TD 11 — Dictionnaires

1 Un peu de théorie

Python fournit nativement une structure de *dictionnaire*, qui généalise la notion de liste. Au lieu que les éléments soient identifiées par leur indice, ils sont repérés par une *clé*, qui n'est pas forcément de type `int`. Les dictionnaires sont des objets *mutables*, dans lesquels chaque élément est un couple clé/valeurs. L'accès aux valeurs se fait par la clé, qui doit être unique.

Par exemple, on peut construire un dictionnaire dont chaque clé est un pays, et chaque valeur est la capitale :

```python
capitales = {'Estonie': 'Tallinn', 'Sénégal': 'Dakar', 'Inde': 'New Dehli'}
print(capitales)              # Affiche tout le dictionnaire.
print(capitales['Sénégal'])   # Affiche 'Dakar'.
print(capitales['Mexique'])   # Erreur : la clé n'existe pas.
```

Les dictionnaires sont de manière naturelle des objets itérables :

```python
for key, value in capitales.items():
    print(key + "; " + value)
```

Il est possible d'itérer sur les clés uniquement :

```python
for key in capitales.keys():
    print(key)
```

De même en itérant sur les valeurs uniquement :

```python
for values in capitales.values():
    print(values)
```

On peut ajouter simplement des entrées au dictionnaire en créant des nouvelles clés avec l'opérateur d'affectation `capitales['Australie'] = 'Camberra'`. Si la clé est déjà présente, la valeur associée est modifiée.

2 Exercices d'application

2.1 Commencer

Certains exercices de ce TD ont été repris de domart, sous licence CC-BY-NC-SA 4.0

Exercice 1. On veut mémoriser les notes qu'un·e élève a obtenues dans chaque matière. On utilise pour cela un dictionnaire dont chaque clé est le nom d'une matière, et la valeur est la liste des notes obtenues dans cette matière.
On prendra comme exemples :

```python
releve1 = {'maths': [12,15.5], 'anglais': [13,8,9.5], 'physique': [13,8,12,10,15.5]}
releve2 = {'philo': [12,15.5], 'anglais': [11,7,12.5, 15.5], 'svt': [13,8,12,10,16,15]}
```

1) Écrire un code Python qui ajoute une note à une matière.
2) Écrire un code qui calcule la moyenne d'une personne par matière. Enregistrer les résultats dans des dictionnaires.
3) Proposez un code qui permet de déterminer la meilleure note obtenue par une personne, toutes matières confondues, et la liste du ou des noms de(s) matière(s) dans laquelle ou lesquelles elle a obtenu cette note.

Exercice 2 (Dictionnaire des antécédents)**.** On considère le dictionnaire suivant :

```python
{"Paris": "P", "Lyon": "L", "Nantes": "N", "Lille": "L"}
```

Suivant les cas, une même valeur peut être associée à une ou plusieurs clés. Dans l'exemple précédent, la valeur `"L"` est associée aux clés `"Lyon"` et `"Lille"`, on les appelle les antécédents de `"L"`, tandis que `"P"` a la clé `"Paris"` pour seul et unique antécédent.
On peut ainsi construire le dictionnaire des antécédents :

```python
{"P": ["Paris"], "L": ["Lyon", "Lille"], "N": ["Nantes"]}
```

Écrire une fonction `antecedents`, qui reçoit en paramètre un dictionnaire `dico`, et qui qui renvoie le dictionnaire associant les valeurs de `dico` à la liste de leurs antécédents dans `dico`.

Exercice 3. On se donne une chaine de caractères appelée motif ainsi qu'un ensemble de règles du type « Si le caractère lu est un `a`, le remplacer par `ab` ».
L'ensemble de ces règles est stocké dans un dictionnaire Python dans lequel :

- une clé est un caractère lu ;
- la valeur associée est une chaine de caractères par laquelle le caractère doit être remplacé.

Si le caractère lu ne fait pas partie des clés du dictionnaire, il n'est associé à aucune règle et est donc recopié à l'identique.

Par exemple, on considère le dictionnaire de règles suivantes : `{'a': 'ac', 'b': 'd', 'c': 'ab'}`. On prend `'a'` comme motif de départ. Après une transformation, on obtient `'ac'`. Après deux transformations, on obtient `'acab'`. Après trois transformations, on obtient `'acabacd'`.

1) Écrire une fonction Python `transformation(motif, regles)` prend en argument un motif initial (une chaine de caractères) ainsi qu'un ensemble de règles (un dictionnaire) et renvoie la chaine obtenue après une transformation.

2) Écrire une fonction Python `n_transformations(motif, regles, n)` prend en argument un motif initial (une chaine de caractères) ainsi qu'un ensemble de règles (un dictionnaire) et un entier `n` et renvoie la chaine obtenue après `n` transformations.

