W5-S4-C2-Complement-module-datetime

December 15, 2014

1 Un exemple de classes de la librairie standard

1.1 Complément - niveau basique

1.1.1 Le module `time`

En semaine 3, lorsque nous avons étudié la performance des itérateurs, nous avions introduit le module `time`. Il s'agit d'une interface de très bas niveau avec l'OS, et en guise de rappel :

```
In []: import time

       # on obtient l'heure courante sous la forme d'un flottant
       t_now = time.time()
       # et pour calculer l'heure qu'il sera dans trois heures on fait
       t_later = t_now + 3 * 3600
```

Nous sommes donc ici clairement dans une approche non orientée objet; on manipule des types de base, ici le type flottant :

```
In []: print 'dans trois heures', t_later
```

Et comme on le voit, les calculs se font sous une forme pas très lisible. En effet, la méthode `time` retourne le nombre de secondes écoulées depuis le 1 janvier 1970 à l'instant de l'appel. Pour rendre ce nombre de secondes plus lisible, on utilise des conversions, pas vraiment explicites non plus; voici par exemple un appel à `gmtime` qui convertit le flottant obtenu par la méthode `time()` en heure UTC (`gm` est pour Greenwhich Meridian) :

```
In []: struct_later = time.gmtime(t_later)
       print struct_later
```

Et on met en forme ce résultat en utilisant des méthodes comme, par exemple, `strftime()` pour afficher l'heure UTC dans 3 heures :

```
In []: print 'heure UTC dans trois heures', time.strftime("%Y-%m-%d at %H:%M", struct_later)
```

1.1.2 Le module datetime

Voyons à présent, par comparaison, comment ce genre de calculs se présente lorsqu'on utilise la programmation par objets.

Le module `datetime` expose un certain nombre de classes, que nous illustrons brièvement avec les classes `datetime` (qui modélise la date et l'heure d'un instant) et `timedelta` (qui modélise une durée).

La première remarque qu'on peut faire, c'est qu'avec le module `time` on manipulait un flottant pour représenter ces deux sortes d'objets (instant et durée); avec deux classes différentes notre code va être plus clair quant à ce qui est réellement représenté.

Le code ci-dessus s'écrirait alors, en utilisant le module `datetime`

```
In []: from datetime import datetime, timedelta

       dt_now = datetime.now()
       dt_later = dt_now + timedelta(hours=3)
```

Vous remarquez que c'est déjà un peu plus clair.

Voyez aussi qu'on n'a déjà moins besoin de s'escrimer pour en avoir un aperçu lisible

```
In []: print 'maintenant', dt_now
```

```
In []: print 'dans trois heures',dt_later.strftime("%Y-%m-%d at %H:%M")
```

Je vous renvoie au détail du module, et notamment ici, pour le détail des options de formatage disponibles au travers de `strftime`.

1.1.3 Conclusion

Une partie des inconvénients du module `time` vient certainement du parti-pris de l'efficacité. De plus, c'est un module très ancien, mais auquel on ne peut guère toucher pour des raisons de compatibilité ascendante.

Par contre, le module `datetime`, tout en vous procurant un premier exemple de classes exposées par la librairie standard, vous montre, nous l'espérons, certains des avantages de la programmation orientée objet en général, et des classes de python en particulier.

Si vous devez manipuler des dates ou des heures, le module `datetime` constitue très certainement un bon candidat; voyez la documentation complète du module pour plus de précisions sur ses possibilités.

1.2 Complément - niveau intermédiaire

1.2.1 Fuseaux horaires et temps local

Le temps nous manque pour traiter ce sujet dans toute sa profondeur.

En substance, c'est un sujet assez voisin de celui des accents, en ce sens que lors d'échanges d'informations de type *timestamp* entre deux ordinateurs, il faut échanger d'une part une valeur (l'heure et la date), et d'autre part le référentiel (s'agit-il de temps UTC, ou bien de l'heure dans un fuseau horaire, et si oui lequel).

