:02d}.{:02d}'{:02d}''".format(d, m, s)
17
18
19  def lat_d_m_s(f):
20      """
21      degree-minute-second conversion on a latitude float
22      """
23      if f >= 0:
24          return "{} N".format(d_m_s(f))
25      else:
26          return "{} S".format(d_m_s(-f))
27
28
29  def lon_d_m_s(f):
30      """
31      degree-minute-second conversion on a longitude float
32      """
33      if f >= 0:
34          return "{} E".format(d_m_s(f))
35      else:
36          return "{} W".format(d_m_s(-f))
```

shipdict (suite) - Semaine 6 Séquence 4

```
1
2
3  class Position(object):
4      "a position atom with timestamp attached"
5
6      def __init__(self, latitude, longitude, timestamp):
7          "constructor"
8          self.latitude = latitude
9          self.longitude = longitude
10         self.timestamp = timestamp
11
12 # all these methods are only used when merger.py runs in verbose mode
13     def lat_str(self):
14         return lat_d_m_s(self.latitude)
15
16     def lon_str(self):
17         return lon_d_m_s(self.longitude)
18
19     def __repr__(self):
20         """
21         only used when merger.py is run in verbose mode
22         """
23         return f"<{self.lat_str()} {self.lon_str()} @ {self.timestamp}>"
```

shipdict (suite) - Semaine 6 Séquence 4

```
1
2
3   class Ship(object):
4       """
5       a ship object, that requires a ship id,
6       and optionnally a ship name and country
7       which can also be set later on
8
9       this object also manages a list of known positions
10      """
11
12      def __init__(self, id, name=None, country=None):
13          "constructor"
14          self.id = id
15          self.name = name
16          self.country = country
17          # this is where we remember the various positions over time
18          self.positions = []
19
20      def add_position(self, position):
21          """
22          insert a position relating to this ship
23          positions are not kept in order so you need
24          to call ‘sort_positions‘ once you’re done
25          """
26          self.positions.append(position)
27
28      def sort_positions(self):
29          """
30          sort list of positions by chronological order
31          """
32          self.positions.sort(key=lambda position: position.timestamp)
```

```
1
2
3   class ShipDict(dict):
4       """
5       a repository for storing all ships that we know about
6       indexed by their id
7       """
8
9       def __init__(self):
10          "constructor"
11          dict.__init__(self)
12
13      def __repr__(self):
14          return f"<ShipDict instance with {len(self)} ships>"
15
16      def is_abbreviated(self, chunk):
17          """
18          depending on the size of the incoming data chunk,
19          guess if it is an abbreviated or extended data
20          """
21          return len(chunk) <= 7
22
23      def add_abbreviated(self, chunk):
24          """
25          adds an abbreviated data chunk to the repository
26          """
27          id, latitude, longitude, *_, timestamp = chunk
28          if id not in self:
29              self[id] = Ship(id)
30          ship = self[id]
31          ship.add_position(Position(latitude, longitude, timestamp))
32
33      def add_extended(self, chunk):
34          """
35          adds an extended data chunk to the repository
36          """
37          id, latitude, longitude = chunk[:3]
38          timestamp, name = chunk[5:7]
39          country = chunk[10]
40          if id not in self:
41              self[id] = Ship(id)
42          ship = self[id]
43          if not ship.name:
44              ship.name = name
45              ship.country = country
46          self[id].add_position(Position(latitude, longitude, timestamp))
```

shipdict (suite) - Semaine 6 Séquence 4

```
    def add_chunk(self, chunk):
        """
        chunk is a plain list coming from the JSON data
        and be either extended or abbreviated

        based on the result of is_abbreviated(),
        gets sent to add_extended or add_abbreviated
        """
        if self.is_abbreviated(chunk):
            self.add_abbreviated(chunk)
        else:
            self.add_extended(chunk)

    def sort(self):
        """
        makes sure all the ships have their positions
        sorted in chronological order
        """
        for id, ship in self.items():
            ship.sort_positions()

    def clean_unnamed(self):
        """
        Because we enter abbreviated and extended data
        in no particular order, and for any time period,
        we might have ship instances with no name attached
        This method removes such entries from the dict
        """
        # we cannot do all in a single loop as this would amount to
        # changing the loop subject
        # so let us collect the ids to remove first
        unnamed_ids = {id for id, ship in self.items()
                       if ship.name is None}
        # and remove them next
        for id in unnamed_ids:
            del self[id]
```

shipdict (suite) - Semaine 6 Séquence 4

```
    def ships_by_name(self, name):
        """
        returns a list of all known ships with name <name>
        """
        return [ship for ship in self.values() if ship.name == name]

    def all_ships(self):
        """
        returns a list of all ships known to us
        """
        # we need to create an actual list because it
        # may need to be sorted later on, and so
        # a raw dict_values object won't be good enough
        return self.values()

```

stairs - Semaine 7 Séquence 05

```
def stairs(k):
    """
    la pyramide en escaliers telle que décrite dans l'énoncé
    """
    # on calcule n
    n = 2 * k + 1
    # on calcule les deux tableaux d'indices
    # tous les deux de dimension n
    ix, iy = np.indices((n, n))
    # il n'y a plus qu'à appliquer la formule qui va bien
    return 2 * k - (np.abs(ix - k) + np.abs(iy - k))
```

stairs (bis) - Semaine 7 Séquence 05

```
def stairs_bis(k):
    """
    Bien sûr on peut préciser le type mais ce n'est pas
    réellement nécessaire ici
    """
    n = 2 * k + 1
    ix, iy = np.indices((n, n), dtype=np.int8)
    return 2 * k - (np.abs(ix - k) + np.abs(iy - k))
```