W6-S6-M2-Mini-projet-disk-usage

December 15, 2014

1 Utilisation du disque dur

1.1 Mini-Projet

1.1.1 Introduction

De temps en temps, vous vous rendez compte que votre disque dur est plein ou que vous avez rempli votre quota. En application de la loi de Murphy, en général c'est à un mauvais moment ; il est vrai qu'on a toujours mieux à faire que de nettoyer un disque.

1.1.2 Objectifs

Dans ce mini-projet, nous allons écrire un utilitaire permettant de nous aider dans ce genre de situations. Les objectifs que l'on pourrait avoir sont : * de trouver rapidement les gros répertoires, en partant d'une racine ou d'un répertoire utilisateur ; * de stocker les données de taille de façon à ne pas avoir à attendre plusieurs minutes à recalculer sans cesse les tailles des différents morceaux ; * et notamment de pouvoir lancer toutes les nuits un scan silencieux, de façon à avoir immédiatement, dans la journée, accès à des informations - même approchées - de nature à identifier le ou les répertoires qui pose problème ou qui a un fort potentiel de libération d'espace.

1.1.3 Approche

Pour cela on conçoit un système simple qui fonctionne en deux passes : * une première passe de type *batch* qui stocke dans chaque répertoire, dans un fichier spécial (nous avons utilisé le nom `.du`) la taille totale de ce répertoire, * une seconde passe interactive, qui peut * afficher les tailles des sous-répertoires triés, précisément, par taille, * naviguer dans les répertoires sur cette base, * et procéder au nettoyage proprement dit.

Aussi voici les choix que j'ai faits pour mon implémentation : * un seul module qui contient tout le programme, et qui s'appelle `diskusage.py` * par défaut le programme ne lance que la seconde passe * on peut ne lancer que la première passe avec l'option `-1` * ou les deux passes en séquence avec l'option `--both`

Voici l'aide en ligne

```
$ diskusage.py --help
usage: diskusage.py [-h] [-1] [-b] [-v] dir

positional arguments:
  dir

optional arguments:
  -h, --help         show this help message and exit
  -1, --pass1        Run pass1, that computes .du in all subdirs
  -b, --both-passes  Run pass1, that computes .du in all subdirs,
                     and then pass2 that is interactive
```

```
  -v, --verbose      turn on verbose output
```

Vous reconnaissez sans doute l'utilisation à nouveau de `ArgumentParser` importé du module `argparse` pour la définition de la syntaxe d'appel de `diskusage.py`

1.1.4 Exemple d'utilisation

Nous publions un directory de test pour vous permettre de vérifier vos résultats, comme d'habitude dans les formats suivants : * format tar * format tgz * format zip

qui donne une structure de fichiers telles que ceci :

La première passe Si j'installe cette structure sur mon propre disque, voici ce que j'obtiens :

```
% diskusage.py spam
Welcome to inspection of path spam
-------- Path spam has a total size of xxx 0 xxx
1    xxx 0 xxx   big
2    xxx 0 xxx   little
3    xxx 0 xxx   medium
4    xxx 0 xxx   small
Enter number (h for help) q
%
```

Ce qui est 'normal' ou en tous cas attendu, car je n'ai pas lancé la première passe:

```
% find spam -name .du
% diskusage.py -1 spam
%
```

La deuxième passe Maintenant si je recommence, l'outil me montre les directories triés par taille, le plus gros en dernier - parce que c'est sans doute celui qui m'intéresse le plus :

```
% diskusage.py spam
Welcome to inspection of path spam
-------- Path spam has a total size of 3.16 MiB
1     1.15 KiB   little
2   139.73 KiB   small
3     1.09 MiB   medium
4     1.93 MiB   big
%
```

À ce stade-là je peux naviguer dans l'arbre en tapant :

- soit un nombre pour me déplacer dans l'arbre

```
Enter number (h for help) 2
-------- Path spam/small has a total size of 139.73 KiB
Enter number (h for help)
```

- soit `u` ou `..` pour remonter

```
Enter number (h for help) u
-------- Path spam has a total size of 3.16 MiB
1     1.15 KiB   little
2   139.73 KiB   small
3     1.09 MiB   medium
4     1.93 MiB   big
Enter number (h for help)
```

- soit '+' (ou une ligne vide) pour choisir le répertoire le plus gros

```
 Enter number (h for help)
-------- Path spam/big has a total size of 1.93 MiB
1         68 B   empty
2     4.60 KiB   f
3   126.17 KiB   d
4   252.17 KiB   b
5   558.23 KiB   a
6  1008.17 KiB   c
Enter number (h for help)
```