2.2 Expressions bien ou mal parenthésées

Pour cette partie, vous programmerez les fonctions dans une fichier nommé `expressions.py`.

On dit qu'une expression est *bien parenthésée* si toute parenthèse ouverte est ensuite fermée, et une parenthèse ne peut être fermée que si elle a été préalablement ouverte.

Exercice 4. Écrire une fonction Python `bien_parenthesee(chaine)` qui prend en paramètre une expression, sous forme de chaine de caractères, et qui renvoie un booléen indiquant si elle est bien parenthésée.

On souhaite généraliser le test précédent, avec des parenthésages utilisant les symboles `()`, `{}`, `[]` et `<>`. Une expression est bien parenthésée si chaque symbole ouvrant correspond à un symbole fermant et si l'expression contenue à l'intérieur est bien parenthésée. Par exemple, l'extrait de code suivant est bien parenthésé :

```c
#include <stdio.h>

int main(void) {
    printf("%d", f());

    return EXIT_SUCCESS;
}

int f() {
    int liste[2] = {4, 2};

    return (10*liste[0] + liste[1]);
}
```

Exercice 5. Écrire une fonction `bien_parenthesee_multiple` qui détermine si une expression passée en paramètre est bien parenthésée avec les couples `()`, `{}`, `[]` et `<>`. La fonction renvoie un booléen. L'expression sera une chaine de caractères de longueur au plus 1000.

On cherche maintenant à enrichir notre test, en gérant le symbole `"` (guillemet) comme parenthésage, comme c'est le cas dans de nombreux langages de programamtion (puisque les guillements délimitent les chaines de caractères). Une chaine de caractères est bien parenthésée si tout guillemet ouvert est fermé par la suite. Mais attention : lorsque l'on est entre les guillemets, les autres parenthésages sont ignorés.

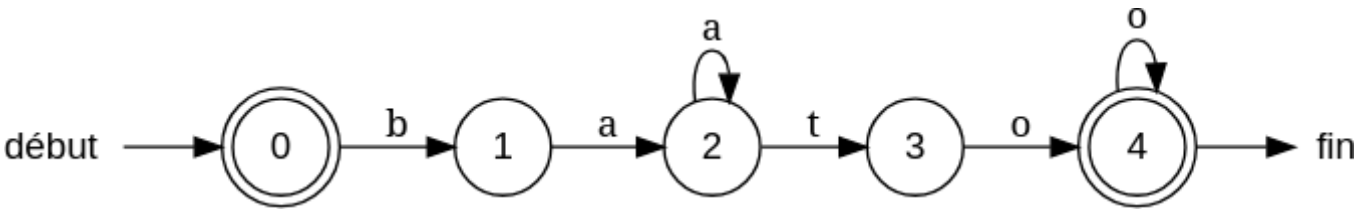
Exercice 6. Écrire une fonction `bien_delimitee_guillemets` qui vérifie sur une chaine de caractères (contenant éventuellement des guillemets…) est correctement parenthésée.

2.3 Automates et dictionnaires

Dans cette section, vous programmerez les fonctions demandées dans un fichier `automates.py`.

Une *expression régulière* est une chaine de caractères qui permet de représenter d'autres chaines de caractères.

Par exemple, la chaine `'ba+to+'` permet de décrire les mots `'bato'` et `'baato'`, `'batoo'` et `baatoooo'` : un caractère `'b'`, suivi d'au moins un `'a'`, puis un `'t'` et un ou plusieurs `'o'`.
Les expressions régulières peuvent être représentées sous forme d'automates. Ainsi, la chaine `'ba+to+'` est équivalente à l'automate ci-dessous :



Dans ce graphe, chaque *nœud* représente un *état* de l'automate. L'état 0 est l'état de début et 4 celui de fin. On parcourt l'automate de la façon suivante :

- on débute dans l'état de départ en lisant le premier caractère de la chaine ;
- si ce caractère est un `'b'`, on passe dans l'état 1 et on lit le caractère suivant ;
- dans l'état 1, si le caractère lu est un `'a'`, on passe dans l'état 2 et on lit le caractère suivant ;
- dans l'état 2, si le caractère lu est un `'a'`, on reste dans l'état 2 et on lit le caractère suivant ;
- etc.

Si on se trouve dans l'état final après avoir lu le dernier caractère de la chaine, alors celle-ci correspond au motif cherché. La chaine `'baato'` est reconnue car elle correspond au parcours $0 \to 1 \to 2 \to 2 \to 3 \to 4$.

