Programmation système orientée objet

Auteur: Olivier Laurent
Contact: oli@aragne.com
Organisation: Python Blanc Bleu Belge
Date: 10/02/2005

Table des matières

1 Introduction

Il est supposé, ici, que vous avez des notions de programmation orientée objet. Si ce n'est pas le cas, il est conseillé de se renseigner sur le sujet avant de poursuivre. En gros, la grosse différence entre la programmation structurée et la programmation objet, c'est que l'on crée des objets qui vont faire le travail pour lequel ils sont programmés. Ces objets possèdent des **attributs** (en gros, les variables) et des **méthodes** (les fonctions). Il ne vous reste plus qu'à invoquer les méthodes de ces objets dans votre programme et le travail sera effectué.

L'avantage, c'est, entre autre, la réutilisabilité: il est beaucoup plus facile de réutiliser et de combiner des objets entre eux dans les programmes que vous créez.

Pour rappel, nous essayons, ici, de créer un programme capable de lister les fichiers d'un répertoire donné de manière récursive (donc y compris les fichiers des sous-répertoires). Cette liste sera triée sur la taille des fichiers (les plus gros en premier).

2 Créer l'objet fichier

En Python, un objet se déclare par le mot-clé **class** suivit du nom de l'objet et de deux-points.

```
#!/usr/bin/python

import os, sys

class FileObj:
    pass

def main():
    mon_objet = FileObj()
    print mon_objet

if __name__ == "__main__":
    main()
```

Vous créez une instance de l'objet en attribuant le nom de la classe (suivit de parenthèses) à une variable.

Sauvegardons notre programme dans un fichier appelé fileobj.py et lançons notre programme :

```
% python fileobj.py
<__main__.FileObj instance at 0x4022e3cc>
%
```

Python nous informe qu'un objet a été créé à tel endroit de la mémoire. Cet objet, jusqu'à présent ne fait strictement rien. Étoffons le un peu. L'idéal, lorsque l'on crée une classe est de définir une **méthode constructeur** (c'est-à-dire une méthode qui va construire l'objet suivant une certaine logique). En Python, cette méthode s'appelle **__init__** (c'est-à-dire init entre deux doubles underscores : deux devant et deux derrière).

> **Note**
>
> Vous rencontrerez beaucoup de méthodes commençant et finissant par deux underscores en Python : ce sont des méthodes spéciales. Nous en avons déjà rencontrée auparavant (__main__ et __name__).

Cet objet aura besoin du chemin vers le répertoire que l'on veut lister.

Voyons comment faire. J'expliquerai juste après.

```
#!/usr/bin/python

import os, sys

class FileObj:
    def __init__(self, filepath):
        self.filepath = os.path.abspath(os.path.normpath(filepath))

def main():
    mon_objet = FileObj('/tmp')
    print mon_objet.filepath

if __name__ == "__main__":
    main()
```

La méthode __init__ a besoin de deux arguments : self et le chemin du répertoire. Le premier argument de toute méthode Python est toujours **self** (plus ou moins équivalent au **this** du C++). self est l'objet en lui même. Donc, à la ligne suivante, **self.filepath** se réfère à l'attribut filePath de l'objet fileobj.

```
% python fileobj.py
/tmp
```

Vérifions si ce répertoire existe vraiment et rajoutons un attribut à cet objet: le nom du répertoire ou du fichier.

```
#!/usr/bin/python

import os, sys

class FileObj:
    def __init__(self, filepath):
        self.filepath = self._get_path(filepath)
        self.filename = os.path.basename(self.filepath)

    def _get_path(self, filepath):
        fullpath = os.path.abspath(os.path.normpath(filepath))
        if os.path.exists(fullpath):
            return fullpath
        else:
            raise EnvironmentError('This file does not exist')

def main():
    mon_objet = FileObj('/tmp')
    print mon_objet.filepath
    print mon_objet.filename

if __name__ == "__main__":
    main()
```

La méthode **os.path.exists** vérifie que le fichier existe bel et bien. Dans le cas contraire, nous élevons une exceptions agrémentée d'un message d'erreur.

