Deuxième partie

Support de TP

Série 1

Prise en main

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 :

— Connaissance de certains éléments du langage Python :
  — Syntaxe de base du langage : indentation, affectations, structures de contrôle (`if` , `while` , `for` , ...), fonctions, etc.
— Des notions générales en algorithmique.
— A noter que la série présente uniquement les fonctions de bases. Elle ne peut, en aucun cas, se substituer de la documentation officielle du langage.

Eléments à apprendre : (exercice 1) les listes en Python

→ Une liste est définie par la syntaxe : `[...]`

◇ Exemple : `my_list=[1,2,5,6]` , `z=[]`

◇ Une liste peut contenir n'importe quel type de données (même hétérogènes). Elle peut même contenir des listes : `[1,2,[3,4],4.5]`

→ L'indexation se fait par `list[index]` :

◇ Exemple : `my_list[2]` renvoie 5

◇ Mise à jour : `my_list[1]=9`

→ On peut sélectionner un *slice* (une partie) d'une liste :

◇ Exemple : `my_list[0:2]` renvoie [1,2]

◇ Exemple : `my_list[1:]` renvoie [2,5,6]

◇ Exemple : `my_list[:3]` renvoie [1,2,5]

→ Opérations sur les listes :

◇ Longueur : `len(m_list)` renvoie 4

◇ Le nombre d'occurrence d'un élément : `my_list.count(e)`

◇ Concaténation : `[1,2]+[4,6]` donne [1,2,4,6]

◇ Répéter une liste : `[1,2]*3` donne [1,2,1,2,1,2]

◇ Vérifier si un élément figure dans une liste : `x in my_list` (résultat booléen)

◇ La fonction range : `list(range(0,5))` construit la liste `[0,1,2,3,4]` .

→ Programmer avec les listes :

◇ Parcourir les éléments d'une liste : `for e in my_list:print(e)`

◇ Parcourir les éléments par leurs indices :
```
for i in range(len(my_list)):print(my_list[i])
```

◇ Parcourir les indices et les éléments d'une liste :
```
for i,e in enumerate(my_list):print(i,"=>",e)
```

→ Compréhension : générer une liste d'éléments vérifiant un critère donné :

◇ Générer les nombres paires de my_list : `[ e for e my_list if e % 2==0 ]`

◇ Renvoyer tous les nombres multiples de 5 inférieurs à 100 : `[ e for e in range(101) if e % 5==0 ]`

Exercice 1

Écrire les fonctions suivantes (il faut écrire un programme testant les fonctions construites) :

Activité 1

Soustraire deux listes (garder les éléments de la première liste qui ne figurent pas dans la deuxième.

En-tête

```
def substract(L1,L2)
```

Test

```
substract([5,8,12,1,13,17],[6,7,12,5,17])
```

Activité 2

Répéter les éléments d'une liste en fonction d'une autre liste (exemple : L1=[6,1] et L2=[2,4], le résultat est : [[6,6],[1,1,1,1]]

En-tête

```
def repeat_by_list(L1,L2)
```

Test

```
repeat_by_list([6,1],[2,4])
```

Activité 3

Réarranger les éléments d'une liste : premier élément, dernier élément, deuxième élément, avant-dernier élément, troisième élément, ...

En-tête

```
def symmetric_browse(L)
```

Test

```
symmetric_browse([1,3,5,7,9,8,6,4,2])
```

Eléments à apprendre : (exercice 2) les chaînes de caractères en Python

→ La plupart des fonctions des listes sont applicables aux chaînes.

→ Une chaîne de caractères est une séquence de caractères (placées entre deux " ou deux '). On peut également avoir une chaînes sur plusieurs lignes avec les délimiteurs """.

◇ `my_str="abcdef"`

→ Une chaîne est une liste immuable

◇ `L'affectation s[2]="v"` provoque une erreur

→ Opérations sur les chaînes :

◇ La fonction `chr(...)` donne le caractère dont le code ASCII ou unicode est passé comme paramètre (exemple `chr(1585)`

◇ La fonction `ord(...)` donne le code ASCII ou unicode du caractère passé comme paramètre (exemple : `ord("a")`)

◇ Longueur : `len(my_str)` renvoie 5

◇ Le nombre d'occurrence d'un caractère : `my_str.count(e)`

◇ Concaténation : `"ab"+"cd"` donne "abcd"

◇ Répéter une chaîne : `"abc"*3` donne "abcabcabc"

◇ Vérifier si un élément figure dans une liste avec le test `x in my_list` (le résultat est booléen). Ceci fonctionne pour un caractère ou pour une chaîne (test de sous-chaînes).