- soit `l` pour lister les **fichiers** (jusqu'ici la commande n'a listé que des répertoires)

```
 Enter number (h for help) l
---- Plain files in spam/big
F          8 B   .du
F        576 B   zfile-01
F     1.12 KiB   zfile-02
F     2.25 KiB   zfile-03
F     4.50 KiB   zfile-04
F     6.00 KiB   .DS_Store
F        9 KiB   zfile-05
Enter number (h for help)
```

- voici d'ailleurs l'aide en ligne

```
Enter number (h for help) h
num go to listed directory
+   (default) go to last (and thus biggest) directory
u   go one step up - can be also '0' or '..'
l   list files in the current directory
.   come again (stay in place)
!   re-run pass1
v   toggle verbose on and off
q   quit
h   this help
```

1.1.5 Que faut-il faire au juste ?

Tout ce qui précède vous est donné à titre purement indicatif, pour vous décrire ce que fait l'implémentation qui sera donnée comme corrigé de cet exercice.

Gardez bien présent à l'esprit toutefois qu'il ne s'agit pas d'une évaluation, aussi vous pouvez librement vous inspirer de cette implémentation. Dans l'état que je viens de décrire, l'outil est passif (il ne permet pas de détruire des fichiers), mais relativement exhaustif (vous pouvez voir toutes les tailles de tous les répertoires et de tous les fichiers). À vous de choisir l'étendue de ce que vous voulez faire.

Une variante **plus simple** consisterait à se contenter de ne montrer que les répertoires, puisqu'en général une fois qu'on a isolé le répertoire on peut utiliser un navigateur de fichiers plus classique.

Une variante **plus compliquée** consisterait à permettre des destructions de fichiers ou de répertoires nativement dans l'outil. Dans ce cas il serait bon de mettre à jour les tailles enregistrées dans les `.du`.

De même, ma version incorpore un mode bavard (*verbose*) ; lorsqu'il est activé on peut montrer plus de détails sur le fonctionnement interne de l'outil, par exemple en imprimant une ligne à chaque fois qu'on calcule vraiment la taille d'un répertoire - pour mieux comprendre ce qui se passe. Bien entendu, pour commencer, vous n'avez pas besoin de prévoir un mode bavard.

De même, vous voyez dans l'aide en ligne quelques utilitaires supplémentaires, comme notamment `!` pour recalculer les tailles à partir de l'état du disque. En principe, ajouter ce genre de features ne prend que quelques lignes de code, mais là encore c'est à vous de voir.

Enfin, le code étant extrêmement basique en termes d'utilisation de librairies, je vous propose si vous avez le temps d'essayer de faire fonctionner votre code aussi en python3 - c'est-à-dire, une fois que votre code fonctionne en python2, d'essayer de le modifier à la marge pour obtenir un code unique qui fonctionne avec les deux versions. Pour cela attendez tout de même d'avoir lu le complément que nous consacrons à ce sujet en Semaine 7.

Quelques indices Une des difficultés de cet exercice est de gérer le fait que le système est 'vivant' pendant la sesssion ; aussi les données cachées dans les `.du` peuvent être anciennes. Ces deux facteurs font qu'il se peut que certains répertoires ont un `.du` et pas d'autres. Ici j'ai adopté une stratégie simple qui est de retourner une taille nulle (pour pouvoir tout de même afficher le résultat de la somme) et de faire en sorte que l'impression d'une taille nulle attire l'oeil (voir le tout premier exemple ci-dessus).

Notez aussi, pour ceux qui n'y sont pas habitués, qu'**un répertoire a une taille propre**. Du point de vue de l'utilisateur un répertoire ne contient 'rien' mais il faut de la place pour ranger les noms des fichiers. C'est en général négligeable, mais avec un grand nombre de petits fichiers c'est mieux d'en tenir compte.

Si vous voulez aborder l'exercice dans une optique 'niveau avancé', vous pouvez vous arrêter là, et essayer d'implémenter l'outil sans plus d'indications.

Librairies En termes de librairies tierces, c'est difficile de faire plus simple ; notre implémentation utilise uniquement : * le module `os` et spécifiquement * `os.walk`, `os.listdir`, * `os.path` et dans ce module * `join`, `getsize`, `exists`, `isdir`, `isfile`, et `dirname`, * et donc `argparse.ArgumentParser`

1.1.6 Classes et fonctions

Toujours à titre indicatif, voici comment est conçue notre implémentation. On a défini 3 classes :

- `HumanReadableSize` est une classe qui ne sert que pour l'affichage des tailles sous la forme que vous avez vue dans les exemples comme `139.73 KiB` ou `1.09 MiB`
- `Cache` est la classe qui mémorise la taille (totale) des différents répertoires
- et `ToplevelDir` est la classe qui implémente la logique de traitement du répertoire d'entrée.

