W5-S6-C2-Complement-surcharge-operateurs-2

December 15, 2014

1 Surcharge opérateurs (2)

1.1 Complément - niveau avancé

Nous poursuivons dans ce complément la sélection de méthodes spéciales entreprise en première partie.

1.1.1 `__contains__`, `__len__`, `__getitem__` et apparentés

La méthode `__contains__` permet de donner un sens à :

```
item in objet
```

Sans grande surprise, elle prend en argument un objet et un item, et doit renvoyer un booléen. Nous l'illustrons ci-dessous avec la classe `DualQueue`.

La méthode `__len__` est utilisée par la fonction *builtin* `len` pour retourner la longueur d'un objet.

La classe `DualQueue` Nous allons illustrer ceci avec un exemple de classe, un peu artificiel, qui implémente une queue de type FIFO. Les objets sont d'abord admis dans la file d'entrée (`add_input`), puis déplacés dans la file de sortie (`move_input_to_output`), et enfin sortis (`emit_output`).

Clairement, cet exemple est à but uniquement pédagogique; on veut montrer comment une implémentation qui repose sur deux listes séparées peut donner l'illusion d'une continuité, et se présenter comme un container unique. De plus cette implémentation ne fait aucun contrôle pour ne pas obscurcir le code.

```
In []: class DualQueue (object):
           """Une double file d'attente FIFO"""

           def __init__(self):
               "constructeur, sans argument"
               self.inputs = []
               self.outputs = []

           def __repr__ (self):
               "affichage"
               return "<DualQueue, inputs={inputs}, outputs={outputs}>".format(**vars(self))

           # la partie qui nous intéresse ici
           def __contains__(self, item):
               "appartenance d'un objet à la queue"
               return item in self.inputs or item in self.outputs
           def __len__(self):
               "longueur de la queue"
```

```
            return len(self.inputs) + len(self.outputs)

        # l'interface publique de la classe
        # le plus simple possible et sans aucun contrôle
        def add_input(self, item):
            "faire entrer un objet dans la queue d'entrée"
            self.inputs.insert(0,item)
        def move_input_to_output (self):
            "l'objet le plus ancien de la queue d'entrée est promu dans la queue de sortie"
            self.outputs.insert(0,self.inputs.pop())
        def emit_output (self):
            "l'objet le plus ancien de la queue de sortie est émis"
            return self.outputs.pop()
```

```
In []: # on construit une instance pour nos essais
       queue = DualQueue ()
       queue.add_input ('zero')
       queue.add_input ('un')
       queue.move_input_to_output()
       queue.move_input_to_output()
       queue.add_input ('deux')
       queue.add_input ('trois')

       print queue
```

Longueur et appartenance Avec cette première version de la classe `DualQueue` on peut utiliser `len` et le test d'appartenance :

```
In []: print 'len()',len(queue)

       print "deux appartient-il ?", 'deux' in queue
       print "(1,) appartient-il ?", (1,) in queue
```

Accès séquentiel (accès par un index entier) Lorsqu'on a la notion de longueur de l'objet avec `__len__`, il peut être opportun - quoique cela n'est pas imposé par le langage, comme on vient de le voir - de proposer également un accès indexé par un entier pour pouvoir faire :

```
queue[1]
```

Pour ne pas répéter tout le code de la classe, nous allons étendre `DualQueue`; pour cela nous définissons une fonction, que nous affectons ensuite à `DualQueue.__getitem__`, comme nous avons déjà eu l'occasion de le faire :

```
In []: # une première version de DualQueue.__getitem__
       # pour uniquement l'accès par index

       # on définit une fonction
       def dual_queue_getitem (self, index):
           "redéfinit l'accès [] séquentiel"

           # on vérifie que l'index a un sens
           if index < 0 or index >= len(self):
               raise IndexError("Mauvais indice {} pour DualQueue".format(index))
           # on décide que l'index 0 correspond à l'élément le plus ancien
           # ce qui oblige à une petite gymnastique
```

```
            li = len(self.inputs)
            lo = len(self.outputs)
            if index < lo:
                return self.outputs[lo-index-1]
            else:
                return self.inputs[li-(index-lo)-1]

        # et on affecte cette fonction à l'intérieur de la classe
        DualQueue.__getitem__ = dual_queue_getitem
```

À présent, on peut **accéder** aux objets de la queue **séquentiellement** :

```
In []: print queue[0]
```

ce qui lève la même exception qu'avec une vraie liste si on utilise un mauvais index :

```
In []: try:
            print queue[5]
        except IndexError as e:
            print 'ERREUR',e
```

Amélioration : accès par slice Si on veut aussi supporter l'accès par slice comme ceci :

```
queue[1:3]
```

il nous faut modifier la méthode `__getitem__`.

