

Corrigés de la semaine 5

```

1  # une troisième implémentation de RPCProxy
2
3  class Forwarder(object):
4      """
5      Une instance de la classe Forwarder est un callable
6      qui peut être utilisée comme une méthode sur l
7      class RPCProxy
8      """
9      def __init__(self, rpc_proxy, methodname):
10         """
11         le constructeur mémorise l'instance de RPCProxy
12         et le nom de la méthode qui a été appelée
13         """
14         self.methodname = methodname
15         self.rpc_proxy = rpc_proxy
16
17     def __call__(self, *args):
18         """
19         en rendant cet objet callable, on peut l'utiliser
20         comme une méthode de RPCProxy
21         """
22         print "Envoi à {} \nde la fonction {} -- args= {}".\
23             format(self.rpc_proxy.url, self.methodname, args)
24         return "retour de la fonction " + self.methodname
25
26 class RPCProxy(object):
27     """
28     Une troisième implémentation de RPCProxy qui sous-traite
29     à une classe annexe 'Forwarder' qui se comporte comme
30     une *factory* de méthodes
31     """
32     def __init__(self, url, login, password):
33         self.url = url
34         self.login = login
35         self.password = password
36
37     def __getattr__(self, methodname):
38         """
39         Crée à la volée une instance de Forwarder
40         correspondant à 'methodname'
41         """
42         return Forwarder(self, methodname)

```

```

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

```

```

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

```

```

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

```

```

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

```

```

1  def add_chunk(self, chunk):
2      """
3      chunk is a plain list coming from the JSON data
4      and be either extended or abbreviated
5
6      based on the result of is_abbreviated(),
7      gets sent to add_extended or add_abbreviated
8      """
9      if self.is_abbreviated(chunk):
10         self.add_abbreviated(chunk)
11     else:
12         self.add_extended(chunk)
13
14     def sort(self):
15         """
16         makes sure all the ships have their positions
17         sorted in chronological order
18         """
19         for id, ship in self.iteritems():
20             ship.sort_positions()
21
22     def clean_unnamed(self):
23         """
24         Because we enter abbreviated and extended data
25         in no particular order, and for any time period,
26         we might have ship instances with no name attached
27         This method removes such entries from the dict
28         """
29         # we cannot do all in a single loop as this would amount to
30         # changing the loop subject
31         # so let us collect the ids to remove first
32         unnamed_ids = { id for id, ship in self.iteritems()
33                         if ship.name is None }
34         # and remove them next
35         for id in unnamed_ids:
36             del self[id]

```

```

1  def ships_by_name(self, name):
2      """
3      returns a list of all known ships with name <name>
4      """
5      return [ ship for ship in self.values() if ship.name == name ]
6
7  def all_ships(self):
8      """
9      returns a list of all ships known to us
10     """
11     return self.values()
12
13  def sort_ships_by_name (self, ships):
14     """
15     New in version 2.0
16
17     given a list of ships, returns a sorted version
18     this uses sorted() so a shallow copy is returned
19
20     sorting criteria is first on names, and then with
21     identical ship names use ship id instead
22
23     """
24     # to be completely deterministic, we cannot use only
25     # key=lambda ship: ship.name
26     # because of duplicate names in the fleet
27     # use good old cmp instead
28     def ship_compare (s1, s2):
29         return -1 if s1.name < s2.name \
30             else 1 if s1.name > s2.name \
31             else s1.id - s2.id
32     return sorted (ships, cmp = ship_compare)
33

```