La complexité est tout de même moindre que dans le cas des accents; on s'en sort en général en convenant d'échanger systématiquement des heures UTC. Par contre, il existe une réelle diversité quant au format utilisé pour échanger ce type d'information, et cela reste une source d'erreurs assez fréquente.

1.3 Complément - niveau avancé

1.3.1 Classes et *marshalling*

Ceci nous procure une transition pour un sujet beaucoup plus général.

Nous avons évoqué, en début de semaine 4, les formats comme JSON pour échanger les données entre applications, au travers de fichiers ou d'un réseau.

On a vu, par exemple, que JSON est un format "proche des langages" en ce sens qu'il est capable d'échanger des objets de base comme des listes ou des dictionnaires entre plusieurs langages comme, par exemple, JavaScript, python ou ruby. En XML, on a davantage de flexibilité puisqu'on peut définir une syntaxe sur les données échangées.

Mais il faut être bien lucide sur le fait que, aussi bien pour JSON que pour XML, il n'est **pas possible** d'échanger entre applications des **objets** en tant que tel. Ce que nous voulons dire, c'est que ces technologies de *marshalling* prennent bien en charge le *contenu* en termes de données, mais pas les informations de type, et *a fortiori* pas non plus le code qui appartient à la classe.

Il est important d'être conscient de cette limitation lorsqu'on fait des choix de conception, notamment lorsqu'on est amené à choisir entre classe et dictionnaire pour l'implémentation de telle ou telle abstraction.

Voyons cela sur un exemple inspiré de notre fichier de données liées au trafic maritime. En version simplifiée, un bateau est décrit par trois valeurs, son identité (id), son nom et son pays d'attachement.

Nous allons voir comment on peut échanger ces informations entre, disons, deux programmes dont l'un est en python, via un support réseau ou disque.

Si on choisit de se contenter de manipuler un dictionnaire standard, avec trois couples *("id", 1000), ("name", "Toccata"), ("country", "FRA")*, on peut utiliser essentiellement tels quels les mécanismes d'encodage et décodage de, disons, JSON. En effet c'est exactement ce genre d'informations que sait gérer la couche JSON (ou XMLRPC par exemple).

Si au contraire on choisit de manipuler les données sous forme d'une classe on pourrait avoir envie d'écrire quelque chose comme ceci :

```
In []: class Bateau:
           def __init__(self, id, name, country):
               self.id = id
               self.name = name
               self.country = country

       bateau = Bateau(1000, "Toccata", "FRA")
```

Maintenant, si vous avez besoin d'échanger cet objet avec le reste du monde, en utilisant par exemple JSON, tout ce que vous allez pouvoir faire passer par ce médium, c'est la valeur des trois champs, dans un dictionnaire. Vous pouvez facilement obtenir le dictionnaire en question pour le passer à la couche d'encodage :

```
In []: vars(bateau)
```

Mais à l'autre bout de la communication il va vous falloir : * déterminer d'une manière ou d'une autre que les données échangées sont en rapport avec la classe `Bateau`, * et construire vous même un objet de cette classe, par exemple avec un code comme :

```
In []: class Bateau:
           def __init__(self, *args):
               if len(args) == 1 and isinstance(args[0], dict):
                   self.__dict__ = args[0]
               elif len(args) == 3:
                   id, name, country = args
                   self.id = id
                   self.name = name
                   self.country = country

       bateau1 = Bateau({'id': 1000, 'name': 'Leon', 'country': 'France'})
       bateau2 = Bateau(1001, 'Maluba', 'SUI' )
```

1.3.2 Conclusion

Pour reformuler ce dernier point, il n'y a pas en python l'équivalent de jmi (Java Metadata Interface) intégré à la distribution standard.

Et, là aussi, contrairement à Java, on peut écrire du code en dehors des classes. On n'est pas forcément obligé d'écrire une classe pour tout. Chaque situation doit être jugée dans son contexte naturellement, mais, de manière générale, la classe n'est pas la solution universelle ; il peut y avoir des mérites dans le fait de manipuler certaines données sous une forme allégée comme un type natif.