Le second argument de `__getitem__` correspond naturellement au contenu des crochets `[]`, on utilise donc `isinstance` pour écrire un code qui s'adapte au type d'indexation, comme ceci :

```
In []: # une deuxième version de DualQueue.__getitem__
       # pour l'accès par index ou par slice

       def dual_queue_getitem (self, key):
           "redéfinit l'accès par [] pour entiers, slices, et autres"

           # l'accès par slice queue[1:3]
           # nous donne pour key un objet de type slice
           if isinstance(key, slice):
               # key.indices donne les indices qui vont bien
               return [self[index] for index in xrange(*key.indices(len(self)))]

           # queue[3] nous donne pour key un entier
           elif isinstance(key, int):
               index = key
               # on vérifie que l'index a un sens
               if index < 0 or index >= len(self):
                   raise IndexError("Mauvais indice {} pour DualQueue".format(index))
               # on décide que l'index 0 correspond à l'élément le plus ancien
               # ce qui oblige à une petite gymnastique
               li = len(self.inputs)
               lo = len(self.outputs)
               if index < lo:
                   return self.outputs[lo-index-1]
               else:
                   return self.inputs[li-(index-lo)-1]
```

```
            # queue ['foo'] n'a pas de sens pour nous
            else:
                raise KeyError("[] avec type non reconnu {}".format(type(key)))

        # et on affecte cette fonction à l'intérieur de la classe
        DualQueue.__getitem__ = dual_queue_getitem
```

Maintenant on peut accéder par slice

```
In []: queue[1:3]
```

Et on reçoit bien une exception si on essaie d'accéder par clé :

```
In []: try:
            queue['key']
        except KeyError as e:
            print "On ne peut pas accéder par clé",e
```

L'objet est itérable (même sans avoir `__iter__`) Avec seulement `__getitem__`, on peut **faire une boucle** sur l'objet queue. On l'a mentionné rapidement dans la séquence sur les itérateurs, mais la **méthode `__iter__` n'est pas la seule façon** de rendre un objet itérable :

```
In []: for item in queue:
           print item
```

On peut faire un test sur l'objet De manière similaire, même sans la méthode `__nonzero__`, cette classe sait **faire des tests de manière correcte** grâce uniquement à la méthode `__len__` :

```
In []: # un test fait directement sur la queue
        if queue:
            print "La queue {} est considérée comme True".format(queue)

        # on vide la queue pour tester sur une queue vide
        for i in range(2):
            queue.move_input_to_output()
        for i in range(4):
            queue.emit_output()

        # maintenant le test est négatif (notez bien le *not* ici)
        if not queue:
            print "La queue {} est considérée comme False".format(queue)
```

1.1.2 `__call__` et les *callables*

Le langage introduit de manière similaire la notion de ***callable*** - littéralement, qui peut être appelé. L'idée est très simple, on cherche à donner un sens à un fragment de code du genre de :

```
# on crée une instance
objet = Classe(arguments)

# et c'est l'objet (et pas la classe) qu'on utilise comme une fonction
objet(arg1, arg2)
```

Le protocole ici est très simple; cette dernière ligne a un sens en python dès lors que : * `objet` possède une méthode `__call__`, * et que celle-ci peut être envoyée à `objet` avec les arguments `arg1, arg2`, pour nous donner le résultat qui sera retourné par `objet(arg1, arg2)`.

```
objet(arg1, arg2)   ===>   objet.__call__(arg1, arg2)
```

