Introduction à la programmation, cours 4

I. Les dictionnaires

Un **dictionnaire** est une **structure de donnée** qui, comme la liste, permet de sauvegarder des données en mémoire. Contrairement à une liste, les valeurs d'un dictionnaire ne sont pas stockés dans un ordre particulier.

Création et utilisation d'un dictionnaire

Déclarer un dictionnaire :

```
D = {cle1: valeur1,  cle2: valeur2, … ,cleN: valeurN}
```

Un exemple :

```
note = {"Tatiana": 8.5, "Pierre": 17.5}
```

La variable **note** est un dictionnaire contenant les notes de deux étudiants. Les chaînes de caractères "Tatiana" et "Pierre" sont les **clés** du dictionnaire. Les float 8.5 et 17.5 sont les **valeurs** du dictionnaire. Dans un dictionnaire, à chaque clé est associés une valeur (dans un dictionnaire au sens classique, les clés sont les mots et les valeurs sont les définitions).
Attention, les éléments d'un dictionnaire **ne sont pas ordonnés** (i.e. il n'y a pas de premier ni de deuxième élément).

L'accès à une **valeur** du dictionnaire se fait non par sa position (il n'en a pas), mais grâce à sa **clé** :

```
note = {"Tatiana" : 8.5, "Pierre" : 17.5}
print note["Tatiana"]
```

Dans l'exemple ci-dessus, les **clés** sont de types **string**, et les **valeurs** de type **int**. Notons que les **valeurs** d'un dictionnaire peuvent être de n'importe quel type (y compris de type dictionnaire), contrairement aux **clés** qui ne peuvent pas prendre n'importe quel type. Dans ce cours, les **clés** seront de type **int** ou **string**.

Ajouter/supprimer/modifier une entrée dans un dictionnaire

Pour **ajouter** une nouvelle entrée (= clé + valeur), il suffit d'utiliser l'opérateur = comme suit :

```
D = {}  #création d'un dictionnaire vide
print D
D["a"] = 1        #ajout de la nouvel entrée, clé = "a", valeur associée = 1
print D
```

Si la clé existe déjà elle prend une nouvelle valeur :

```
D = {}
print D
D["a"] = 1        #On ajoute une entrée ("a " : 1)
print D
D["a"] = 3        #l'entrée est modifiée ("a " : 3)
print D
```

Pour **supprimer** une entrée, il faut utiliser l'opérateur **del** :

```
D = {"a" : 1, "b" : 2, "c" : 3}
print D
del D["a"]        #on supprime l'entrée ("a" : 1)
print D
```

Pour **modifier** une entrée, rien de plus simple:

```
D = {"a" : 1, "b" : 2, "c" : 3}
```

```
print D
D["a"] = 7          #la valeur associée à la clé "a" est maintenant 7
print D
D["b"] = D["b"] + 3       #la valeur associée à l'entrée "b" est maintenant 5
```

Vérifier l'existence d'une entrée dans un dictionnaire

Pour vérifier si une clé donnée apparaît dans un dictionnaire, on utilise l'opérateur **in** comme dans le cas des liste.

```
D = {"a" : 1, "b" : 2, "c" : 3}
print "b" in D    #observer que l'expression "b" in D est un booléen
print "d" in D
```

Si on tente d'accéder à une entrée qui n'existe pas, le programme renvoie une erreur de clé (KeyError):

```
lettres = {"a" : 103, "b" : 8, "e" : 150}
print lettres["d"]
```

Ainsi, avant d'accéder à une valeur, on prendra toujours soin de vérifier que la clé associée existe :

```
lettres = {"a" : 1, "b" : 2, "c" : 3}
if "c" in lettres :
        lettres["c"] = lettres["c"] + 1      #on incrémente la valeur associé à "c" de 1 dans la cas où "c" est
                                             #une clé de notre dictionnaire
else :
        lettres["c"] = 1                     #sinon on crée l'entrée "c" : 1
```

Attention, l'opérateur **in** vérifie l'existence d'une **clé** et non pas d'une **valeur** :

```
D = {"a" : 1, "b" : 2, "c" : 3}
print 1 in D       #renvoie une erreur
```