On représente un automate par une série de règles stockées dans un dictionnaire Python. Chaque règle associe à un couple (état, caractère lu) la valeur du prochain état.
Ainsi l'automate ci-dessus est représenté par le dictionnaire suivant :

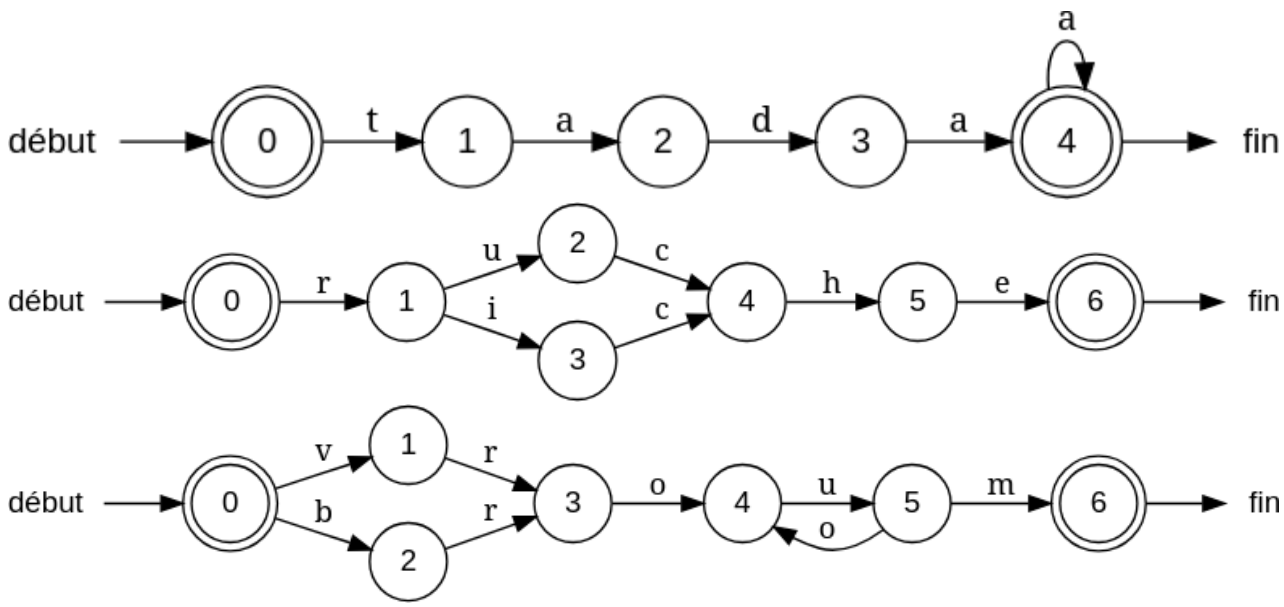
automates.py

```
automate_0 = {(0, 'b'): 1, (1, 'a'): 2, (2, 'a'): 2,
              (2, 't'): 3, (3, 'o'): 4, (4, 'o'): 4}
debut_0 = 0
fin_0 = 4
```

Notez que dans ce cas, les clés du dictionnaires sont des tuples, et le valeurs des entiers [1].
L'état initial est le 0, le final 4.

1) Écrire les dictionnaires `automate_1`, `automate_2` et `automate_3` représentant respectivement les automates suivants. On identifiera aussi les états de départ et de fin dans des variables `debut_1`, `fin_1`, `debut_2`, `fin_2`, `debut_3`, `fin_3`, :

1. En réalité, les valeurs peuvent être de n'importe quel type. La seule condition est que le premier élément de chaque tuple dans les clés soit du même type que les valeurs.



2) L'objectif est maintenant de programmer une fonction qui reçoit une chaine de caractères et un automate en paramètres, et vérifie si la chaine de caractères correspond à l'automate. Plus précisément, écrivez une fonction `correspond(chaine, regles, debut, fin)`, qui

- prend en prend en paramètres :
  - une chaine de caractères `chaine` ;
  - un automate, dans le dictionnaire `regles` ;
  - l'état de départ `debut` ;
  - l'état final `fin`.
- renvoie `True` si la chaine satisfait au règles de l'automate et `False` sinon.

Nous allons maintenant tester que la fonction programmée fonctionne correctement. Pour ce faire, nous allons la tester sur les différents automates que nous avons programmés, avec des chaines pour laquelles nous connaissons le résultat attendu.

Le langage Python fournit pour cela le mot-clé `assert`. On sait par exemple que les chaines de caractères `'bato'` et `'baato'` sont compatibles avec l'automate 0 représenté par le dictionnaire `automate\_0`, mais que la chaine `'bota'` ne l'est pas. On peut tester le bon fonctionnemet de la fonction `correspond` avec le code suivant :

automates.py

```
assert correspond('bato', automate_0, debut_0, fin_0) is True
assert correspond('baato', automate_0, debut_0, fin_0) is True
assert correspond('bota', automate_0, debut_0, fin_0) is False
```

Si le code s'exécute sans erreurs, c'est que les tests ont été passés avec succès.

3) En utilisant le mot-clé `assert`, réaliser les tests pour tous les automates, permettant de vérifier que la fonction `correspond` ne contient pas de bugs.