Voyons cela sur un exemple :

```
In []: class PlusClosure (object):
           """Une classe callable qui permet de faire un peu comme la
           fonction built-in sum mais avec en ajoutant une valeur initiale"""
           def __init__(self, initial):
               self.initial = initial
           def __call__(self, *args):
               return self.initial + sum(args)

       # on crée une instance avec une valeur initiale 2 pour la somme
       plus2 = PlusClosure (2)

In []: # on peut maintenant utiliser cet objet
       # comme une fonction qui fait sum(*arg)+2

       print '[] ->', plus2()

       print '1 ->', plus2(1)

       print '1,2 ->', plus2(1, 2)
```

Pour ceux qui connaissent, nous avons choisi à dessein un exemple qui s'apparente à une clôture. Nous reviendrons sur cette notion de *callable* lorsque nous verrons les décorateurs en semaine 7.

1.1.3 `__getattr__` et apparentés

Dans cette dernière partie nous allons voir comment avec la méthode `__getattr__`, on peut redéfinir la façon que le langage a d'évaluer

```
objet.attribut
```

Avertissement: on a vu dans la séquence consacrée à l'héritage que, pour l'essentiel, le mécanisme d'héritage repose **précisément** sur la façon d'évaluer les attributs d'un objet, aussi nous vous recommandons d'utiliser ce trait avec précaution, car il vous donne la possibilité de "faire muter le langage" comme on dit.

Un exemple : la classe `RPCProxy` Pour illustrer `__getattr__`, nous allons considérer le problème suivant. Une application utilise un service distant, avec laquelle elle interagit au travers d'une API.

C'est une situation très fréquente: lorsqu'on utilise un service météo, ou de géolocalisation, ou de réservation, le prestataire vous propose une **API** (Application Programming Interface) qui se présente bien souvent comme une **liste de fonctions**, que votre fonction peut appeler à distance au travers d'un mécanisme de **RPC** (Remote Procedure Call).

Imaginez pour fixer les idées que vous utilisez un service de réservation de ressources dans un Cloud, qui vous permet d'appeler les fonctions suivantes : * `GetNodes`(...) pour obtenir des informations sur les noeuds disponibles, * `BookNode`(...) pour réserver un noeud, * `ReleaseNode`(...) pour abandonner un noeud.

Naturellement ceci est une API extrêmement simplifiée. Le point que nous voulons illustrer ici est que le dialogue avec le service distant : * requiert ses propres données - comme l'URL où on peut joindre le service, et les identifiants à utiliser pour s'authentifier, * et possède sa propre logique - dans le cas d'une

authentification par session par exemple, il faut s'authentifier une première fois avec un login/password, pour obtenir une session qu'on peut utiliser dans les appels suivants.

Pour ces raisons il est naturel de concevoir une classe `RPCProxy` dans laquelle on va rassembler à la fois ces données et cette logique, pour soulager toute l'application de ces détails, comme on l'a illustré ci-dessous :

Pour implémenter la plomberie liée à RPC, à l'encodage et décodage des données, et qui sera interne à la classe `RPCProxy`, on pourra en vraie grandeur utiliser des outils comme : * `xmlrpclib` qui fait partie de la librairie standard, * ou pour JSON, une des nombreuses implémentations qu'un moteur de recherche vous exposera si vous cherchez `python rpc json`, comme par exemple `json-rpc`

Cela n'est toutefois pas notre sujet ici, et nous nous contenterons, dans notre code simplifié, d'imprimer un message.

Une approche naïve Se pose donc la question de savoir quelle interface la classe `RPCProxy` doit offrir au reste du monde. Dans une première version naïve on pourrait écrire quelque chose comme :

```
In []: # la version naïve de la classe RPCProxy

       class RPCProxy(object):

           def __init__(self, url, login, password):
               self.url = url
               self.login = login
               self.password = password

           def _forward_call(self, function, *args):
               """
               helper method that marshalls and forwards
               the function and arguments to the remote end
               """
               print "Envoi à {}\nde la fonction {} -- args= {}".\
                   format(self.url, function, args)
               return "retour de la fonction " + function

           def GetNodes (self, *args):
               return self._forward_call ('GetNodes', *args)
           def BookNode (self, *args):
               return self._forward_call ('BookNode', *args)
           def ReleaseNode (self, *args):
               return self._forward_call ('ReleaseNode', *args)
```