Enfin la fonction `main` est le point d'entrée dans le programme.

Notons à titre de curiosité, avant de lister les différentes méthodes, que * `HumanReadableSize` hérite de `int` ; on a redéfini `__repr__` pour implémenter le format dans lequel on veut afficher les tailles en octets ; * `Cache` hérite de `dict` ; on a redéfini `__getitem__` et `__setitem__` de façon à utiliser `cache[path]` et `cache[path]=size` pour mémoriser le résultat à la fois en mémoire pour le processus courant, et sur disque dans les différents fichiers `.du`

Classe `HumanReadableSize`

```
Help on class HumanReadableSize in module diskusage:

class HumanReadableSize(__builtin__.int)
 |  helper class for displaying size in bytes
 |  in a human-readable form
```

```
 |
 |  Method resolution order:
 |      HumanReadableSize
 |      __builtin__.int
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __repr__(self)
 |      Display size in bytes
 |
 |  __str__ = __repr__(self)
 |
 |  ----------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  LABELS = [(6, 'EiB'), (5, 'PiB'), (4, 'TiB'), (3, 'GiB'), ...
 |
 |  UNIT_LABELS = [(1152921504606846976, 'EiB'), ...
 |
```

Classe `Cache`

```
Help on class Cache in module diskusage:

class Cache(__builtin__.dict)
 |  a dictionary {path: size_in_bytes}
 |
 |  this is also linked to the file system and the .du files
 |  meaning that
 |  (*) cache[path] looks in path/.du if not yet in memory
 |      if nothing else works (not in memory and not in .du)
 |      we return 0
 |  (*) cache[path] = size also writes path/.du
 |      if permission is granted
 |
 |  Method resolution order:
 |      Cache
 |      __builtin__.dict
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __getitem__(self, path)
 |      Look in memory cache first, then in the .du file
 |      returns 0 if nothing works
 |
 |  __init__(self, verbose=False)
 |
 |  __setitem__(self, path, size)
 |      remembers path cache in dictionary
 |      and stores in special file as  far as possible
 |
```

```
 |      ignores if not possible for any reason
 |      like Permission Denied or the like
 |
 |  ----------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  special_name = '.du'
 |
```

Classe ToplevelDir

```
Help on class ToplevelDir in module diskusage:

class ToplevelDir(__builtin__.object)
 |  toplevel object - only one is created
 |  for the directory that diskusage.py is run on
 |
 |  it can run pass1()
 |  it has one instance of Cache for keeping track
 |     of the sizes of all subdirs
 |  it can also run pass2
 |
 |  Methods defined here:
 |
 |  __init__(self, path, verbose=False)
 |
 |  list_files(self, subpath)
 |      passive list of plain files in a given dir
 |      the ones in *that* directory, not the subtree
 |      just list with biggest file last
 |
 |      it's easier to re-read the file size here
 |      as there is no recursion
 |      would need to be optimized for directories
 |      with a large number of plain files
 |
 |  move_to_subdir(self, subpath)
 |      this is the active part of pass2
 |      it is the place where we prompt
 |      for the user's answer and
 |      where we implement the mainloop
 |
 |      this method returns the path for the next
 |      subtree to visit (can also be one step up)
 |
 |      we show the immediate subdirs sorted
 |      (biggest comes last)
 |      and can thus be selected using '+'
 |
 |      subdirs are listed with a number that
 |      can be selected for moving down the tree
 |
 |  pass1(self)
```

```
 |      scans a whole tree, and writes
 |      individual (total) size in .du
 |
 |      this is done through a Cache object so
 |      that if we run both passes in the same process
 |      pass2 will not even need to read .du files
 |
 |  pass2(self)
 |      entry point for pass2
 |
 |  ----------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  help_message = 'number\tgo to listed directory\n+\t.....
```

Fonction `main`

```
Help on function main in module diskusage:

main()
    The entry point for diskusage.py

    This function parses the command line arguments
    using an instance of ArgumentParser

    It essentially creates an instance of ToplevelDir
    and sends it the pass1() and/or pass2() methods

    It returns an int suitable to be returned to the OS
    that is to say 0 when everything is fine and 1 otherwise
```

À nouveau toutes ces précisions sont données pour vous donner des idées mais il n'est pas du tout obligatoire de procéder de cette manière.

À vous de jouer.