Tous les corrigés

Table des matières

composite - Semaine 2 Séquence 3

```
# Pour calculer inconnue, on extrait une sous-chaine de composite
# qui commence a l'index len(connue)
# qui se termine a l'index len(composite)-len(connue)
# ce qui donne en utilisant une slice
inconnue = composite [ len(connue) : len(composite)-len(connue) ]
#
# on peut aussi faire encore plus simplement
inconnue = composite [ len(connue) : -len(connue) ]
```

divisible - Semaine 2 Séquence 7

```
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
```

spam - Semaine 2 Séquence 7

```
def spam(l):
    """
Prend en argument une liste, et retourne la liste modifiée:
 * taille paire: on intervertit les deux premiers éléments
 * taille impaire, on retire le dernier élément
"""
    # si la liste est vide il n'y a rien à faire
    if not l:
        pass
    # si la liste est de taille paire
    elif len(l)%2 == 0:
        # on intervertit les deux premiers éléments
        l[0], l[1] = l[1], l[0]
    # si elle est de taille impaire
    else:
        # on retire le dernier élément
        l.pop()
    # et on n'oublie pas de retourner la liste dans tous les cas
    return l
```

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
```

liste_racines - Semaine 2 Séquence 7

```
from math import e, pi

def liste_racines(p):
    "retourne la liste des racines p-ièmes de l'unité"
    # une simple compréhension fait l'affaire
    # souvenez vous que 1j c'est notre 'i' complexe
    return [e**((2*pi*1j*n)/p) for n in range(p)]

# Il est tout à fait possible aussi de construire les racines pas à pas
# C'est un peu moins élégant mais ça fonctionne très bien aussi
def liste_racines_bis(p):
    "retourne la liste des racines p-ièmes de l'unité"
    # on va construire le résultat petit à petit
    # en partant d'une liste vide
    resultat = []
    # pour chaque n dans {0 .. p-1}
    for n in range(p):
        # on ajoute dans le résultat la racine d'ordre n
        resultat.append(e**((2*pi*1j*n)/p))
    # et on retourne le résultat
    return resultat
```

produit_scalaire - Semaine 2 Séquence 7

```
def produit_scalaire(X,Y):
    # initialisation du resultat
    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

# 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):
    """retourne le produit scalaire de deux listes de même taille"""
    return sum([x*y for x, y in zip(X, Y)])
```

affichage - Semaine 2 Séquence 8

```
# un élève a remarqué très justement que ce code ne fait pas
# exactement ce qui est demandé, en ce sens qu'avec
# l'entrée correspondant à Ted Mosby on obtient A:><
# je préfère toutefois publier le code qui est en
# service pour la correction en ligne, et vous laisse
# le soin de l'améliorer si vous le souhaitez
def affichage(s):
    # pour ignorer les espaces et les tabulations
    # le plus simple est de les enlever
    s=s.replace(' ', '').replace('\t','')
    # la liste des mots séparés par une virgule
    # nous est donnée par un simple appel à split
    mots = s.split(',')
    # si on n'a même pas deux mots, on retourne None
    if len(mots) < 2:
        return None
    # maintenant qu'on sait qu'on a deux mots
    # on peut extraire le prénom et le nom
    prenom = mots.pop(0)
    nom = mots.pop(0)
    # on veut afficher "??" si l'âge est inconnu
    age = "??"
    # mais si l'âge est précisé dans la ligne
    if len(mots) >= 2:
        # alors on le prend
        age = mots.pop(1)
    # il ne reste plus qu'à formater
    return "N:>{}< P:>{}< A:>{}<".format(nom, prenom, age)
```

carre - Semaine 2 Séquence 8

```
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(";") 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])
```

index - Semaine 3 Séquence 2

```
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 - Semaine 3 Séquence 2

```
def index2(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 2

```
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 des listes)
    """
    # on initialise le résultat avec un dictionnaire vide
    result = {}
    # pour les données étendues
    for ship in extended:
        # on affecte les 6 premiers champs
        # et on ignore les champs de rang 6 et au delà
        id, latitude, longitude, timestamp, name, country = ship[:6]
        # 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 - Semaine 3 Séquence 2

```
def merge2(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 - Semaine 3 Séquence 2

```
def merge3(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)
        }
```

diff - Semaine 3 Séquence 3

```
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
```

decode_zen - Semaine 3 Séquence 5

```
# 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 dans 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 - Semaine 3 Séquence 5

```
# une autre version qui marche aussi, en utilisant
# dict.get(key, default)
def decode_zen2(this):
    return "".join([this.d.get(c, c) for c in this.s])
```

dispatch1 - Semaine 3 Séquence 7

```
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 3 Séquence 7

```
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
```

comptage - Semaine 4 Séquence 1

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

pgcd - Semaine 4 Séquence 2

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

numbers - Semaine 4 Séquence 3