◇ Recherche le premier (resp. le dernier) indice d'une sous-chaîne `my_str.find(sub_str)` (resp. `my_str.rfind(...)`)

◇ Joindre les éléments d'une liste pour en faire une chaîne `my_str.join(my_list)` (il faut que la éléments de la liste soient des chaînes de caractères).

◇ Formater une chaînes selon un format (option 1) : `str_format(a1,a2,...)` . Exemple : `"{}+{}={}".format(3,4,3+4)` .

◇ Formater une chaînes selon un format (option 2, disponible à partir de Python 3.6) : `f"..."` . Exemple : `f" {3}+{4}={3+4} "`.

◇ Diviser une chaînes de caractères par rapport à un séparateur : `my_str.split(sub)`

◇ Remplacer toutes les occurrences d'une chaînes par une autre chaîne : `my_str.replace(sub1,sub2)`

Exercice 2

Écrire les fonctions suivantes (il faut écrire un programme testant les fonctions construites) :

Activité 1

La fonction crange permettant de générer une chaîne contenant tous les caractères entre deux caractères (par exemple, `crange(["a","c"],["0","2"])` donne `"abc012"`).

En-tête

```
def crange(*cinterv)
```

Test

```
crange(["a","z"],["A","Z"],["0","9"])
```

Activité 2

Améliorer la fonction replace pour qu'elle prenne deux listes de chaînes de caractères [s1,s2,...], [t1,t2,...]. La fonction change s1 par t1, ensuite s2 par t2, etc.

En-tête

```
def replace_many(s,L1,L2)
```

Test

```
replace_many("aabc",["a","b","c"],["b","a","d"])
```

Activité 3

Implémenter une fonction replace conditionnelle. La chaîne sub1 est remplacée par sub3 uniquement si sub1 est suivie par sub2, sinon elle est remplacée par sub4.

En-tête

```
def conditional_replace(s,sub1,sub2,sub3,sub4)
```

Test

```
conditional_replace("abc=ded","=","f","=e","=f")
```

Série 2

Deuxième série : opérations sur les mots et les langages

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 :

— Connaissance du type `string` en Python.
  — Ne pas hésiter à consulter la documentation officielle
— Notions sur la récursivité
— Suivi le cours/TD du premier chapitre

Exercice 1

Écrire des les fonctions Python permettant de calculer :

Activité 1

La fonction `flatten_word(w_tuple)` transforme une liste de tuples représentant les symboles et leurs nombres en une chaîne de caractères. Par exemple, `flatten_word(("a",2),("b",4),("c",3))` renvoie la chaîne "aabbbbccc".

En-tête

```
def flatten_word(w_tuple)
```

Test

```
flatten_word(("a",2),("b",4),("a",2),("b",6),("c",1))
```

Activité 2

La fonction `unflatten_word(word)` effectue la transformation inverse de la fonction précédente. Par exemple, pour le mot "aabbbbccc", la fonction renvoie `("a",2),("b",4),("c",3)` .

En-tête

```
unflatten_word(word)
```

Test

```
unflatten_word("aabbbbccc")
```

Exercice 2

Écrire des les fonctions Python permettant de calculer :

Activité 1

La fonction `prefixes(word)` retourne la liste de tous les préfixes du mot *word*.

En-tête

```
def prefixes(word)
```

Test

```
prefixes("aaabbbccc")
```

Activité 2

La fonction `factors(word)` retourne la liste de tous les facteur du mot *word*.

En-tête

```
def factors(word)
```

Test

```
factors("aaabbbccc")
```

Activité 3

Une fonction permettant de calculer l'entrelacement d'un mot avec un symbole.

En-tête

```
def shuffle_symbol(word,c)
```

Test

```
shuffle_word("abba","c")
```

Activité 4

Une fonction permettant de calculer l'entrelacement de deux mots.

En-tête

```
def shuffle(w1,w2)
```

Test

```
shuffle_word("aaa","bbb")
```

Exercice 3

Pour chacun des langages suivants, écrire une fonction permettant d'accepter le langage. En d'autres termes, étant donné un mot du langage, la fonction doit décider si le mot appartient au langage ou non.

Activité 1

L_1 : les mots sur $\{a, b, c\}$ tel que le nombre de a est pair, le nombre de b est multiple de 3 et le nombre de c est multiple de 6. La lecture du mot se fait de gauche à droite uniquement (chaque symbole est lu une seule fois).

En-tête

```
def accept_l1(word)
```

Test

`accept_l1("ccabbaccbcc")` et `accept_l1("ccabbc")`

Activité 2

$L_2 = \{a^{2p}b^{3q}c^{2p+3q} | p, q \geqslant 0\}$. La lecture du mot se fait de gauche à droite uniquement (chaque symbole est lu une seule fois).