Parcourir un dictionnaire

La boucle **for** peut être utilisée pour parcourir toutes les clés d'un dictionnaire :

```
date_naissance = {"Pierre" : [11, 08, 1988], "Tatiana" : [01, 05, 1986], "Pedro" : [08, 12, 1992]}
        #Notez qu'ici les valeurs sont des listes
for nom in date_naissance :       #nom parcours les clés du dictionnaire
        date = dates_naissance[nom]    #la variable date contient la valeur associé à la clé nom (ici une liste)
        print nom, "fetera son anniversaire le ", date[0], " / ", date[1], " / ", date[2]
```

On peut aussi parcourir les entrées d'un dictionnaire à l'aide de la méthode **items()** :

```
date_naissance = {"Pierre" : [11, 08, 1988], "Tatiana" : [01, 05, 1986], "Pedro" : [08, 12, 1992]}
for nom, date in date_naissance.items() :#nom parcours les clés et date les valeurs associées
        print nom "fetera son anniversaire le ", date[0], " / ", date[1], " / ", date[2]
```

Copier un dictionnaire

L'affectation d'un dictionnaire vers une variable ne fait que référencer le même dictionnaire :

```
D = {"a" : 1, "b" : 2, "c" : 3}
F = D              #F est une variable qui pointe vers le dictionnaire D
F["a"] = 50        #cela modifie aussi D
print D
```

Pour créer une copie d'un dictionnaire, on utilise la fonction **dict()** :

```
D = {"a" : 1, "b" : 2, "c" : 3}
F = dict(D)        #F est une copie de D
F["a"] = 50        #cela ne modifie pas D
print F
print D
```

II. Lecture/écriture dans un fichier

Jusqu'à présent, nous avons utilisé input() et print pour lire des données (données rentrées par l'utilisateur) et afficher des résultats. Mais parfois les données sont stockés dans un fichier et on aimerait pouvoir y accéder sans devoir les saisir manuellement au clavier. De même, il est souvent utile de sauvegarder des données dans un fichier. Un exemple typique est lorsque vous jouez à un jeu vidéo, vous pouvez sauvegarder votre partie (qui est une donnée) et la récupérer quand vous voulez.

Pour ouvrir un fichier, on utilise la fonction **open()**. Elle prends deux paramètres :

- le chemin menant au fichier à ouvrir (nous on écrit juste `"fichier.txt"`, cf l'exemple, si vous voulez ouvrir un fichier qui n'est pas dans le même dossier que le fichier python sur lequel vous codez c'est un poil plus compliqué.)
- le mode d'ouverture qui est une chaîne de caractères, voici les trois principaux modes :
  - `"r"` : ouverture en mode lecture (Read),
  - `"w"` :ouverture en mode écriture (Write). Le contenu du fichier est **écrasé**. Si le fichier n'existe pas, il est créé,
  - `"a"` : ouverture en mode ajout (Append). Comme le mode écriture mais **sans effacer** le contenu du fichier.txt s'il existe déjà.

Ecrire dans un fichier

Pour le premier exemple, on va créer un fichier **exemple.txt** et écrire dedans à l'aide de la fonction **write()** :

```
f = open("exemple.txt","a")
f.write("Le coeur a ses raisons\nQue la raison ignore.")
f.close()
```

Exécutez ce programme (qu'une seule fois). Allez voir dans le dossier où se trouve le fichier.py sur lequel vous travaillez et vérifiez qu'un fichier *exemple.txt* est bien apparu. Ouvrez-le voir ce qu'il contient. Il devrait contenir :
Le coeur à ses raisons
Que la raison ignore.
Maintenant décortiquons les trois lignes de codes.
Dans la première ligne, on **ouvre** (ici on **crée** puisque le fichier n'existait pas) un fichier texte nommé *exemple.txt* en mode **Append**, c'est à dire qu'on va pouvoir écrire dessus. Le fichier est accessible depuis la variable **f** (si vous êtes curieux, regardez de quel type est cette variable).
Dans la deuxième ligne, on utilise la méthode **write()** pour écrire dedans. Notez que la chaîne de caractères `\n` permet un retour à la ligne.
Dans la troisième ligne, on referme le fichier avec la méthode **close()**. N'oubliez **JAMAIS** de **fermer** un fichier après l'avoir ouvert. Si d'autres applications, ou d'autres morceaux de votre propre code, souhaitent accéder à ce fichier, ils ne pourront pas si le fichier est ouvert.
Pour finir, notez que si vous exécutez votre code une deuxième fois, *exemple.txt* comportera notre bout de texte deux fois à la suite. Essayez de faire la même chose en mode `"w"`. Cette fois, vous pouvez exécuter votre programme mille fois, la phrase n'apparaîtra toujours qu'une seule fois, car en mode `"w"` le fichier est effacé à chaque fois que `f = open("exemple.txt","a")` est exécutée.

