# W5-S6-E1-classes

December 15, 2014

## 1 Exercice sur l'utilisation des classes

### 1.0.1 Introduction

**Objectifs de l'exercice** Maintenant que vous avez un bagage qui couvre toutes les bases du langage, cette semaine nous ne ferons qu'un seul exercice de taille un peu plus réaliste. Vous devez écrire quelques classes, que vous intégrez ensuite dans un code écrit pas nos soins.

L'exercice comporte donc autant une part lecture qu'une part écriture. Vous êtes également incités à améliorer autant que possible votre environnement de travail sur votre ordinateur.

**Objectifs de l'application** Dans le prolongement des exercices de la semaine 3 sur les données maritimes, l'application dont il est question ici fait principalement ceci : * agréger des données obtenues auprès de marinetraffic, * et produire en sortie * un fichier texte qui liste par ordre alphabétique les bateaux concernés, et le nombre de positions trouvées pour chacun, * et un fichier KML, pour exposer les trajectoires trouvées à google earth, maps ou autre outil similaire; dans la version 2.0 de l'exercice, les données générées dans ce fichier KML sont également dans l'ordre alphabétique des noms de bateau.

Plus précisément, pour les deux sorties on tries les bateaux selon le critère suivant : * ordre alphabétique, * et ordre sur les `id` en cas d'ex-aequo.

Voici à quoi ressemble le fichier KML obtenu avec les données que nous fournissons, une fois ouvert sous google earth :

**Choix d'implémentation** En particulier, dans cet exercice nous allons voir comment on peut gérer des données sous forme d'instances de classes plutôt que de types de base. Cela résonne avec la discussion commencée en Semaine 3, Séquence "Les dictionnaires", dans le complément "record-et-dictionnaire".

Dans les exercices de cette semaine-là nous avions uniquement utilisé des types 'standard' comme listes, tuples et dictionnaires pour modéliser les données, cette semaine nous allons faire le choix inverse et utiliser plus souvent des (instances de) classes.

**Principe de l'exercice** On a écrit une application complète, constituée de 4 modules ; on vous donne le code de 3 de ces modules et vous devez écrire le module manquant.

**Correction** Tout d'abord nous fournissons un jeu de données d'entrées. De plus, l'application vient avec son propre système de vérification, qui est très rustique. Il consiste à comparer, une fois les sorties produites, leur contenu avec les sorties de référence, qui ont été obtenues avec notre version de l'application.

Du coup, le fait de disposer de google earth sur votre ordinateur n'est pas strictement nécessaire, on ne s'en sert pas à proprement parler pour l'exercice.

---

### 1.0.2 Mise en place

**Partez d'un directory vierge** Pour commencer, créez-vous un répertoire pour travailler à cet exercice

**Les données** Commencez par y installer les données que nous publions dans les formats suivants : * format tar * format tgz * format zip

Une fois installées, ces données doivent se trouver dans un sous-répertoire `json/` qui contient 133 fichiers

```
json/2013-10-01-00-00--t=10--ext.json
...
json/2013-10-01-23-50--t=10.json
```

Comme vous pouvez le deviner, il s'agit de données sur le mouvement des bateaux dans la zone en date du 10 Octobre 2013 ; et comme vous le voyez également, on a quelques exemplaires de données étendues, mais dans la plupart des cas il s'agit de données abrégées.

**Les résultats de référence** De même il vous faut installer les résultats de référence que vous trouvez ici : * format tar * format tgz * format zip

ce qui vous doit vous donner trois fichiers

```
ALL_SHIPS.kml.ref
ALL_SHIPS.txt.ref
ALL_SHIPS-v.txt.ref
```

**Le code** Vous pouvez à présent aller chercher les 3 modules suivants : * `merger.py` * `compare.py` * `kml.py`

et les sauver dans le même répertoire.

Vous remarquerez que le code est cette fois entièrement rédigé en anglais, ce que nous vous conseillons de faire aussi souvent que possible.

Votre but dans cet exercice est d'écrire le module manquant `shipdict.py` qui permettra à l'application de fonctionner comme attendu.

