Série 4

Quatrième série : les expressions régulières

1. Éléments nécessaires pour la réalisation des exercices

Afin de réaliser les différents activités de cette série de TP, il est indispensable de d'avoir :

— Syntaxe des expressions régulières (notation POSIX) expliquée dans le chapitre 3 du cours.

Exercice 1

Utilisez un éditeur de texte supportant les expressions régulières (Notepad++, GEdit, Kate, Visual Code Studio, etc). A noter que dans le cas de Visual Studio Code, le nommage de groupes dans la substitution se fait par $1, $2, etc. au lieu \1, \2, etc.

Activité 1

Chargez le fichier `data.txt` dans l'éditeur de votre choix. Trouvez comment rechercher les motifs suivants :

Tâches à réaliser

1. les mots commençant par une majuscule
2. les mots se trouvant à la fin d'une phrase
3. les mots placés entre parenthèses
4. les phrases ne contenant aucun chiffre
5. les mots se répétant deux fois de suite tel que les deux occurrences sont séparées par deux points (:)

Activité 2

Utilisez l'éditeur pour apporter les modifications suivantes.

Tâches à réaliser

1. les mots entre parenthèses seront placés entre crochets
2. chaque phrase se tient sur une ligne séparée
3. lorsqu'un mot (composé de lettres) est suivi par un nombre suivi lui-même par le premier mot, alors remplacer ce motif par le mot suivi par le nombre uniquement (supprimer la deuxième occurrence)

Eléments à apprendre : Module re en Python

Le module `re` implémente les expressions régulières et offre un certains nombre de fonctions utiles :

→ La fonction `match(reg_ex,string)` permet de chercher un motif correspondant `reg_ex` dans `string` . S'il y a une occurrence, alors un objet `re.Match` est retourné, sinon `None` .

◇ `m=re.match("a+","aabc")` retourne un objet `m` . En invoquant `m.group(0)` , on obtient `"aa"` . Dans le cas `m=re.match("ba+","aabc")` , on obtient `None`.

→ La fonction `findall(reg_ex,string)` permet de rendre une liste des chaînes de `string` correspondant à `reg_ex` .

◇ `re.findall("a+","aabaaaacbcaaba")` retourne `["aa","aaaa","aa","a"]` .

→ La fonction `fullmatch(reg_ex,string)` est similaire à `match` sauf que la correspondance se fait avec toute la chaîne et non pas une sous-chaîne seulement.

◇ L'appel de `re.fullmatch("a+","aabc")` retourne `None` . Cependant, l'appel `re.fullmatch("a+bc","aabc")` retourne un objet Match correspondant à la chaîne toute entière.

Exercice 2

On considère un fichier contenant des lignes dont la forme doit être : $entier_1$: $entier_2$,..., $entier_n$. Tous les entiers sont positifs (sans possibilité d'utiliser des signes).

Activité 1

Ecrivez une fonction gardant uniquement les lignes correctement écrites.

En-tête

```
def filter_file(file)
```

Test

```
def filter_file("data2.txt")
```
(le fichier sera fourni)

Activité 2

Ecrivez une fonction qui garde uniquement les lignes correctes dans le cas suivant : pour une ligne $entier_1$: $entier_2$,..., $entier_n$, il faut que $entier_1 = \sum_2^n entier_i$.

En-tête

```
def context_filter_file(file)
```

Test

```
def context_filter_file("data2.txt")
```

Eléments à apprendre : les dictionnaires en Python

→ Un dictionnaire est défini par : `{...}` (attention à la différence avec les ensembles).

◇ Exemple : `my_dict={"a' :5,"b" :2,"e" :" e" :5,"f" :1," z":[6,7]}`

→ Accès et mises à jour :

◇ Accès : `my_dict[key]` . Exemple : `my_dict["a"]` donne 5

◇ L'accès à une clé inexistante génère une erreur (`KeyError`). On peut tester l'existence d'une clé dans un dictionnaire par la condition : `key in my_dict`

◇ Mise à jour : `my_dict[key]=value`

→ Programmer avec les dictionnaires :

◇ Parcourir les clés d'un dictionnaire : `for k in my_dict:print(k)`

◇ Parcourir les valeurs d'un dictionnaire (option 1) :
```
for k in my_dict:print(my_list[k])
```

◇ Parcourir les valeurs d'un dictionnaire (option 2) :
```
for v in my_dict.values():print(v)
```

◇ Parcourir les clés et valeurs d'un dictionnaire :
```
for k,v in my_dict.item():print(k," :",v)
```

Exercice 3

On veut concevoir une fonction permettant de saisir les informations d'un enregistrement.

Activité 2

La saisie est guidée par un dictionnaire dont les clés représentent les champs de l'enregistrement. Les valeurs du dictionnaire contiennent deux clés : la première est `description` et représente une chaîne de caractères à afficher avant la saisie du champ, la deuxième est `forme` et représente une expression régulière modélisant les valeurs admissibles pour le champs. La fonction doit retourner le dictionnaire saisi.

En-tête

```
def input_record(record)
```

Test

Tester la fonction avec les champs suivants :

- **Code** : la description à afficher est `"Code de l'étudiant"` . Ce champ commence par `"UN"` suivi de 6 chiffres.
- **Nom** : la description à afficher est `"Nom de l'étudiant"` . Ce champ commence par une lettre suivie de lettres et/ou espaces.
- **Prénom** : la description à afficher est `"Prénom de l'étudiant"` . Ce champ commence par une lettre suivie de lettres et/ou espaces.
- **Date de naissance** : la description à afficher est `"Date de naissance"` . Ce champ spécifie le jour par un entier à un ou deux chiffres, le mois sur un ou deux chiffres et l'année sur quatre chiffres.
- **Email** : la description à afficher est `"Email de l'étudiant"` . Ce champ commence par une suite non-vide de caractères, suivie de `"@"` , suivi d'une suite non-vide de caractères. L'adresse doit contenir `"@"` une seule fois.
- **Note** : la description à afficher est `"Note de l'étudiant"` . Ce champ permet de saisir la note d'un étudiant sous la forme xx.xx, où x est un chiffre et seul le premier chiffre est obligatoire.