ATTENTION : l'argument passé dans **write()** doit **obligatoirement être une chaîne de caractères**, si vous voulez écrire un nombre, il faut d'abord le transformer en chaîne de caractères avec la fonction **str()**.

Lire un fichier

Pour lire un fichier, on utilise la méthode **read()**, qui permet d'encapsuler le contenu du fichier en une chaîne de caractère :

```
f = open("exemple.txt","r")  #notez qu'ici on l'ouvre en mode lecture, sinon on ne pourra pas utiliser la
                             #méthode read()
contenu = f.read()         #la variable contenu est une chaîne de caractère contenant le texte de exemple.txt
print type(contenu)        #contenu est une chaîne de caractère
print contenu
```

```
f.close()
```

On peut aussi lire le fichier ligne par ligne (une ligne est une chaîne de caractères se terminant par un retour à la ligne) avec la méthode **readlines()** :

```
f = open("exemple.txt","r")
lignes = f.readlines()      #la variable ligne contient une liste dont les éléments sont les lignes de exemple.txt
print lignes
for l in lignes :  #affiche tout le texte.
            print l
for i in range (len(lignes)) :        #une autre façon d'afficher le texte.
            print l[i]
print len(liste)
```

III. Les modules

Les développeurs de Python ont implémenté de nombreuses fonctions utiles, et vous devriez les utiliser. Les fonctions sont regroupées dans des collections appelées «modules». Par exemple, il y a un module *random* qui permet de simuler le hasard. Pour utiliser un module, vous devez l'importer, ce qui consiste à ajouter une petite ligne de code, pour utiliser le module random il faut écrire `from random import*` (sachez qu'il y a d'autres façon de faire). Voyons un exemple avec trois fonctions disponibles dans le module *random* :

```
from random import*
var_1 randint(1, 10)       #renvoie un entier au hasard entre 1 et 10
print var
liste = ["Tatiana", "Pierre", "Serge", "Chien-Chung"]
var_2 = choice(liste)      #renvoie un élément au hasard dans la liste
print var_2
lettres = ["a" , "b", "c", "d", "e"]
shuffle(lettres)              #mélange les éléments de la liste
print(lettres)
```

Exécuter le code plusieurs fois pour bien voir que le résultat est à chaque fois différent.
La liste des modules est accessibles ici : listes des modules. Si vous vouelz par exemples la liste des fonction existante dans le module random : module random.

Exercice : écrire un programme qui simule le lancé de deux dés, fait la somme, et affiche le résultat de la somme.

Exercice lire un fichier : générateur de poèmes

Cent mille milliards de poèmes est une œuvre de poésie combinatoire de Raymond Queneau, publiée en 1961. Le livre est composé de dix pages, chacune séparée en quatorze bandes horizontales, chaque bande portant sur son recto un vers (rien sur le verso). Le lecteur peut donc, en tournant les bandes horizontales comme des pages, choisir pour chaque vers une des dix versions proposées par Queneau. Les dix versions de chaque vers ont la même scansion et la même rime, ce qui assure que chaque sonnet ainsi assemblé est régulier dans sa forme. Télécharger *verses_Queneau.txt* depuis moodle. Vous pouvez aussi regarder le fichier *verses.txt* pour bien comprendre la structure du texte.

L'objectif de cette exercice est d'écrire une fonction générant au hasard un poème de Queneau.

Hint : commencer par ouvrir le fichier, le transformer en liste avec **readlines(),** puis créer une nouvelle liste contenant les 140 vers du texte (il faut donc enlever les "\n"), puis générer le poème (qui va avoir 14 vers) en utilisant la fonction randint (il faut choisir un vers parmi les éléments 0 à 9 de la liste, puis un vers parmi les éléments 10 à 19 etc etc, utiliser une boucle for).

Exercice sur les dictionnaires : gérer un stock de fruit

Dans cet exercice, nous allons gérer un stock de fruits qui sera représenté par un dictionnaire, dont les clés seront les noms de fruits, et les valeurs seront le nombre de fruits correspondant dans le stock. Par exemple, si le stock contient 2 pommes et 6 bananes, il sera représenté par le dictionnaire suivant: `{"pomme" : 2, "banane" : 6}`. Pour simplifier l'écriture des exemples, on supposera que ce dictionnaire sera stocké dans une

variable appelée stock.

