W5-S1-C1-attributs

December 15, 2014

1 Les attributs

1.1 Compléments - niveau basique

1.1.1 La notation . et les attributs

La notation `module.variable` que nous avons vue dans la vidéo est un cas particulier de la notion d'attribut, qui permet d'étendre un objet, ou si on préfère de lui accrocher des annotations.

Nous avons déjà rencontré ceci plusieurs fois, c'est exactement le même mécanisme d'attribut qui est utilisé également pour les méthodes; pour le système d'attribut il n'y a pas de différence entre `module.variable`, `module.fonction`, `objet.classe`, etc..

Nous verrons très bientôt que ce mécanisme est massivement utilisé également dans les instances de classe.

1.1.2 Les fonctions de gestion des attributs

Pour accéder programmativement aux attributs d'un objet, on dispose des 3 fonctions *builtin* `getattr`, `setattr`, et `hasattr`, que nous allons illustrer tout de suite

Lire un attribut

```
In []: import math
       # la forme la plus simple
       math.pi
```

La fonction *builtin* `getattr` permet de lire un attribut programmativement

```
In []: # si on part d'une chaîne qui désigne le nom de l'attribut
       # la formule équivalente est alors
       getattr(math, 'pi')
```

```
In []: # on peut utiliser les attributs avec la plupart des objets
       # ici nous allons le faire sur une fonction
       def foo() :
           "une fonction vide"
           pass

       # on a déjà vu certains attributs des fonctions
       print 'nom', foo.__name__, 'docstring', foo.__doc__
```

```
In []: # on peut préciser une valeur pour défaut pour le cas où l'attribut
       # n'existe pas
       getattr(foo, "attribut_inexistant", 'valeur_par_defaut')
```

Écrire un attribut

```
In []: # on peut ajouter un attribut arbitraire (toujours sur l'objet fonction)
       foo.hauteur = 100

       foo.hauteur
```

Comme pour la lecture on peut écrire un attribut programativement avec la fonction *builtin* `setattr`

```
In []: setattr(foo, "largeur", 200 )

       getattr(foo, "largeur")
```

Liste des attributs La fonction *builtin* `hasattr` permet de savoir si un objet possède ou pas un attribut:

```
In []: # pour savoir si un attribut existe
       hasattr(math, 'pi')
```

Ce qui peut aussi être retrouvé autrement, avec la fonction *builtin* `vars`

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

1.1.3 Sur quels objets

Il n'est pas disponible d'ajouter des attributs sur les types de base

```
In []: for builtin_type in (int, str, float, long, complex, tuple, dict, set, frozenset):
           obj = builtin_type()
           try:
               obj.foo = 'bar'
           except AttributeError as e:
               print "type", builtin_type.__name__, "exception", e
```

Il est par contre disponible sur virtuellement tout le reste, et notamment là où il est très utile, c'est-à-dire pour ce qui nous concerne: * modules * packages * fonctions * classes * instances

1.2 Compléments - niveau avancé

Vous pouvez vous demander pourquoi il n'est pas possible d'ajouter des attributs à un objet *builtin*. Le problème n'est pas tant d'ajouter des attributs que de modifier des attributs existants. On comprend aisément que modifier un objet *builtin* risque de corrompre le fonctionnement non seulement de votre programme d'une manière difficile à prévoir, mais aussi de tous les modules que vous utilisez. Dit autrement, les objets *builtin* sont la base de python et si vous les modifiez, vous modifiez le fonctionnement de python.

Pour rendre impossible la modification d'un objet *builtin*, python utilise, au lieu d'un dictionnaire classique pour son espace de nommage, un proxy à ce dictionnaire, appelé *dictproxy* qui interdit les modifications. On rappelle que l'espace de nommage d'un objet est presque toujours référencé par l'attribut `__dict__`. Regardons alors le type de l'espace de nommage pour la fonction `foo`

```
In []: type(foo.__dict__)
```

mais le type de l'espace de nommage pour l'objet *builtin* `int` est

```
In []: type(int.__dict__)
```