Ainsi l'application utilise la classe de cette façon :

```
In []: # création d'une instance de RPCProxy

       rpc_proxy = RPCProxy (url='http://cloud.provider.com/JSONAPI',
                             login='dupont',
                             password='***')

       # cette partie du code, en tant qu'utilisateur de l'API,
       # est supposée connaître les détails
       # des arguments à passer
       # et de comment utiliser les valeurs de retour
       nodes_list = rpc_proxy.GetNodes (
           [ ('phy_mem', '>=', '32G') ] )
```

```
# reserver un noeud
node_lease = rpc_proxy.BookNode (
    { 'id' : 1002, 'phy_mem' : '32G' } )
```

Discussion Quelques commentaires en vrac au sujet de cette approche :

- l'interface est correcte; l'objet `rcp_proxy` se comporte bien comme un proxy, on a donné au programmeur l'illusion complète qu'il utilise une classe locale (sauf pour les performances bien entendu...);
- la séparation des rôles est raisonnable également, la classe RPCProxy n'a pas à connaître le détail de la signature de chaque méthode, charge à l'appelant d'utiliser l'API correctement;
- par contre ce qui cloche, c'est que l'implémentation de la classe RPCProxy dépend de la liste des fonctions exposées par l'API; imaginez une API avec 100 ou 200 méthodes, cela donne une dépendance assez forte et surtout inutile;
- enfin, nous avons escamoté la nécessité de faire de RPCProxy un singleton, mais c'est une toute autre histoire.

Une approche plus subtile Pour obtenir une implémentation qui conserve toutes les qualités de la version naïve, mais sans la nécessité de définir une à une toutes les fonctions de l'API, on peut tirer profit de `__getattr__`, comme dans cette deuxième version :

```
In []: # une deuxième implémentation de RPCProxy

       class RPCProxy(object):

           def __init__(self, url, login, password):
               self.url = url
               self.login = login
               self.password = password

           def __getattr__ (self, function):
               """
               Crée à la volée une méthode sur RPCProxy qui correspond
               à la fonction distante 'function'
               """
               def forwarder (*args):
                   print "Envoi à {}\nde la fonction {} -- args= {}".\
                       format(self.url, function, args)
                   return "retour de la fonction " + function
               return forwarder
```

Qui est cette fois **totalement découplée** des détails de l'API, et qu'on peut utiliser exactement comme tout à l'heure :

```
In []: # création d'une instance de RPCProxy

       rpc_proxy = RPCProxy (url='http://cloud.provider.com/JSONAPI',
                             login='dupont',
                             password='***')

       # cette partie du code, en tant qu'utilisateur de l'API,
       # est supposée connaître les détails
       # des arguments à passer
       # et de comment utiliser les valeurs de retour
```

```
nodes_list = rpc_proxy.GetNodes (
    [ ('phy_mem', '>=', '32G') ] )

# reserver un noeud
node_lease = rpc_proxy.BookNode (
    { 'id' : 1002, 'phy_mem' : '32G' } )
```

Exercice Les étudiants courageux et/ou inspirés peuvent s'amuser à reprendre cette dernière version de `RPCProxy`, mais en utilisant une classe de **callables** comme une *factory* pour générer les attributs.

```
In []: # une troisième implémentation de RPCProxy

class Callable:
    "votre code"

class RPCProxy(object):

    def __init__(self, url, login, password):
        self.url = url
        self.login = login
        self.password = password

    def __getattr__ (self, function):
        "votre code"
```

Nous ne proposons pas de correction en ligne mais vous pouvez simplement évaluer le même code :

```
In []: # création d'une instance de RPCProxy

rpc_proxy = RPCProxy (url='http://cloud.provider.com/JSONAPI',
                      login='dupont',
                      password='***')

# cette partie du code, en tant qu'utilisateur de l'API,
# est supposée connaître les détails
# des arguments à passer
# et de comment utiliser les valeurs de retour
nodes_list = rpc_proxy.GetNodes (
    [ ('phy_mem', '>=', '32G') ] )

# reserver un noeud
node_lease = rpc_proxy.BookNode (
    { 'id' : 1002, 'phy_mem' : '32G' } )
```