Cet objet peut aussi bien représenter un fichier qu'un répertoire (puisque c'est, à peu de choses près, la même chose). Mais nous aimerions quand même savoir si le chemin représente un fichier ou un répertoire. Ces deux méthodes effectuerons cette tâche:

```
#!/usr/bin/python
# -*- coding: latin1 -*-

import os, sys

class FileObj:
    def __init__(self, filepath):
        self.filepath = self._get_path(filepath)
        self.filename = os.path.basename(self.filepath)

    def _get_path(self, filepath):
        fullpath = os.path.abspath(os.path.normpath(filepath))
        if os.path.exists(fullpath):
            return fullpath
        else:
            raise EnvironmentError('This file does not exist')

    def isdir(self):
        """est-ce un répertoire ?"""
        if os.path.isdir(self.filepath):
            return True
        else:
            return False

    def isfile(self):
        """est-ce un fichier ?"""
        if self.isdir():
            return False
        else:
            return True

def main():
    mon_objet = FileObj('/tmp')
    print "est un répertoire ?", mon_objet.isdir()
    print "est un fichier ?", mon_objet.isfile()

if __name__ == "__main__":
    main()
```

isdir et **isfile** sont des méthodes de l'objet FileObj. Ces méthodes, lorsqu'elles sont appelées dans un programme, ne prennent pas d'arguments (alors qu'elles sont définies par isDir(self) et isFile(self)). Mais nous avons vu que **self** désignait l'objet en lui même et permet donc de définir cette méthode comme appartenant à l'objet et à lui seul.

Vous avez peut-être remarqué que j'appelais la méthode isdir dans la méthode isfile. D'un point de vue logique, si le chemin n'est pas un répertoire, c'est un fichier. D'un point de vue **programmation**

objet, c'est comme cela que l'on fait en Python pour appeler une méthode à l'intérieur de la définition de l'objet: **self.method**

3 Lister le contenu d'un répertoire

Nous allons ajouter deux méthodes à notre objet Fichier. La première **get_dirs** nous retournera la liste des répertoires contenus dans notre objet. La seconde **get_files** listera les fichiers.

```
#!/usr/bin/python
# -*- coding: latin1 -*-

import os, sys

class FileObj:
    def __init__(self, filepath):
        self.filepath = self._get_path(filepath)
        self.filename = os.path.basename(self.filepath)

    def _get_path(self, filepath):
        fullpath = os.path.abspath(os.path.normpath(filepath))
        if os.path.exists(fullpath):
            return fullpath
        else:
            raise EnvironmentError("The '%s' file does not exist" % filepath)

    def isdir(self):
        """est-ce un répertoire ?"""
        return os.path.isdir(self.filepath)

    def isfile(self):
        """est-ce un fichier ?"""
        return not self.isdir()

    def get_dirs(self):
        """Retourne la liste des répertoires"""
        if self.isdir():
            # liste complète (fichiers + répertoires):
            full_file_list = os.listdir(self.filepath)
            # liste filtrée (fichiers seuls):
            return [file for file in full_file_list if
                      FileObj(os.path.join(self.filepath, file)).isdir()]
        else:
            return []

    def get_files(self):
        """Retourne la liste des fichiers"""
        if self.isdir():
            # liste complète (fichiers + répertoires):
            full_file_list = os.listdir(self.filepath)
            # liste filtrée (fichiers seuls):
            return [file for file in full_file_list if
                      not FileObj(os.path.join(self.filepath, file)).isdir()]
```

```
        else:
            return []

def main():
    mon_objet = FileObj('/tmp')
    print mon_objet.get_dirs()
    print mon_objet.get_files()

if __name__ == "__main__":
    main()
```

C'est, on le voit, assez simple. Une petite remarque : seul un répertoire contient des fichiers : d'où le test **if self.isdir()**.

La méthode **os.listdir** nous retourne la liste (non récursivement) des fichiers de notre objet fichier. Cette liste n'est pas triée. Elle contient aussi bien des fichiers, que des répertoires. Or, nous avons uniquement besoin des répertoires. Nous trions alors la liste complète et retournant la liste triée (contenant uniquement les répertoires).

Pour retourner la liste de répertoire, nous avons utilisé une fonctionnalité très pratique de Python: la création fonctionnelle de liste (ou *list comprehension* en anglais).

