Corrigés de la semaine 5

RPCProxy - Semaine 5 Séquence 6

```
# une troisième implémentation de RPCProxy

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

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

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

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

shipdict - Semaine 5 Séquence 6

```
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 the degree sign 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()

    def sort_ships_by_name (self, ships):
        """
        New in version 2.0

        given a list of ships, returns a sorted version
        this uses sorted() so a shallow copy is returned

        sorting criteria is first on names, and then with
        identical ship names use ship id instead

        """
        # to be completely deterministic, we cannot use only
        # key=lambda ship: ship.name
        # because of duplicate names in the fleet
        # use good old cmp instead
        def ship_compare (s1, s2):
            return -1 if s1.name < s2.name \
                else 1 if s1.name > s2.name \
                     else s1.id - s2.id
        return sorted (ships, cmp = ship_compare)

```