Dans tout l'exercice, le(s) dictionnaire(s) passé(s) en argument ne doi(ven)t pas être modifié(s) (i.e. on prendra garde à **commencer la fonction par copier le dictionnaire passé en argument**, et de modifier seulement cette copie dans le corps de la fonction).

NB_1 : faites attention à la différence entre une fonction qui **renvoie** quelque chose et une fonction qui **affiche** quelque chose.

NB_2 : après chaque question, faites des testes pour vérifier que votre question fonctionne bien. Si elle ne fonctionne pas bien et que vous ne comprenez pourquoi, utiliser http://pythontutor.com/visualize.html#mode=edit

1. Ecrire une fonction **ajoute** qui prend en argument un stock (dictionnaire), un nom de fruit et une quantité q, et qui renvoie le nouveau stock, dans lequel on a ajouté une quantité q du type de fruit précisé.
   Exemples:
   - ajoute(stock , 'pomme', 5) renvoie {'pomme' : 7, 'banane' : 6}
   - ajoute(stock , 'poire', 4) renvoie {'pomme' : 2, 'banane' : 6, 'poire' : 4}
2. Ecrire une fonction **enleve** qui prend en argument un stock (dictionnaire), un nom de fruit et une quantité q, et qui renvoie le nouveau stock où l'on a enlevé la quantité q du type de fruit précisé. De même que pour la fonction enleve1, si le stock de ce fruit tombe à zéro, il faut enlever la clé du dictionnaire. Si le stock n'était pas suffisant, le programme affichera "Erreur: quantité insuffisante de *(*nom du fruit*)*" et renverra le stock initial non modifié.
   Exemples:
   - enleve(stock , 'pomme', 2) renvoie {'banane' : 6}
   - enleve(stock , 'banane', 10) affiche "Erreur: Quantité insuffisante de banane" et renvoie {'pomme' : 2, 'banane' : 6}
3. Ecrire une fonction **apres_livraison** qui prend en argument un stock (dictionnaire) ainsi que le contenu de la livraison (représenté aussi par un dictionnaire) et qui renvoie le nouveau stock après la livraison.
   Exemple:
   - apres_livraison(stock , {'peche' : 4, 'pomme' : 5}) renvoie {'pomme' : 7, 'banane' : 6, 'peche' : 4}
4. Ecrire une fonction **commande** qui prend en argument le stock actuel (dictionnaire) ainsi que le stock minimum voulu (dictionnaire aussi) et qui renvoie le dictionnaire correspondant à la commande qu'il faut faire pour obtenir le stock voulu. Si le fruit apparaît déjà en quantité suffisante dans le stock actuel (supérieure ou égal au stock voulu), il ne doit pas apparaître dans la commande.
   Exemple:
   - En supposant que stock_voulu={'pomme': 15, 'orange': 20}, alors commande(stock , stock_voulu) renvoie {'pomme' : 13, 'orange' : 20}.
   - En supposant que stock_voulu={'pomme': 10, 'banane': 4}, alors commande(stock , stock_voulu) renvoie {'pomme' : 8}.
5. Ecrire une fonction **total** qui prend en argument le stock et qui renvoie le nombre total de fruits présents dans le stock (tous types confondus)
   Exemple:
   - total(stock) renvoie 8.
6. Ecrire une fonction **quantite** qui prend en argument le stock ainsi qu'une liste de noms de fruits fruits_a_compter, et qui renvoie la quantité de fruits présents dans le stock dont le nom est dans la liste fruits_a_compter.
   Exemple:
   - En supposant que stock_bis={'pomme': 15, 'peche': 4, 'citron': 3, 'orange': 20}, alors quantite(stock_bis , ['pomme', 'citron', 'poire']) renvoie 18.
7. Ecrire une fonction **quantite_agrumes** qui prend en argument le stock et qui renvoie la quantité d'agrumes présents dans le stock. Seront considérés comme noms d'agrumes: orange, citron, mandarine, clémentine et pamplemousse.
   Exemple:

- En supposant que stock_bis={'pomme': 15, 'peche': 4, 'citron': 3, 'orange': '20'}, alors quantite_agrumes(stock_bis) renvoie 23.