On pourrais traduire cette ligne par:

```
Pour chaque fichier contenu dans la liste complète, si je crée un objet
FileObj avec ce fichier et que c'est un répertoire, ajoute le à la liste.
```

En *Pythonique Ancien*, on aurait dit:

```
dir_list = []
for file in full_file_list:
fullpath = os.path.join(self.filepath, file)
fileobj = FileObj(fullpath)
if fileobj.isdir():
  dir_list.append(file)
return dir_list
```

Note

Comme vous l'avez peut-être constaté, nous avons également modifié les méthodes **isdir** et **isfile**. Retourner **True** ou **False** est superflux puisque la méthode **os.isdir** retourne elle même **True** ou **False**.

Plus subtil maintenant : la liste des répertoires de manière récursive. Pour cela, choisissez un répertoire contenant de nombreux répertoires.

Nous allons utiliser la fonction **walk** du module **os.path**. Cette fonction possède la syntaxe suivante : **os.path.walk(path, visitfunc, arg)**. Où path est le chemin racine (ou chemin de départ). **walk** appelle la fonction **visitfunc** pour chaque répertoire trouvé. Arg est un argument optionnel (que nous fixerons à None car nous n'en avons pas besoin).

La fonction **visitfunc** possède la syntaxe suivante : **visit(arg, dirname, names)** où **arg** est un argument optionnel, **dirname** est le répertoire actuellement visité et **names**, la liste des fichiers du répertoire actuellement visité.

```
#!/usr/bin/python
# -*- coding: latin1 -*-

import os, sys

class FileObj:

    [...]

    def get_recursive_dirs(self):
        """Retourne la liste complète des répertoires."""
        recursive_dir_list = []

        def visit(arg, dirname, files):
            recursive_dir_list.append(dirname)

        if self.isdir():
            os.path.walk(self.filepath, visit, None)
            return recursive_dir_list
        else:
            return []

    def get_recursive_files(self):
        """Retourne la liste complète des fichiers (hormis les répertoires)"""
        file_list = []
        if self.isdir():
            for dir in self.get_recursive_dirs():
                files = os.listdir(dir)
                for file in files:
                    if os.path.isfile(os.path.join(dir, file)):
                        file_list.append(os.path.join(dir, file))
            return file_list
        else:
            return []

def main():
    from pprint import pprint as pp
    mon_objet = FileObj('/tmp')
    pp(mon_objet.get_recursive_files())

if __name__ == "__main__":
    main()
```

Je défini la méthode **visit** dans la méthode **get_recursive_dir**. Cela me permet d'utiliser la liste **recursive_dir_list** dans cette méthode. Cela évite de créer une variable globale ou une nouvelle variable d'instance.

> **Note**
>
> Nous avons utilisé le module **pprint** pour afficher la liste de manière plus jolie. **pprint** veut, en fait, dire **Pretty Print**, c'est-à-dire **Joli affichage**

Voici le résultat que l'on peut obtenir:

```
% python fileobj.py
\home\billou\python\graph
\home\billou\python\html
\home\billou\python\html\data
\home\billou\python\html\data\css
\home\billou\python\html\data\frames
\home\billou\python\html\data\images
\home\billou\python\html\data\main
\home\billou\python\html\testsite
\home\billou\python\html\tmp
\home\billou\python\pyaf-0.1.2
\home\billou\python\pyaf-0.1.2\doc
\home\billou\python\pyaf-0.1.2\examples
\home\billou\python\pyaf-0.1.2\pyaf
%
```

L'étape suivante consistera à lister les fichiers contenu dans tous ces répertoires. C'est la méthode **get_recursive_files** qui s'en chargera. Cela reste relativement simple : Chaque répertoire de la liste retournée par **get_recursive_dirs** est visité par la méthode **os.listdir** qui retourne une liste des fichiers contenu par ce répertoire. Chaque fichier (et répertoire) de cette nouvelle liste est alors testé (les véritables fichiers sont ajoutés à la liste totale). Les répertoires sont écartés de cette liste totale.

