Langages de script (LS4)

Examen du 27 mai 2011

Durée: 3 heures, sans documents

- Vous devez éteindre et ranger vos téléphones.
- Les programmes sont à faire en PYTHON3.
- Chaque exercice doit être rédigé sur une copie distincte.
- L'annexe du sujet contient
  - des rappels de PYTHON ;
  - la documentation de `grep` ;
  - la documentation du module `re`.

Exercice 1 – *Presque GREP*

Dans cet exercice vous écrirez un script qui reproduit les fonctionnalités principales de l'outil fameux en ligne de commande GREP (voir annexe A.2). Pour simplifier le problème on supposera que le motif est une expression régulière PYTHON (voir annexe A.3).

1. Écrivez une fonction `cherche(motif,fichier)` qui renvoie la liste des numéros de toutes les lignes du `fichier` contenant une partie (appelée concordance) correspondant à l'expression régulière `motif`.

2. Écrivez une fonction `cherche_mot(motif,fichier)` similaire mais avec comme contrainte que la concordance forme un mot complet.

3. Écrivez un script `pygrep.py` qui doit être appelé par la ligne de commande :
`pygrep.py [options] MOTIF FICHIER`
et qui se comporte comme `grep` pour les cas sans option, avec options `-c`, `-w` et `-n` (une seule option à la fois). Si la ligne de commande n'est pas conforme, un petit message d'aide doit s'afficher.

Exercice 2 – *Carnet de recettes*

Le but de cet exercice est de faire un script permettant de gérer un carnet de recettes.

1. Une recette est une chaîne de caractères ayant la forme suivante :

```
Nom: Gâteau au chocolat
Type: Dessert
Ingrédients: Farine, oeufs, chocolat, sucre, beurre
Étapes:
1- battre les blancs d'oeufs
2- fondre le beurre et le chocolat
3- mélanger tous les ingrédients
4- verser dans un moule et faire cuire à 200 degré pendant 30min
```

Chaque recette sera représentée par un dictionnaire dont les clefs sont `Type, Ingr, Etap` :
– À la clef `Type`, on associe un caractère `E, P, D, B` représentant respectivement entrée, plat, dessert, boisson.
– À la clef `Ingr`, on associe l'ensemble des ingrédients.
– À la clef `Etap`, on associe la liste des étapes.
Écrivez une fonction appelée `lire_recette` prenant en entrée la recette sous la forme d'une chaîne de caractères et renvoyant le dictionnaire correspondant à la recette.

2. Un carnet de recettes est un fichier regroupant différentes recettes, séparées par des lignes blanches.
   Le carnet sera représenté par un dictionnaire dont les clefs sont les noms des recettes et les valeurs sont les dictionnaires associés à chacune des recettes.
   Écrivez une fonction appelée `lire_carnet` prenant en entrée le fichier carnet et renvoyant le dictionnaire représentant le carnet.
3. Écrivez une fonction appelée `sauve_carnet` prenant en entrée un dictionnaire et le nom de fichier dans lequel on veut écrire les recettes au format décrit à la question précédente.
4. Écrivez une fonction `ajoute_recette` permettant à un utilisateur d'ajouter de façon interactive une recette au dictionnaire représentant le carnet.
5. Écrivez une fonction `supprime_recette` permettant à un utilisateur de supprimer de façon interactive une recette au dictionnaire représentant le carnet.
6. Écrivez une fonction `recherche_recette` permettant d'afficher tous les noms de recettes contenant un ingrédient donné en argument sous la forme d'une chaîne de caractères.
7. Écrivez la fonction principale d'un script permettant de gérer un carnet de recettes.

Exercice 3 – *Pythoneries et générateurs*

Amusements pythonesques : Que font les fonctions suivantes ? Répondez avec **une seule phrase** et si nécessaire, un exemple. Vous pourrez consulter la partie A.1 pour répondre.

1. On suppose que `selectors` est une liste de booléens.

```
def compress(data, selectors):
    return (d for d, s in zip(data, selectors) if s)
```

2.
```
def mystere(liste):
    dict([(x, liste.count(x)) for x in set(liste)])
```

3.
```
def math(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
```

Comprendre les générateurs : Nous allons maintenant nous intéresser aux générateurs, un outil très puissant permettant la création d'objets sur lesquels on peut itérer. Un générateur utilise le mot clef `def` comme les fonctions mais au lieu de `return` c'est le mot clef `yield` qui est utilisé. On ne peut pas appeler directement le générateur mais on peut itérer dessus. A chaque passage de la boucle, l'exécution reprendra juste après l'appel à `yield`, toutes les variables ayant été sauvegardées.

```
>>> def toto():
...     yield 1
...     yield 2
...     yield 3
...
>>> toto()
<generator object toto at 0xb7549144>
>>> for x in toto():
...     print(x)
...
1
2
3
```

On peut évidemment faire des choses plus intéressantes avec, comme par exemple un générateur qui permet d'itérer directement un élément sur deux :

```
>>> def it_un_sur_deux(liste):
...     for x, y in enumerate(liste):
...             if  x%2 == 0:
...                     yield y
...
>>> for x in it_un_sur_deux([1,2,3,4,5,6,7]):
...     print(x)
...
1
3
5
7
```

Que font les générateurs suivants ? Encore une fois, répondez avec une seule phrase, et si nécessaire un exemple.

