isinstance

Complément - niveau basique

Typage dynamique

En première semaine, nous avons rapidement mentionné les concepts de typage statique et dynamique.

Avec la fonction prédéfinie `isinstance` – qui peut être par ailleurs utile dans d'autres contextes – vous pouvez facilement:

- vérifier qu'un argument d'une fonction a bien le type attendu,
- et traiter différemment les entrées selon leur type.

Voyons tout de suite sur un exemple simple comment on pourrait définir une fonction qui travaille sur un entier, mais qui par commmodité peut aussi accepter un entier passé comme une chaîne de caractères, ou même une liste d'entiers (auquel cas on renvoie la liste des factorielles) :

```
from types import StringTypes

def factoriel(argument):
    # si on reçoit un entier
    if isinstance(argument, int):
        return 1 if argument <= 1 else argument * factoriel(argument - 1)
    # convertir en entier si on reçoit une chaîne
    elif isinstance(argument, StringTypes):
        return factoriel(int(argument))
    # la liste des résultats si on reçoit un tuple ou une liste
    elif isinstance(argument, (tuple, list)):
        return [factoriel(i) for i in argument]
    # sinon on lève une exception
    else:
        raise TypeError, argument


print "entier", factoriel(4)
print "chaine", factoriel("8")
print "tuple", factoriel((4, 8))
```

Remarquez que la fonction `isinstance` **possède elle-même** une logique de ce genre, puisqu'en ligne 5 nous lui avons passé en deuxième argument un type (`int`), alors qu'en ligne 11 on lui a passé un tuple de deux types. Dans ce second cas naturellement, elle vérifie si l'objet

(le premier argument) est **de l'un des types** mentionnés dans le tuple.

Le module `types`

Dans cet exemple nous avons également utilisé le module `types`, et notamment le symbole `types.StringTypes`, plutôt que simplement `str`.

C'est une pratique recommandée pour le cas où on appellerait notre fonction avec en argument un objet de type `unicode`.

```
factoriel(u'4')
```

Complément - niveau intermédiaire

Le module `types` (suite)

Le module `types` définit un certain nombre de constantes de ce genre - vous trouverez une liste exhaustive à la fin de ce notebook. Il n'est pas très difficile de deviner que

```
StringTypes == (str, unicode)
```

Un autre symbole utile est `FunctionType`

```
from types import FunctionType
isinstance(factoriel, FunctionType)
```

Mais méfiez vous toutefois que les fonctions ***built-in*** sont de type `BuiltinFunctionType`

```
from types import BuiltinFunctionType
isinstance(len, BuiltinFunctionType)
```

`isinstance` ***vs*** `type`

Il est recommandé d'utiliser `isinstance` par rapport à la fonction `type`. Tout d'abord, cela permet, on vient de le voir, de prendre en compte plusieurs types.

Mais aussi et surtout `isinstance` supporte la notion d'héritage qui est centrale dans le cadre de la programmation orientée objet, sur laquelle nous allons anticiper un tout petit peu par rapport aux présentations de la semaine prochaine.

Avec la programmation objet, vous pouvez définir vos propres types. On peut par exemple définir une classe `Animal` qui convient pour tous les animaux, puis définir une sous-classe `Mammifere`. On dit que la classe `Mammifere` ***hérite*** de la classe `Animal`, et on l'appelle sous-classe parce qu'elle représente une partie des animaux; et donc tout ce qu'on peut faire sur les animaux peut être fait sur les mammifères.

En voici une implémentation très rudimentaire, uniquement pour illustrer le principe de

l'héritage. Si ce qui suit vous semble difficile à comprendre, pas d'inquiétude, nous reviendrons dessus la semaine prochaine lorsque nous parlerons des classes.

```
class Animal:
    def __init__(self, name):
        self.name = name

class Mammifere(Animal):
    def __init__(self, name):
        Animal.__init__(self, name)
```

Encore une fois tout ceci sera vu en détail la semaine prochaine. Ce qui nous intéresse dans l'immédiat c'est que `isinstance` permet dans ce contexte de faire des choses qu'on ne peut pas faire directement avec la fonction `type`, comme ceci

```
# c'est comme ceci qu'on peut créer un objet de type `Animal` (méthode __init__)
requin = Animal('requin')
# idem pour un Mammifere
baleine = Mammifere('baleine')

# bien sûr ici la réponse est 'True'
print "l'objet baleine est-il un mammifere", isinstance(baleine, Mammifere)
# ici c'est moins évident, mais la réponse est 'True' aussi
print "l'objet baleine est-il un animal", isinstance(baleine, Animal)
```

Vous voyez qu'ici, bien que l'objet baleine est de type `Mammifere`, on peut le considérer comme étant **aussi** de type `Animal`.

Ceci est motivé de la façon suivante. Comme on l'a dit plus haut, tout ce qu'on peut faire (en termes notamment d'envoi de méthodes) sur un objet de type `Animal`, on peut le faire sur un objet de type `Mammifere`. Dit en termes ensemblistes, l'ensemble des mammifères est inclus dans l'ensemble des animaux.

Annexe - Les symboles du module `types`

```
import types
dir(types)
```