4 Trier la liste des fichiers

4.1 Obtenir la taille des fichiers

Avant tout, pour que notre programme ait plus de souplesse, nous allons faire en sorte qu'il puisse accepter un paramètre (le chemin vers le fichier) en ligne de commande. Cela est très simple. Nous modifions la méthode **main** de notre module qui devient:

```
def main():
    import sys
    try:
        root = sys.argv[1]
    except IndexError:
        root = '/tmp'
    mon_objet = FileObj(root)
    print mon_objet.get_recursive_files()

if __name__ == "__main__":
    main()
```

Pour ce qui est de la taille du fichier, nous allons nous servir de la fonction **stat** du module **os**. **stat** nous renvoie, en fait, un tuple contenant diverses informations sur le fichier en question. Cela va du dernier temps d'accès au fichier, son numéro d'inode (sur Unix) ou sa taille. Windows étant un système plus rudimentaire, plusieurs de ces valeurs son vides (ex : pas de numéro d'inode).

La valeur qui nous intéresse dans ce tuple est la valeur numéro 7. Voici ce que cela donne :

```
#!/usr/bin/python
# -*- coding: latin1 -*-
```

```
import os, sys
import time

class FileObj:

    [...]

    def getstat(self):
        stats = os.stat(self.filepath)
        return stats

    def getsize(self):
        a = self.getstat()
        return a[6] # élément 7 du tuple

    def get_last_access_time(self):
        a = self.getstat()
        return time.ctime(a[7])

    def get_last_modification_time(self):
        a = self.getstat()
        return time.ctime(a[8])

    def get_last_status(self):
        a = self.getstat()
        return time.ctime(a[9])
```

Très simple (comme toujours en Python). La méthode **getstat** nous retourne le tuple de statistiques. cette méthode va nous servir à accéder à d'autres éléments d'information, notamment les derniers temps d'axes ou de modification du fichier. Pour cela, nous aurons besoin d'importer le module **time** car les temps retourné par stat sont bruts: un gros nombre qui ne veut rien dire pour le commun des mortels. En fait, c'est le nombre de seconde depuis le commencement des temps. Euh, le commencement des temps pour l'OS : c'est-à-dire le 1er janvier 1970 sur Unix ou le 1er janvier 1900 pour Windows. La fonction **ctime** du module **time** permet de convertir ce nombre en une forme plus compréhensible pour un être humain (anglophone mais bon).

4.2 Les dates en français

Le module **locale** permet de remédier au problème des dates en anglais.

```
#!/usr/bin/python
# -*- coding: latin1 -*-

import os, sys
import time

import locale

my_locale = "fr_FR@euro"
locale.setlocale(locale.LC_ALL, my_locale)

[...]
```

```
    def _format_date(self, seconds):
        return time.strftime('%d %B %Y', time.localtime(seconds))

    def get_last_access_time(self):
        stats = self.getstat()
        return self._format_date(stats[7])

    def get_last_modification_time(self):
        stats = self.getstat()
        return self._format_date(stats[8])

    def get_last_status(self):
        stats = self.getstat()
        return self._format_date(stats[9])
```

Après avoir importé le module **locale**, nous définissons la locale à utiliser. Ici 'fr_FR@euro'.

Nous définissons alors une méthode **_format_date**. La méthode **time.strftime** permet de formater une date. Elle prend, comme premier argument la chaîne de formatage et comme second argument un tuple représentant la date.

Nous pouvons obtenir ce tuple grâce à la méthode **time.localtime** qui prend comme argument le nombre retourné par la méthode **os.stat**.

4.3 Trier par ordre de taille

Après cette petite digression, revenons à notre problème principal : trier la liste des fichiers par ordre de taille. Nous allons, pour commencer, créer une liste contenant la totalité des fichiers du répertoire pointé. La méthode getRecursiveFiles fait déjà ce travail mais nous aimerions lier le nom du fichier à sa taille dans un tuple. Nous allons donc nous servir de la liste retournée par cette méthode. Pour chaque fichier de la liste, nous allons créé un nouvel objet FileObj. Le tuple sera formé de la taille du fichier et de sa taille. En attendant quelques remarques, voyons ce que cela donne :

```
#!/usr/bin/python
# -*- coding: latin1 -*-

[...]

class FileObj:
    """Classe représentant un fichier"""

    [...]

    def get_recursive_files(self):
        """Retourne la liste complète des fichiers (hormis les répertoires)"""
        file_list = []
        if self.isdir():
            for dir in self.get_recursive_dirs():
                files = os.listdir(dir)
                for file in files:
```

```
                    if os.path.isfile(os.path.join(dir, file)):
                        file_list.append(os.path.join(dir, file))
            return file_list
        else:
            return []

    def get_files_sorted_by_size(self):
        if self.isdir():
            bigger_file_list = []
            for file in self.get_recursive_files():
                obj = FileObj(file)
                bigger_file_list.append( (obj.getsize(), obj.filepath) )
            return bigger_file_list
        else:
            return []

    [...]

def main():
    mon_objet = FileObj('/tmp')
    for file in mon_objet.get_files_sorted_by_size():
        print file

if __name__ == "__main__":
    main()
```

