W3-S5-C3-recapitulatif-import

December 15, 2014

1 Récapitulatif sur `import`

1.1 Complément - niveau basique

Nous allons récapituler les différentes formes d'importation, et introduire la clause `import *` - et voir pourquoi il est déconseillé de l'utiliser.

1.1.1 Importer tout un module

L'import le plus simple consiste donc à uniquement mentionner le nom du module

```
In []: import un_deux
```

Ce module se contente de définir deux fonctions de noms `un` et `deux`. Une fois l'import réalisé de cette façon on peut accéder au contenu du module en utilisant un nom de variable complet:

```
In []: # la fonction elle-même
       print un_deux.un

       un_deux.un()
```

Mais bien sûr on n'a pas de cette façon défini de nouvelle variable `un`; la seule nouvelle variable dans la portée courante est donc `un_deux`

```
In []: # dans le scope courant on peut accéder au module lui-même
       print un_deux
```

```
In []: # mais pas à la variable 'un'
       try:
           print un
       except NameError:
           print "La variable 'un' n'est pas définie"
```

1.1.2 Importer une variable spécifique d'un module

On peut également importer un ou plusieurs symboles spécifiques d'un module en faisant maintenant (avec un nouveau module du même tonneau)

```
In []: from un_deux_trois import un, deux
```

À présent nous avons deux nouvelles variables dans la portée locale

```
In []: un()
       deux()
```

Mais le module lui même n'est pas accessible

```
In []: try:
           print un_deux_trois
       except NameError:
           print "La variable 'un_deux_trois' n'est pas définie"
```

Il est important de voir que la variable locale ainsi créée, un peu comme dans le cas d'un appel de fonction, est une **nouvelle variable** qui est initialisée avec l'objet du module. Ainsi si on importe le module **et** une variable du module comme ceci

```
In []: import un_deux_trois
```

Nous avons maintenant **deux variables différentes** qui désignent la fonction `un` dans le module

```
In []: print un_deux_trois.un
       print un
```

En on peut modifier l'une **sans affecter** l'autre

```
In []: # les deux variables sont différentes
       # un n'est plus un 'alias' vers un_deux_trois.un
       un = 1
       print un_deux_trois.un
       print un
```

1.1.3 `import *`

La dernière forme d'`import` consiste à importer toutes les variables d'un module comme ceci

```
In []: from un_deux_trois_quatre import *
```

Cette forme, pratique en apparence, va donc créer dans le scope courant les variables

```
In []: un()
       deux()
       trois()
       quatre()
```

1.1.4 Quand utiliser telle ou telle forme

Les deux premières formes - import de tout un module ou de variables spécifiques - peuvent être utilisées indifféremment; souvent lorsqu'une variable est utilisée très souvent dans le code on pourra préférer la deuxième forme pour raccourcir le code.

À cet égard, citons des variantes de ces deux formes qui permettent d'utiliser des noms plus courts. Vous trouverez par exemple très souvent

```
import numpy as np
```

qui permet d'importer le module numpy mais de l'utiliser sous un nom plus court - car avec `numpy` on ne cesse d'utiliser des symboles dans le module.

Avertissement: nous vous recommandons de **ne pas utiliser la dernière forme** `import *` - sauf dans l'interpréteur interactif - car cela peut gravement nuire à la lisibilité de votre code.

python est un langage à liaison statique; cela signifie que lorsque vous concentrez votre attention sur un (votre) module, et que vous voyez une référence en lecture à un variable `spam` disons à la ligne 201, vous devez forcément trouver dans les deux cents premières lignes quelque chose comme une déclaration de `spam`, qui vous indique en gros d'où elle vient.

`import *` est une construction qui casse cette bonne propriété (pour être tout à fait exhaustif, cette bonne propriété n'est pas non plus remplie avec les fonctions *built-in* comme `len`, mais il faut vivre avec...)

Mais le point important est ceci : imaginez que dans un module vous faites plusieurs `import *` comme par exemple

```
from django.db import *
from django.conf.urls import *
```

Peu importe le contenu exact de ces deux modules, il nous suffit de savoir qu'un des deux modules expose la variable `patterns`.

Dans ce cas de figure vécu, le module utilise cette variable `patterns` sans avoir besoin de la déclarer explicitement, si bien qu'à la lecture on voit une utilisation de la variable `patterns`, mais on n'a plus aucune idée de quel module elle provient, sauf à aller lire le code correspondant...

1.2 Complément - niveau avancé

1.2.1 `import` de manière "programmative"

Étant donné la façon dont est conçue l'instruction `import`, on rencontre une limitation lorsqu'on veut, par exemple, **calculer le nom d'un module** avant de l'importer.

Si vous êtes dans ce genre de situation, reportez-vous au module `importlib` et notamment sa fonction `import_module` qui, cette fois, accepte en argument une chaîne.

Voici une illustration dans un cas simple. Nous allons importer le module `modtools` (qui fait partie de ce MOOC) de deux façons différentes et montrer que le résultat est le même:

```
In []: # on importe la fonction 'import_module' du module 'importlib'
       from importlib import import_module
       imported_modtools = import_module('mod' + 'tools')

       # on peut aussi importer modtools "normalement"
       import modtools

       # les deux objets sont identiques
       imported_modtools is modtools
```

1.2.2 `import` de manière "programmative" en python-2.6 et avant

Dans du code un peu ancien, qui daterait de 2.6 ou antérieur, vous pouvez trouver aussi des appels à la fonction *built-in* `__import__`, qui a la même sémantique que `importlib.import_module`, mais dont l'usage est en principe réservé.

Signalons toutefois que jusqu'à python-2.6, pour faire une importation de manière programmative, on n'avait pas beaucoup de choix: * soit utiliser, précisément, `__import__` * soit utiliser `exec`, qui quelque part est encore plus vilain, et pas tout à fait équivalent...