W5-S2-C1-import

December 15, 2014

1 Précisions sur l'importation

1.1 Complément - niveau basique

1.1.1 Importations multiples - rechargement

Un module n'est chargé qu'une fois De manière générale, à l'intérieur d'un interpréteur python, un module donné n'est chargé qu'une seule fois. L'idée est naturellement que si plusieurs modules différents importent le même module, (ou si un même module en importe un autre plusieurs fois) on ne paie le prix du chargement du module qu'une seule fois.

Voyons cela sur un exemple simpliste, importons un module pour la première fois

```
In []: import multiple_import
```

Ce module est très simple, comme vous pouvez le voir

```
In []: from modtools import show_module
       show_module(multiple_import )
```

Si on le charge une deuxième fois (peu importe où, dans le même module, un autre module, une fonction..), vous remarquez qu'il ne produit aucune impression

```
In []: import multiple_import
```

Ce qui confirme que le module a déjà été chargé, donc cette instruction `import` n'a aucun effet autre qu'affecter la variable `multiple_import` de nouveau à l'objet module déjà chargé. En résumé, l'instruction `import` fait l'opération d'affectation autant de fois qu'on appelle `import`, mais elle ne charge le module qu'une seule fois à la première importation.

Les raisons de ce choix Le choix de ne charger le module qu'une seule fois est motivé par plusieurs considérations.

D'une part, et principalement, cela permet à deux modules de dépendre l'un de l'autre (ou plus généralement à avoir des cycles de dépendances), sans avoir à prendre de précaution particulière. Pour ceux qui pratiquent C/C++, ceci évite d'avoir à recourir à une pratique comme

```
#ifndef _STDLIB_H_
#define _STDLIB_H_
<...>
#endif /* _STDLIB_H_ */
```

D'autre part, naturellement, cette stratégie améliore les performances.

Marginalement, `import` est une instruction comme une autre, et vous trouverez occasionnellement un avantage à l'utiliser à l'intérieur d'une fonction, **sans aucun surcoût** puisque vous ne payez le prix de l'import qu'au premier appel et non à chaque appel de la fonction.

```
def ma_fonction():
    import un_module_improbable
    ....
```

Cet usage n'est pas recommandé en général, mais de temps en temps peut s'avérer très pratique pour alléger les dépendances entre modules dans des contextes particuliers, comme du code multi-plateformes.

Les inconvénients de ce choix - la fonction `reload` L'inconvénient majeur de cette stratégie de chargement unique est perceptible dans l'interpréteur interactif pendant le développement. Nous avons vu comment IDLE traite le problème en remettant l'interpréteur dans un état vierge lorsqu'on utilise la touche F5. Mais dans l'interpréteur syntaxique "de base" on n'a pas cette possibilité.

Pour cette raison, python fournit une fonction `reload`, qui permet comme son nom l'indique de forcer le rechargement d'un module, comme ceci

```
In []: reload(multiple_import)
```

Remarquez bien que `reload` est une fonction et non une instruction comme `import` - d'où la syntaxe avec les parenthèses qui n'est pas celle de `import`.

1.2 Complément - niveau avancé

Signalons enfin, pour ceux qui sont intéressés par les détails, les deux variables suivantes.

1.2.1 `sys.modules`

L'interpréteur utilise cette variable pour conserver la trace des modules actuellement chargés.

```
In []: import sys

       import math
       math is sys.modules['math']
```

La documentation sur cette variable indique qu'il est possible de forcer le rechargement d'un module en l'enlevant de cette 'sys.modules.

```
In []: del sys.modules['multiple_import']
       import multiple_import
```

Notez enfin qu'on y trouve des modules écrits en C aussi bien qu'en python. Pour trouver le fichier python chargé par l'import du module, on peut utiliser la variable `__file__` sur le module. Notez que les modules écrits en C, comme le module `math`, n'ont pas une telle variable (chercher la discussion sur les modules dans la documentation).

```
In []: math.__file__
```

```
In []: multiple_import.__file__
```

1.2.2 `sys.builtin_module_names`

Signalons enfin cette variable qui contient le nom des modules, comme par exemple le garbage collector `gc`, qui sont implémentés en C et font partie intégrante de l'interpréteur.

```
In []: 'gc' in sys.builtin_module_names
```

1.2.3 Pour en savoir plus

Pour aller plus loin, vous pouvez lire la documentation sur l'instruction `import`