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
```

libelle - Semaine 2 Séquence 6

```
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 presenter le rang
    msg_rang = "1er" if rang == "1" \
               else "2nd" if rang == "2" \
                    else "{}-ème".format(rang)
    return f"{prenom}.{nom} ({msg_rang})"
```

carre - Semaine 2 Séquence 6

```
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])
```

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) ]
```

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
    # et il faut regarder aussi si a divise b
    return a % b == 0 or b % a == 0
```

divisible (bis) - Semaine 2 Séquence 6

```
def divisible_bis(a, b):
    if a % b == 0:
        return True
    if b % a == 0:
        return True
    return False
```

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 (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
```

multi_tri - Semaine 2 Séquence 7

```
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 2 Séquence 7

```
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
```

produit_scalaire - Semaine 2 Séquence 7

```
def produit_scalaire(X, Y):
    """
    retourne le produit scalaire
    de deux listes de même taille
    """
    # initialisation du résultat
    scalaire = 0
    # ici encore avec zip() on peut faire correspondre
    # les X avec les Y
    for x, y in zip(X, Y):
        scalaire += x * y
    # on retourne le résultat
    return scalaire
```

produit_scalaire (bis) - Semaine 2 Séquence 7

```
# Il y a plein d'autres solutions qui marchent aussi
# en voici notamment une qui utilise la fonction builtin sum
# (que nous n'avons pas encore vue, nous la verrons en semaine 4)
# en voici toutefois un avant-goût: la fonction sum est très pratique
# pour faire la somme de toute une liste de valeurs
def produit_scalaire_bis(X, Y):
    return sum([x * y for x, y in zip(X, Y)])

```

produit_scalaire (ter) - Semaine 2 Séquence 7

```
# Et encore une: celle-ci par contre est assez peu "pythonique"
# on aime bien en général éviter les boucles du genre
# for i in range(l)
#     ... l[i]
def produit_scalaire_ter(X, Y):
    scalaire = 0
    n = len(X)
    for i in range(n):
        scalaire += X[i] * Y[i]
    return scalaire
```

aplatir - Semaine 2 Séquence 7

```
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 2 Séquence 7

```
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 2 Séquence 7

```
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 2 Séquence 7

```
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}
```

comptage - Semaine 3 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, encoding='utf-8') as input:
        # on ouvre la sortie en écriture
        with open(out_filename, "w", encoding='utf-8') as output:
            # initialisations
            total_words = 0
            total_chars = 0
            # pour toutes les lignes du fichier d'entrée
            # le numéro de ligne commence à 1
            for lineno, line in enumerate(input, 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
            # lineno est une variable de boucle, elle "fuite"
            # on peut donc utiliser sa dernière valeur
            # mais remarquez que ce code ne fonctionnerait
            # pas sur un fichier vide, ou on aurait lineno non définie
            output.write("{}:{}:{}\n"
                         .format(lineno, total_words, total_chars))
```

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
```

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)
        }
```

parse_graph - Semaine 3 Séquence 4

```
from collections import defaultdict

def parse_graph(filename):
    g = defaultdict(list)
    with open(filename) as f:
        for line in f:
            begin, value, end = line.split()
            g[begin].append( (end, int(value)))
    return g
```

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
```

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]))
```

doubler_premier - Semaine 4 Séquence 6

```
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 4 Séquence 6

```
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 6

```
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 6

```
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]
```

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   def d_m_s(f):
6       """
7       make a float readable; e.g. transform 2.5 into 2.30'00''
8       we avoid using the degree sign to keep things simple
9       input is assumed positive
10      """
11      d = int (f)
12      m = int((f-d)*60)
13      s = int( (f-d)*3600 - 60*m)
14      return "{:02d}.{:02d}'{:02d}''".format(d,m,s)
15
16  def lat_d_m_s(f):
17      """
18      degree-minute-second conversion on a latitude float
19      """
20      if f >= 0:
21          return "{} N".format(d_m_s(f))
22      else:
23          return "{} S".format(d_m_s(-f))
24
25  def lon_d_m_s(f):
26      """
27      degree-minute-second conversion on a longitude float
28      """
29      if f >= 0:
30          return "{} E".format(d_m_s(f))
31      else:
32          return "{} W".format(d_m_s(-f))
```

shipdict (suite) - Semaine 6 Séquence 4

```
class Position(object):
    "a position atom with timestamp attached"

    def __init__(self, latitude, longitude, timestamp):
        "constructor"
        self.latitude = latitude
        self.longitude = longitude
        self.timestamp = timestamp

# all these methods are only used when merger.py runs in verbose mode
    def lat_str(self):
        return lat_d_m_s(self.latitude)
    def lon_str(self):
        return lon_d_m_s(self.longitude)

    def __repr__(self):
        """
        only used when merger.py is run in verbose mode
        """
        return "<{} {} @ {}>".format(self.lat_str(),
                                     self.lon_str(), self.timestamp)
```

shipdict (suite) - Semaine 6 Séquence 4

```
class Ship(object):
    """
    a ship object, that requires a ship id,
    and optionnally a ship name and country
    which can also be set later on

    this object also manages a list of known positions
    """
    def __init__(self, id, name=None, country=None):
        "constructor"
        self.id = id
        self.name = name
        self.country = country
        # this is where we remember the various positions over time
        self.positions = []

    def add_position(self, position):
        """
        insert a position relating to this ship
        positions are not kept in order so you need
        to call ‘sort_positions‘ once you’re done
        """
        self.positions.append(position)

    def sort_positions(self):
        """
        sort list of positions by chronological order
        """
        self.positions.sort(key=lambda position: position.timestamp)
```

shipdict (suite) - Semaine 6 Séquence 4

```
class ShipDict(dict):
    """
    a repository for storing all ships that we know about
    indexed by their id
    """
    def __init__(self):
        "constructor"
        dict.__init__(self)

    def __repr__(self):
        return "<ShipDict instance with {} ships>".format(len(self))

    def is_abbreviated(self, chunk):
        """
        depending on the size of the incoming data chunk,
        guess if it is an abbreviated or extended data
        """
        return len(chunk) <= 7

    def add_abbreviated(self, chunk):
        """
        adds an abbreviated data chunk to the repository
        """
        id, latitude, longitude, *_, timestamp = chunk
        if id not in self:
            self[id] = Ship(id)
        ship = self[id]
        ship.add_position (Position (latitude, longitude, timestamp))

    def add_extended(self, chunk):
        """
        adds an extended data chunk to the repository
        """
        id, latitude, longitude = chunk[:3]
        timestamp, name = chunk[5:7]
        country = chunk[10]
        if id not in self:
            self[id] = Ship(id)
        ship = self[id]
        if not ship.name:
            ship.name = name
            ship.country = country
        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()

```