En résumé, pour chaque fichier rencontré, un nouvel objet est créé et on construit un tuple formé de sa taille et de son chemin. A chaque itération, ce tuple est ajouté à une liste. On retourne alors cette liste qui est, mais pas avant de l'avoir triée avec la méthode **sort**.

Voici le programme complet:

```
#!/usr/bin/python
# -*- coding: latin1 -*-

import os, sys
import time

import locale

my_locale = "fr_FR@euro"
locale.setlocale(locale.LC_ALL, my_locale)


class FileObj:
    """Classe représentant un fichier"""

    def __init__(self, filepath):
        self.filepath = self._get_path(filepath)
        self.filename = os.path.basename(self.filepath)

    def _get_path(self, filepath):
```

```
        fullpath = os.path.abspath(os.path.normpath(filepath))
        if os.path.exists(fullpath):
            return fullpath
        else:
            raise EnvironmentError("The '%s' file does not exist" % filepath)

    def isdir(self):
        """est-ce un répertoire ?"""
        return os.path.isdir(self.filepath)

    def isfile(self):
        """est-ce un fichier ?"""
        return not self.isdir()

    def get_dirs(self):
        """Retourne la liste des répertoires"""
        if self.isdir():
            # liste complète (fichiers + répertoires):
            full_file_list = os.listdir(self.filepath)
            # liste filtrée (fichiers seuls):
            return [file for file in full_file_list if
                      FileObj(os.path.join(self.filepath, file)).isdir()]
        else:
            return []

    def get_files(self):
        """Retourne la liste des fichiers"""
        if self.isdir():
            # liste complète (fichiers + répertoires):
            full_file_list = os.listdir(self.filepath)
            # liste filtrée (fichiers seuls):
            return [file for file in full_file_list if
                      not FileObj(os.path.join(self.filepath, file)).isdir()]
        else:
            return []

    def get_recursive_dirs(self):
        """Retourne la liste complète des répertoires."""
        recursive_dir_list = []

        def visit(arg, dirname, files):
            recursive_dir_list.append(dirname)

        if self.isdir():
            os.path.walk(self.filepath, visit, None)
            return recursive_dir_list
        else:
            return []

    def get_recursive_files(self):
        """Retourne la liste complète des fichiers (hormis les répertoires)"""
        file_list = []
        if self.isdir():
```

```
            for dir in self.get_recursive_dirs():
                files = os.listdir(dir)
                for file in files:
                    if os.path.isfile(os.path.join(dir, file)):
                        file_list.append(os.path.join(dir, file))
            return file_list
        else:
            return []

    def get_files_sorted_by_size(self):
        if self.isdir():
            bigger_file_list = []
            for file in self.get_recursive_files():
                obj = FileObj(file)
                bigger_file_list.append( (obj.getsize(), obj.filepath) )
            bigger_file_list.sort()
            return bigger_file_list
        else:
            return []

    def getstat(self):
        stats = os.stat(self.filepath)
        return stats

    def getsize(self):
        a = self.getstat()
        return a[6] # élément 7 du tuple

    def _format_date(self, seconds):
        return time.strftime('%d %B %Y', time.localtime(seconds))

    def get_last_access_time(self):
        stats = self.getstat()
        return self._format_date(stats[7])

    def get_last_modification_time(self):
        stats = self.getstat()
        return self._format_date(stats[8])

    def get_last_status(self):
        stats = self.getstat()
        return self._format_date(stats[9])


def main():
    mon_objet = FileObj('/tmp')
    for size, file in mon_objet.get_files_sorted_by_size():
        print '%s: %s octets' % (file, size)

if __name__ == "__main__":
    main()
```