W3-S5-C2-modules-et-chemins

December 14, 2014

1 Où sont cherchés les modules ?

1.1 Complément - niveau basique

Pour les débutants en informatique, le plus simple est de se souvenir que si vous voulez uniquement charger vos propres modules ou packages, il suffit de les placer dans le répertoire où vous lancez la commande python. Si vous n'êtes pas sûr de cet emplacement vous pouvez le savoir en faisant:

```
import os
print 'directory courant', os.getcwd()
```

1.2 Complément - niveau intermédiaire

Dans ce complément nous allons voir, de manière générale, comment sont localisés (sur le disque dur) les modules que vous chargez dans python grâce à l'instruction `import`; nous verrons aussi où placer vos propres fichiers pour qu'ils soient accessibles à python.

Comme expliqué ici, lorsque vous importez le module `spam`, python cherche dans cet ordre: * un module *built-in* de nom `spam` - probablement écrit en C donc * ou sinon un fichier `spam.py` (ou `spam/__init.py__` s'il s'agit d'un package); pour le localiser on utilise la variable `sys.path` (c'est-à-dire donc l'attribut `path` dans le module `sys`), qui est une liste de répertoires, et qui est initialisée avec, dans cet ordre * le répertoire courant (celui dans lequel est lancé python), * la variable d'environnement `PYTHONPATH` * un certain nombre d'emplacements définis au moment de la compilation de python.

Ainsi sans action particulière de l'utilisateur, python trouve l'intégralité de la librarie standard, ainsi que les modules et packages installés dans le répertoire courant.

Voyons par exemple comment cela se présente dans l'interpréteur des notebooks

```
In []: import sys
       print sys.path
```

On remarque que le premier élément de `sys.path` est la chaîne vide, qui correspond à la recherche dans le répertoire courant. Les autres emplacements correspondent à tous les emplacements où peuvent s'installer des librairies tierces.

La variable d'environnement PYTHONPATH est utilisée ici de façon à donner la possibilité d'étendre ces listes depuis l'extérieur, et sans recompiler l'interpréteur, ni modifier les sources. Cette possibilité s'adresse donc à l'utilisateur final - ou à son administrateur système - plutôt qu'au programmeur.

En tant que programmeur par contre, vous avez la possibilité d'étendre `sys.path` avant de faire vos `import`.

Imaginons par exemple que vous avez écrit un petit outil utilitaire qui se compose d'un point d'entrée `main.py`, et de plusieurs modules `spam.py` et `eggs.py`. Vous n'avez pas le temps de packager proprement cet outil, vous voudriez pouvoir distribuer un *tar* avec ces trois fichiers python, qui puissent s'installer n'importe où (pourvu qu'ils soient tous les trois au même endroit), et que le point d'entrée trouve ses deux modules sans que l'utilisateur ait à s'en soucier.

Imaginons donc ces trois fichiers installés sur machine de l'utilisateur dans

```
/usr/share/utilitaire/
                      main.py
                      spam.py
                      eggs.py
```

Si vous ne faites rien de particulier, c'est à dire que `main.py` contient juste

```
import spam, eggs
```

Alors le programme fonctionnera **que s'il est lancé depuis** `/usr/share/utilitaire`, ce qui n'est pas du tout pratique.

Pour contourner cela on peut écrire dans `main.py` quelque chose comme

```
# on calcule le directory où est installé le point d'entrée
import os.path
directory_installation = os.path.dirname(__file__)

# et on l'ajoute au chemin de recherche des modules
import sys
sys.path.append(directory_installation)

# maintenant on peut importer spam et eggs de n'importe où
import spam, eggs
```

1.2.1 Distribuer sa propre librairie avec `distutils`

Notez bien que l'exemple précédent est **uniquement donné à titre d'illustration** pour décortiquer la mécanique d'utilisation de `sys.path`.

Ce n'est pas une technique recommandée dans le cas général. On préfère en effet de beaucoup diffuser une application python, ou une librarie, sous forme de packaging en utilisant le module distutils. Il s'agit d'un outil qui fait partie de la librairie standard, écrit en python, qui permet au programmeur d'écrire - dans un fichier qu'on appelle traditionnellement `setup.py` - le contenu de son application; grâce à quoi on peut ensuite de manière unifiée * installer l'application sur une machine à partir des sources * préparer un package de l'application * diffuser le package dans l'infrastructure PyPI * installer le package depuis PyPI en utilisant `pip`