W4-S3-C1-fonctions

December 14, 2014

1 Programmation fonctionnelle

1.1 Complément - niveau basique

1.1.1 Pour résumer

La notion de programmation fonctionnelle consiste essentiellement à pouvoir manipuler les fonctions comme des objets à part entière, et à les passer en argument à d'autres fonctions, comme cela est illustré dans la vidéo.

On peut créer une fonction par l'intermédiaire de * l'*expression* `lambda:`, on obtient alors une fonction *anonyme*, * ou de l'*instruction* `def`.

Pour des raisons de syntaxe surtout, on a davantage de puissance avec `def`.

On peut calculer la liste des résultats d'une fonction sur une liste d'entrées par * `map`, éventuellement combiné à `filter` * ou par une compréhension de liste, éventuellement assortie d'un `if`.

Nous allons revoir les compréhensions de listes dans la prochaine vidéo.

1.2 Complément - niveau intermédiaire

Voici à présent quelques autres constructions classiques en programmation fonctionnelle

1.2.1 `reduce`

La fonction `reduce` permet d'appliquer une opération associative à une liste d'entrées. Pour faire simple, étant donné un opérateur binaire $\otimes$ on veut pouvoir calculer

$x_1 \otimes x_2 ... \otimes x_n$

De manière un peu moins abstraite, on suppose qu'on dispose d'une **fonction binaire** `f` qui implémente l'opérateur $\otimes$, et alors

`reduce` $(f, [x_1, ..x_n]) = f(...f(f(x_1, x_2), x_3), .., x_n)$

En fait `reduce` accepte un troisième argument - qu'il faut comprendre comme l'élément neutre de l'opérateur/fonction en question - et qui est retourné lorsque la liste en entrée est vide.

Par exemple voici - encore - une autre implémentation possible de la fonction `factoriel`.

On utilise ici le module `operator`, qui fournit sous forme de fonctions la plupart des opérateurs du langage, et notamment, dans notre cas, `operator.mul`; cette fonction retourne tout simplement le produit de ses deux arguments.

```
In []: # la multiplication, mais sous forme de fonction et non d'operateur
       from operator import mul

       def factoriel(n):
           return reduce(mul, range(1, n+1), 1)

       # ceci fonctionne aussi pour factoriel (0)
       for i in range(5):
           print i, '->', factoriel(i)
```

Cas fréquents de `reduce` Par commodité, python fournit des fonctions built-in qui correspondent en fait à des `reduce` fréquents comme la somme et les opérations `min` et `max`:

```
In []: entrees = [8, 5, 12, 4, 45, 7]

       print 'sum', sum(entrees)
       print 'min', min(entrees)
       print 'max', max(entrees)
```

1.2.2 apply

Citons enfin, à titre de curiosité historique, la fonction `apply` qui est un lointain descendant de Lisp, comme `map`, `filter` et `reduce`.

L'idée est simplement que

`apply` $(f, [x_1, ...x_n]) \Longleftrightarrow f(x_1, ...x_n)$

Ainsi par exemple

```
In []: def composite(a, b, c):
           return "{}+{}*{}".format(a, b, c)

       mon_triplet = ['spam', 'bacon', 'eggs']
       apply(composite, mon_triplet)
```

Cette fonction est toutefois très rarement utilisée, et a même été supprimée en python3. Nous verrons très bientôt que le mécanisme général de passage d'arguments en python permet de réaliser nativement l'équivalent de `apply`. Ceci sera abordé dans la dernière vidéo de cette semaine, mais pour vous en donner un avant-goût, on peut remplacer `apply` par l'appel suivant qui est à favoriser en python moderne

```
In []: composite(*mon_triplet)
```