W5-S5-C4-Complement-heritage-typage

December 14, 2014

1 Héritage, typage

1.1 Complément - niveau avancé

Dans ce complément, nous allons introduire la notion de *duck typing*, et attirer votre attention sur cette différence assez essentielle entre python et les langages statiquement typés.

1.1.1 Type concret et type abstrait

Revenons sur la notion de type et remarquons que les types peuvent jouer plusieurs rôles, comme on l'a évoqué rapidement en première semaine ; et pour reprendre des notions standard en langages de programmation nous allons distinguer deux types. 1. **type concret :** d'une part, la notion de type a bien entendu à voir avec l'implémentation ; par exemple, un compilateur C a besoin de savoir très précisément quel espace allouer à une variable, et l'interpréteur python sous-traite à la classe le soin d'initialiser un objet ; 1. **type abstrait :** d'autre part, les types sont cruciaux dans les systèmes de vérification statique, au sens large, dont le but est de trouver un maximum de défauts à la seule lecture du code (par opposition aux techniques qui nécessitent de le faire tourner).

1.1.2 *Duck typing*

En python, ces deux aspects du typage sont relativement décorrélés.

Pour la deuxième dimension du typage, le système de types abstraits de python est connu sous le nom de *duck typing*, une appellation qui fait référence à cette phrase

```
When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird
```

1.1.3 L'exemple des itérables

Pour prendre l'exemple sans doute le plus représentatif, la notion d'*itérable* est un type abstrait, en ce sens que pour que le fragment :

```
for item in container:
    do_something(item)
```

ait un sens, il faut et il suffit que `container` soit un itérable.

On a vu qu'une chaîne est un itérable, de même qu'une liste, ou un tuple. Ces trois types conctrets, `str`, `list` et `tuple`, font donc tous partie du type abstrait *itérable*.

Dans un langage typé statiquement, pour pouvoir donner un type à cette construction, on serait **obligé** de définir un type - qu'on appellerait logiquement une classe abstraite - dont ces trois types seraient des descendants.

En python, et c'est le point que nous voulons souligner dans ce complément, il n'existe pas dans le système python d'objet de type `type` qui matérialise ce type `iterable`. Si on regarde les superclasses de nos trois types concrets, on voit que leur seul ancêtre commun est la classe `object` :

```
In []: str.__bases__

In []: str.__bases__[0].__bases__

In []: list.__bases__

In []: tuple.__bases__
```

Qu'on peut résumer comme ceci :

1.1.4 Un autre exemple

Pour prendre un exemple plus simple, pour que cette expression :

```
graphic.draw()
```

ait un sens, il faut que l'objet `graphic` ait une méthode `draw`.

À nouveau, dans un langage typé statiquement, on serait amené à définir une classe abstraite `Graphic`. En python ce n'est **pas requis** ; vous pouvez utiliser ce code tel quel avec deux classes `Rectangle` et `Texte` qui n'ont pas de rapport entre elles - autres que, à nouveau, d'avoir `object` comme ancêtre commun, pourvu qu'elles aient toutes les deux une méthode `draw`.

1.1.5 Héritage et type abstrait

Pour résumer, en python comme dans les langages typés statiquement, on a bien entendu la bonne propriété que si, par exemple, la classe `Spam` est itérable, alors la classe `Eggs` qui hérite de `Spam` est itérable.

Mais dans l'autre sens, si `Foo` et `Bar` sont itérables, il n'y a pas forcément une superclasse commune qui représente l'ensemble des objets itérables.

1.1.6 `isinstance` sur stéroïdes

D'un autre côté, c'est très utile d'exposer au programmeur un moyen de vérifier si un objet a un *type* donné - dans un sens volontairement vague ici.

On a déjà parlé - en Semaine 4, séquence "les fonctions" - de l'intérêt qu'il peut y avoir à tester le type d'un argument avec `isinstance` dans une fonction, pour parvenir à faire l'équivalent de la surcharge en C++ (la surcharge en C++ c'est quand vous définissez plusieurs fonctions qui ont le même nom mais des types d'arguments différents).

C'est pourquoi, quand on a cherché à exposer au programmeur des propriétés comme "cet objet est-il iterable ?", on a choisi d'étendre *isinstance* au travers de cette initiative. C'est ainsi qu'on peut faire par exemple :

```
In []: from collections import Iterable

In []: isinstance('ab', Iterable)

In []: isinstance([1, 2], Iterable)

In []: # comme on l'a vu un objet qui a une methode  __iter__()
       # et une next() est considere comme un iterable
       class Foo():
           def __iter__(self):
               return self
           def next(self):
               # ceci naturellement est bidon
               return

       foo = Foo()
       isinstance(foo, Iterable)
```

L'implémentation du module `abc` donne l'**illusion** que `Iterable` est un objet dans la hiérarchie de classes, et que tous ces *types* `str`, `list`, et `Foo` lui sont asujettis, mais ce n'est pas le cas en réalité ; comme on l'a vu plus tôt, ces trois types ne sont pas comparables dans la hiérarchie de classes, ils n'ont pas de plus petit (ou plus grand) élément.

Je signale pour finir, à propos de `isinstance` et du module `collections`, que la définition du symbole `Hashable` est à mon avis beaucoup moins convaincante que `Iterable` ; si vous vous souvenez qu'en Semaine 3, Séquence "les dictionnaires", on avait vu que les clés doivent être globalement immuables. C'est une caractéristique qui est assez difficile à écrire, et en tous cas ceci de mon point de vue ne remplit pas la fonction :

```
In []: from collections import Hashable

In []: # un tuple qui contient une liste ne convient
       # pas comme clé dans un dictionnaire
       # et pourtant
       isinstance (([1], [2]), Hashable)
```

1.1.7 python et les classes abstraites

Les points à retenir de ce complément un peu digressif sont : * en python, on hérite des **implémentations** et pas des **spécifications** ; * et le langage n'est pas taillé pour tirer profit de **classes abstraites** - même si rien ne vous interdit d'écrire, pour des raisons documentaires, une classe qui résume l'interface qui est attendue par tel ou tel système de plugin.

Venant de C++ ou de Java, cela peut prendre du temps d'arriver à se débarrasser de l'espèce de réflexe qui fait qu'on pense d'abord classe abstraite, puis implémentations.