---

### 1.0.3 Fonctionnement de l'application

**Comment est structurée l'application** Le point d'entrée s'appelle `merger.py`

Il utilise donc trois autres modules annexes, qui sont

- `shipdict.py`, qui implémente les classes
- `Position` qui contient une latitude, une longitude, et un timestamp,
- `Ship` qui modélise un bateau à partir de son `id` et optionnellement `name` et `country`
- `ShipDict`, qui maintient un index des bateaux (essentiellement un dictionnaire)
- `compare.py` qui implémente

- la classe `Compare` qui se charge de comparer les fichiers résultat avec leur version de référence,
- `kml.py` qui implémente
- la classe `KML` dans laquelle sont concentrées les fonctions liées à la génération de KML ; c'est notamment en fonction de nos objectifs pédagogiques que ce choix a été fait.

**Lancement** Lorsque le programme est complet et qu'il fonctionne correctement, on le lance comme ceci :

```
$ python merger.py json/*
Opening ALL_SHIPS.txt for listing all named ships
Opening ALL_SHIPS.kml for ship ALL_SHIPS
Comparing ALL_SHIPS.txt and ALL_SHIPS.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

qui comme on le voit produit

- `ALL_SHIPS.txt` qui résume, par ordre alphabétique les bateaux qu'on a trouvés et le nombre de positions pour chacun, et
- `ALL_SHIPS.kml` qui est le fichier au format `KML` qui contient toutes les trajectoires.

**Mode bavard (verbose)** On peut également lancer l'application avec l'option `--verbose` ou simplement `-v` sur la ligne de commande, ce qui donne un résultat plus détaillé. Le code KML généré reste inchangé, mais la sortie sur le terminal et le fichier de résumé sont plus étoffés :

```
$ python merger.py --verbose json/*.json
Opening json/2013-10-01-00-00--t=10--ext.json for parsing JSON
Opening json/2013-10-01-00-10--t=10.json for parsing JSON
...
Opening json/2013-10-01-23-40--t=10.json for parsing JSON
Opening json/2013-10-01-23-50--t=10.json for parsing JSON
Opening ALL_SHIPS-v.txt for listing all named ships
Opening ALL_SHIPS.kml for ship ALL_SHIPS
Comparing ALL_SHIPS-v.txt and ALL_SHIPS-v.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

À noter que dans le mode bavard toutes les positions sont listées dans le résumé au format texte, ce qui le rend beaucoup plus bavard comme vous pouvez le voir en inspectant la taille des deux fichiers de référence :

```
$ ls -l ALL_SHIPS*txt.ref v2.0
-rw-r--r--  1 parmentelat  staff  1438681 Dec  4 16:20 ALL_SHIPS-v.txt.ref
-rw-r--r--  1 parmentelat  staff    15331 Dec  4 16:20 ALL_SHIPS.txt.ref
-rw-r--r--  1 parmentelat  staff        0 Dec  4 16:21 v2.0
```

merger.py --help

```
$ merger.py --help
usage: merger.py [-h] [-v] [-s SHIP_NAME] [-z] [inputs [inputs ...]]

positional arguments:
  inputs

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Verbose mode
  -s SHIP_NAME, --ship SHIP_NAME
                        Restrict to ships by that name
  -z, --gzip            Store kml output in gzip (KMZ) format
```

**Un mot sur les données** **Attention** que le contenu détaillé des champs `extended` et `abbreviated` peut être légèrement différent de ce qu'on avait pour les exercices de la semaine 3.

Voici ce avec quoi on travaille cette fois-ci :

```
>>> extended[0]
[228317000, 48.76829, -4.334262, 75, 333, u'2013-09-30T21:54:00', u'MA GONDOLE', 30, 0, u'FGSA', u'FR',
```

c'est-à-dire

```
[ id, latitude, longitude, _, _, timestamp, name, _, _, _, country, ...]
```

et en ce qui concerne les données abrégées

```
>>> abbreviated[0]
[232005670, 49.39331, -5.939922, 33, 269, 3, u'2013-10-01T06:08:00']
```

c'est-à-dire

```
[ id, latitude, longitude, _, _, _, timestamp]
```

Il y a unicité des `id` bien entendu (deux relevés qui portent le même `id` concernent le même bateau).

Par ailleurs, les données sont tout à fait authentiques et n'ont subi aucun filtrage. Pour cette raison, il serait sans doute hasardeux de supposer l'unicité des noms de bateaux - en clair deux bateaux différents peuvent porter le même nom.

---

#### 1.0.4 Notes pour la version 2.0

La première version de l'exercice était erronée, les bateaux dans le fichier KML n'étaient pas triés et il ne vous était donc pas possible de retrouver un résultat exactement identique.