En-tête

```
def accept_l2(word)
```

Test

`accept_l2("aabbbccccccc")` et `accept_l2("aabbbcc")`

Activité 3

$L_3 = \{ww^R | w$ est un mot quelconque sur $\{"a", ..., "z"\}\}$. La lecture du mot se fait de gauche à droite uniquement (chaque symbole est lu une seule fois).

En-tête

```
def accept_l3(word)
```

Test

`accept_l3("abba")` et `accept_l3("aaabbcaa")`

Activité 4

$L_4 = \{wuw | w$ et u sont des mots quelconques sur $\{"a", ..., "z"\}\}$. w a une longueur non nulle.

En-tête

```
def accept_l4(word)
```

Test

`accept_l4("abcdabcd")` et `accept_l4("aababb")`

Activité 5

$L_5 = \{a^{2^n} | n \geqslant 0$. Il n'est pas autorisé de compter le nombre de a puis de le tester.

En-tête

```
def accept_l5(word)
```

Test

`accept_l5("aaaaaaaa")` et `accept_l5("aaaaaa")`

Activité 6

$L_6 = \{a^{n^2} | n \geqslant 0$. Il n'est pas autorisé de compter le nombre de a puis de le tester.

En-tête

```
def accept_l6(word)
```

Test

`accept_l6("aaaaaaaaa")` et `accept_l6("aaaaaa")`

Exercice 4

Soit la grammaire dont les règles de production sont données par : $S \rightarrow aSbS|\varepsilon$.

Activité 1

Écrire une fonction Python permettant de générer tous les mots dont la longueur est inférieure à une certaine limite.

En-tête

```
def generate(lim)
```

Test

```
generate(5)
```

Activité 2

Ecrire une deuxième fonction permettant de vérifier si un mot est généré par cette grammaire.

En-tête

```
def is_generated(w)
```

Test

```
is_generated("aabb") et is_generated("ababba")
```

Série 3

Troisième série : les AEF

1. Éléments nécessaires pour la réalisation des exercices

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

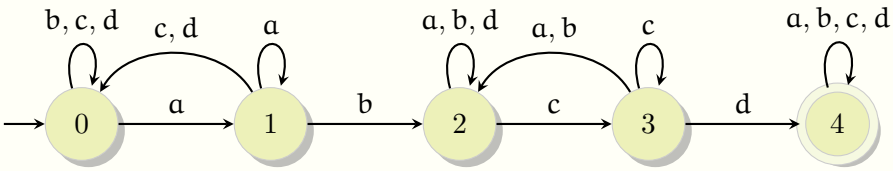
— Connaître le fonctionnement des AEF.

— Avoir des notions sur la récursivité

— Suivre le cours/TD du deuxième chapitre

Exercice 1

On considère le langage de tous les mots sur $\{a, b, c, d\}$ ayant le facteur ab et cd tel que l'occurrence de ab précède celle de cd. On considère alors l'AEF suivant :



Activité 1

Ecrire une fonction simulant cet automate. Pour chaque mot analysé, compter le nombre d'étapes nécessaires pour accepter ou rejeter les mots.

En-tête

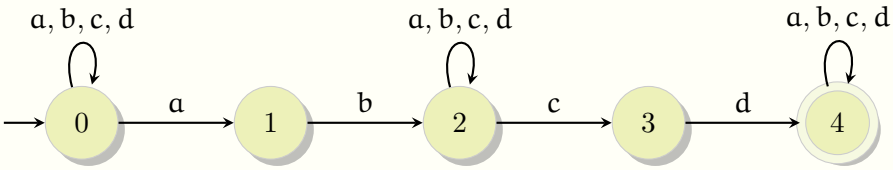
```
def simulate_fsa(word)
```

Test

`simulate_fsa("cdbaabbcdb")` et `simulate_fsa("acdbab")`

Exercice 2

On reprend le langage de l'exercice précédent. On utilise cette fois, l'AEF non-déterministe suivant :



Activité 2

Ecrire une fonction simulant cet automate. Pour chaque mot analysé, compter le nombre d'étapes nécessaires pour accepter ou rejeter les mots.

En-tête

```
def simulate_fsa(word)
```

Test

`simulate_ndet_fsa("cdbaabbcdb")` et `simulate_ndet_fsa("acdbab")`

Exercice 3

On s'intéresse aux chaînes de caractères contenant 0 et/ou 1 censées représenter des nombres binaires.

Activité 3

Ecrire une fonction permettant de calculer le reste de division sur 3 d'un nombre binaire représenté par une chaînes de caractères. Il n'est pas autorisé de transformer la chaîne en un entier ni d'utiliser les opérations d'addition, soustraction, multiplication, division ou modulo.

En-tête

```
def remainder_by_3(word)
```

Test

```
remainder_by_3("001001111")
remainder_by_3("111011")
remainder_by_3("101010")
```

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.