4.
```
def gen1(x, y):
    k = x
    while k < y:
        yield k
        k = k + 1
```

5.
```
def gen2(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

6.
```
def gen3(iterable):
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
              yield element
```

Écrire les générateurs :

7. `unique_everseen` qui permet de lister les éléments de façon unique.
Par exemple `unique_everseen('AAAABBBCCDAABBB')` itérera sur `A B C D`.

8. `takewhile(predicate, iterable)` où `predicate` est un prédicat, c'est à dire une fonction booléenne à un argument. Ce générateur doit itérer sur l'argument `iterable`, tant que le prédicat est vrai. Par exemple `takewhile(plus_petit_que_cinq, [1,4,6,4,1])` itérera sur `1 4`, en supposant que `plus_petit_que_cinq` fasse ce que son nom suggère.

A Annexe

A.1 Rappel de quelques éléments de PYTHON

– `enumerate()` permet d'itérer en même temps sur les positions et les éléments d'une liste.

```
>>> for i, x in enumerate(['a', 'b', 'c']):
...     print(i,x)
...
0 a
1 b
2 c
```

– `zip()` permet d'itérer sur les couples de deux objets itérables. Si un objet est plus court que l'autre, zip s'arrête là. Ex :

```
>>>for x, y in zip([1, 2, 3], ['a', 'b', 'c', 'd']):
...       print(x, y)
1 a
2 b
3 c
```

– On peut factoriser les affectations de cette manière :

```
x, y = a, b
```

Cela permet également d'échanger des valeurs (les affectations se font en parallèle) :

```
x, y = y, x
```

– `break` permet de sortir d'une boucle : Ex :

```
>>> for x in [1, 2, 3, 4, 5]:
...     if x == 3:
...             break
...     else:
...             print(x)
...
1
2
```

A.2 Page `man` de `grep`

```
SYNOPSIS

grep [options] MOTIF FICHIER

DESCRIPTION

 Grep recherche dans le FICHIER indiqué les lignes correspondant à un certain MOTIF.
 Par défaut, grep affiche les lignes qui correspondent au motif.

OPTIONS

-c  Ne pas afficher les résultats normaux. À la place, afficher un décompte des
lignes correspondantes pour le fichier
```

-n Ajouter à chaque ligne de sortie un préfixe contenant son numéro dans le fichier.

-w Ne sélectionner que les lignes contenant une concordance formant un mot complet.

A.3 Module `re` - descriptif basé sur le livre de Harold Erbin

Syntaxe Les regex répondent à une syntaxe très codifiée et possèdent de nombreux symboles ayant un sens particulier. Pour débuter, tout caractère alphanumérique n'a pas de signification spéciale : A correspond simplement à la lettre A, 1 au chiffre 1, etc. Quant aux principaux symboles spéciaux, il sont :

`.` : désigne n'importe quel caractère ;

`^` : indique que le début de la chaine doit correspondre ;

`$` : indique que la fin de la chaine doit correspondre ;

`{n}` : indique que le caractère précédent doit être répété n fois.

`{n,m}` : indique que le caractère précédent doit être répété entre n et m fois.

`*` : le caractère précédent peut être répété aucune ou plusieurs fois. Par exemple, à ab* peuvent correspondre : a, ab, ou a suivi d'un nombre quelconque de b.

`+` : le caractère précédent peut être répété une ou plusieurs fois. Par exemple, à ab+ correspond un a suivi d'un nombre quelconque de b.

`?` : le caractère précédent peut être répété zéro ou une fois. Par exemple, à ab ? correspondent ab et a.

L'antislash permet d'échapper tous ces caractères spéciaux. Les crochets [] permettent d'indiquer une plage de caractère, par exemple [e-h] correspondra à e, f, g ou h. Finalement, il reste quelques caractères spéciaux assez utiles :

`\w` : il correspond à tout caractère alphabétique, c'est à dire qu'il est équivalent à [a-zA-Z] ;

`\W` : il correspond à tout ce qui n'est pas un caractère alphabétique ;

`\b` : il correspond à la frontière (début ou fin) d'un mot ;

`\d` : il correspond à tout caractère numérique, c'est à dire qu'il est équivalent à [0-9] ;

`\D` : il correspond à tout ce qui n'est pas un caractère numérique.

Utilisation

`re.search(pattern, string)` Cherche le motif dans la chaîne passée en argument et retourne un `MatchObject` si des correspondances sont trouvées, sinon retourne `None`.

`re.split(pattern, string)` Découpe la chaine string selon les occurences du motif.

```
>>> re.split(r'\W', 'Truth is beautiful, without doubt.')
['Truth', 'is', 'beautiful', '', 'without', 'doubt', '']
```

`re.findall(pattern, string)` Retourne toutes les sous-chaines de `string` correspondant au motif.

`re.sub(pattern, repl, string)` Retourne la chaîne `string` où le motif a été remplacé par `repl`.