Ceci est corrigé dans la version 2.0. Si vous avez commencé à travailler avec la version 1.x voici ce qu'il vous faut savoir :

- les fichiers de référence ont été modifiés; téléchargez les à nouveau, une fois déballés vous devez trouver dans votre répertoire un fichier qui s'appelle `v2.0` pour confirmer que vous avez la bonne version
- le fichier `merger.py` a été mis à jour; téléchargez-le à nouveau et vérifiez que vous avez bien une ligne avec `# Version :  2.0`
- la classe `ShipDict` doit à présent implémenter la méthode `sort_ships_by_name` qui est appelée par `merger.py` avant de générer les fichiers de sortie. Comme indiqué dans le préambule, on trie les bateaux selon le critère suivant :
- ordre alphabétique,
- et ordre sur les `id` en cas d'ex-aequo.

---

### 1.1 Niveaux pour l'exercice

Quelque soit le niveau auquel vous choisissez de faire l'exercice, nous vous conseillons de commencer par lire intégralement les 3 modules qui sont à votre disposition, dans l'ordre * `merger.py` qui est le chef d'orchestre de toute cette affaire, * `compare.py` qui est très simple, * `kml.py` qui ne présente pas grand intérêt en soi si ce n'est pour l'utilisation de la classe `string.Template` qui peut être utile dans d'autres contextes également.

En **niveau avancé**, l'énoncé pourrait s'arrêter là ; vous lisez le code qui est fourni et vous en déduisez ce qui manque pour faire fonctionner le tout.

Vous pouvez considérer que vous avez achevé l'exercice lorsque les deux appels suivants affichent les deux dernièrs lignes avec OK

```
$ python merger.py json/*
...
Comparing ALL_SHIPS.txt and ALL_SHIPS.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

```
$ python merger.py -v json/*.json
...
Comparing ALL_SHIPS-v.txt and ALL_SHIPS-v.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

Le cas où on lance `merger.py` avec l'option bavarde est facultatif.

---

En **niveau intermédiaire**, nous vous donnons ci-dessous un extrait de ce que donne `help` sur les classes manquantes de manière à vous donner une indication de ce que vous devez écrire.

**Classe** `Position`

```
Help on class Position in module shipdict:

class Position(__builtin__.object)
 |  a position atom with timestamp attached
 |
 |  Methods defined here:
 |
 |  __init__(self, latitude, longitude, timestamp)
 |      constructor
 |
 |  __repr__(self)
 |      only used when merger.py is run in verbose mode
 |
```

**Notes** * certaines autres classes comme `KML` sont également suscpetibles d'accéder aux champs internes d'une instance de la classe `Position` en faisant simplement `position.latitude` * La classe `Position` redéfinit `__repr__`, ceci est utilisé uniquement dans la sortie en mode bavard.

**Classe** `Ship`

```
Help on class Ship in module shipdict:

class Ship(__builtin__.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
 |
 |  Methods defined here:
 |
 |  __init__(self, id, name=None, country=None)
 |      constructor
 |
 |  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
 |
 |  sort_positions(self)
 |      sort list of positions by chronological order
```

**Classe** `Shipdict`

```
Help on class ShipDict in module shipdict:
```

```
class ShipDict(__builtin__.dict)
 |  a repository for storing all ships that we know about
 |  indexed by their id
 |
 |  Method resolution order:
 |      ShipDict
 |      __builtin__.dict
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      constructor
 |
 |  __repr__(self)
 |
 |  add_abbreviated(self, chunk)
 |      adds an abbreviated data chunk to the repository
 |
 |  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
 |
 |  add_extended(self, chunk)
 |      adds an extended data chunk to the repository
 |
 |  all_ships(self)
 |      returns a list of all ships known to us
 |
 |  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
 |
 |  is_abbreviated(self, chunk)
 |      depending on the size of the incoming data chunk,
 |      guess if it is an abbreviated or extended data
 |
 |  ships_by_name(self, name)
 |      returns a list of all known ships with name <name>
 |
 |  sort(self)
 |      makes sure all the ships have their positions
 |      sorted in chronological order
 |
 |  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
|
```

**Un dernier indice** Pour éviter de la confusion, voici le code que nous utilisons pour transformer un flottant en coordonnées lisibles dans le résumé généré en mode bavard. Aussi dans la version 1.2 ce fichier de sortie est purement ASCII et ne contient plus de caractère exotique comme le degré (°).

```
def d_m_s(f):
    """
    make a float readable; e.g. transform 2.5 into 2.30'00''
    we avoid using ° 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)
```