```
from operator import mul

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

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

validation - Semaine 4 Séquence 3

```
def validation(f, g, entrees):
    """
    retourne une liste de booléens, un par entree dans entrees
    qui indique si f(entree) == g(entree)
    """
    # on vérifie pour chaque entrée si f et g retournent
    # des résultats égaux avec ==
    # et on assemble le tout avec une comprehension de liste
    return [f(entree) == g(entree) for entree in entrees]
```

aplatir - Semaine 4 Séquence 4

```
def aplatir(conteneurs):
    "retourne une liste des éléments des éléments de conteneurs"
    # on peut concaténer les éléments de deuxième niveau
    # par une simple imbrication de deux compréhensions de liste
    return [element for conteneur in conteneurs for element in conteneur]
```

alternat - Semaine 4 Séquence 4

```
def alternat(l1, l2):
    "renvoie une liste des éléments pris un sur deux dans l1 et dans l2"
    # pour réaliser l'alternance on peut combiner zip avec aplatir
    # telle qu'on vient de la réaliser
    return aplatir(zip(l1, l2))
```

alternat - Semaine 4 Séquence 4

```
def alternat2(l1, l2):
    "une deuxième version de alternat"
    # la même idée mais directement, sans utiliser aplatir
    return [element for conteneur in zip(l1, l2) for element in conteneur]
```

intersect - Semaine 4 Séquence 4

```
def intersect(A, B):
    """
    prend en entrée deux listes de tuples de la forme
    (entier, valeur)
    renvoie la liste des valeurs associées dans A ou B
    aux entiers présents dans A et B
    """
    # pour montrer un exemple de fonction locale:
    # une fonction qui renvoie l'ensemble des entiers
    # présents dans une des deux listes d'entrée
    def values(S):
        return {i for i, val in S}
    # on l'applique à A et B
    val_A = values(A)
    val_B = values(B)
    #
    # bien sûr on aurait pu écrire directement
    # val_A = {i for i, val in A}
    # val_B = {i for i, val in B}
    #
    # les entiers présents dans A et B
    # avec une intersection d'ensembles
    common_keys = val_A & val_B
    # et pour conclure on fait une union sur deux
    # compréhensions d'ensembles
    return {vala for a, vala in A if a in common_keys} \
         | {valb for b, valb in B if b in common_keys}
```

distance - Semaine 4 Séquence 8

```
import math

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

doubler_premier - Semaine 4 Séquence 8

```
def doubler_premier(f, first, *args):
    """
    renvoie le résultat de la fonction f appliquée sur
    f(2 * first, *args)
    """
    # une fois qu'on a écrit la signature on a presque fini le travail
    # en effet on a isolé la fonction, son premier argument, et le reste
    # des arguments
    # il ne reste qu'à appeler f, après avoir doublé first
    return f(2*first, *args)
```

doubler_premier2 - Semaine 4 Séquence 8

```
def doubler_premier2(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_premier2(muln, x=1, y=2)
# ne marche pas car on n'a pas les deux arguments requis
# par doubler_premier2
#
# et pour écrire, disons doubler_permier3, qui marcherait aussi comme cela
# il faudrait faire une hypothèse sur le nom du premier argument...
```

validation2 - Semaine 4 Séquence 8

```
def validation2(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 validation, 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]
```

RPCProxy - Semaine 5 Séquence 6

```
# une troisième implémentation de RPCProxy

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

class RPCProxy(object):

    def __init__(self, url, login, password):
        self.url = url
        self.login = login
        self.password = password

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

```

shipdict - Semaine 5 Séquence 6

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from __future__ import print_function

# helpers - used for the verbose mode only
# could have been implemented as static methods in Position
# but we had not seen that at the time
def d_m_s(f):
    """
    make a float readable; e.g. transform 2.5 into 2.30'00''
    we avoid using Â° to keep things simple
    input is assumed positive
    """
    d = int (f)
    m = int((f-d)*60)
    s = int( (f-d)*3600 - 60*m)
    return "{:02d}.{:02d}'{:02d}''".format(d,m,s)

def lat_d_m_s(f):
    if f>=0:        return "{} N".format(d_m_s(f))
    else:           return "{} S".format(d_m_s(-f))

def lon_d_m_s(f):
    if f>=0:        return "{} E".format(d_m_s(f))
    else:           return "{} W".format(d_m_s(-f))
```

shipdict-suite - Semaine 5 Séquence 6

```
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 5 Séquence 6

```
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 5 Séquence 6

```
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 5 Séquence 6

```
    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.iteritems():
            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.iteritems()
                        if ship.name is None }
        # and remove them next
        for id in unnamed_ids:
            del self[id]
```

shipdict-suite - Semaine 5 Séquence 6

```
    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
        """
        return self.values()
```

regexp_pythonid - Semaine 6 Séquence 6

```
# 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
regexp_pythonid = "[a-zA-Z_]\w*"
```

regexp_pythonid - Semaine 6 Séquence 6

```
# on peut aussi bien sûr l'écrire en clair
regexp_pythonid2 = "[a-zA-Z_][a-zA-Z0-9_]*"
```

regexp_specials - Semaine 6 Séquence 6

```
# il faut commencer par exactement 2 underscores
# donc le caractère suivant doit être une lettre
# ensuite on peut mettre ce qu'on veut comme alphanumérique,
# mais avant les deux derniers underscores on ne peut pas avoir
# un underscore
# enfin pour traiter le cas où la partie centrale est réduite
# à un seul caractère, on met une option - avec ()?
regexp_specials = "__[a-zA-Z](\w*[a-zA-Z0-9])?__"
```

regexp_url - Semaine 6 Séquence 6

```
# 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
regexp_url = i_flag + protos + "://" + user + hostname + port + '/' + path
```