	Université de Corse - Pasquale PAOLI
	Diplôme : **Licence SPI 3ème année** — 2020-2021
	UE : Ateliers de programmation
	Atelier 4 : Chaînes de caractères
	Enseignants : Paul-Antoine BISGAMBIGLIA, Marie-Laure NIVET, Evelyne VITTORI

PARTIE 1 - EXERCICES ESSENTIELS

EXERCICE 1 - Manipulations simples

1. Ecrire une fonction *full_name(str_arg:str)->str* qui prend en paramètre une chaine de caractère de type 'nom prenom' et qui renvoie la même chaîne avec le nom en majuscule et le prénom avec la première lettre seulement en majuscule.

Exemple :
str_variable2test = 'bisgambiglia paul'
full_name(str_variable2test) doit renvoyer 'BISGAMBIGLIA Paul'

2. Ecrire une fonction is*_mail(str_arg:str)->(int,int)* qui prend en paramètre une chaine de caractère de type adresse mail 'bisgambiglia@univ-corse.fr' et qui renvoie un tuple composé des codes suivants : (validité, code erreur)
    a. (1, x) x n'est pas important, le mail est valide
    b. (0, 1) le mail n'est pas valide, le corp n'est pas valide (bisgambiglia)
    c. (0, 2) le mail n'est pas valide, il manque l'@
    d. (0, 3) le mail n'est pas valide, le domaine n'est pas valide (univ-corse)
    e. (0, 4) le mail n'est pas valide, il manque le .

Exemples :
str_variable2test = 'bisgambiglia_paul@univ-corse.fr'
is_mail(str_variable2test) doit renvoyer (1,0)
str_variable2test = 'bisgambiglia_paulOuniv-corse.fr'
is_mail(str_variable2test) doit renvoyer (0,2)
str_variable2test = 'bisgambiglia_paul@univ-corsePOINTfr'
is_mail(str_variable2test) doit renvoyer (0,4)
str_variable2test = '@univ-corse.fr'
is_mail(str_variable2test) doit renvoyer (0,1)

Conseil : regardez les fonctions de manipulation de string.
Cet exercice ne doit pas prendre en compte le cas ou il y a plusieurs erreurs.
Réalisez également le programme de test qui renvoie le type d'erreur sous forme de chaine de caractères.

EXERCICE 2 - Mots croisés

1. Définissez une fonction ***mots_Nlettres***(lst_mot, *n*) qui prend une liste de mots (lst_mot) en argument et qui renvoie la liste des mots contenant exactement *n* lettres.

Définissez progressivement une procédure de test ***test_exercice1*** pour tester chacune de vos fonctions.

Ex de liste de test :
lst_mot=["jouer","bonjour", "punir", "jour", "aurevoir", "revoir", "pouvoir", "cour", "abajour", "finir", "aimer"]

2. Définissez une fonction ***commence_par***(*mot*, *prefixe*) qui renvoie *True* si l'argument *mot* commence par *prefixe* et *False* sinon.

3. Définissez une fonction ***finit_par***(*mot, suffixe*) qui renvoie *True* si l'argument *mot* se termine par *suffixe* et *False* sinon.

4. Définissez une fonction ***finissent_par(****lst_mot,, suffixe****)*** qui renvoie la liste des mots présents dans la liste *lst_mot* qui se terminent par *suffixe*.

5. Définissez une fonction ***commencent_par(****lst_mot, prefixe****)*** qui renvoie la liste des mots présents dans la liste *lst_mot* qui commencent par *prefixe*.

6. Définissez une fonction ***liste_mots (****lst_mot, prefixe, suffixe, n****)*** qui renvoie la liste des mots présents dans lst_mot qui commencent par *prefixe*, se terminent par *suffixe* et contiennent exactement *n* lettres.
Votre fonction ne devra contenir ni boucle for ni structure conditionnelle if.

7. Définissez une fonction ***dictionnaire***(*fichier*) qui admet en paramètre une chaine de caractères représentant un nom de fichier de texte (ex :littre.txt) et renvoie la liste des mots présents dans ce fichier.
   Vous supposerez que le fichier de texte comporte un mot par ligne.
   Testez le fonctionnement de vos fonctions sur le fichier littre (fichier « littre.txt » à récupérer sur l'ENT)
   Remarque: le terme "dictionnaire" fait ici référence à un dictionnaire au sens classique (le livre!). Cela n'a rien à voir avec les conteneurs de type dictionnaire de python.

Annexe : Exemple de *lecture d'un fichier de texte en python*

```
#Exemple d'affichage du contenu d'un fichier de noms (profs.txt) comportant un nom par ligne)
# ouverture du fichier en lecture (r=read)
f=open("profs.txt","r")
c=f.readline() #lecture d'une ligne dans une chaine
# de caractères
print("** Contenu du fichier **")
while c!="" :
    print(c)
    c=f.readline()
print("** fin **")
```

```
** Contenu du fichier **
Simonnet
Nivet
Vittori
Poggi
Cagnard
** fin **
```

PARTIE 2 - EXERCICES APPROFONDIS

EXERCICE 3 - Jeu du pendu

L'objectif de cet exercice est de coder le jeu du pendu.
Il est basé sur les notions suivantes :

- les structures conditionnelles
- les structures itératives
- les chaînes de caractères
- la gestion (lecture/écriture) des fichiers

Question 1. Début du jeu

Ecrivez une fonction qui doit demander à l'utilisateur une lettre (un caractère) et vérifier si elle est présente dans le mot (déjà réalisé dans l'atelier 2).
Voici le prototype de la fonction ***placesLettre(char : str, mot : str) -> list***
La fonction doit renvoyer une liste vide si le caractère n'est pas présent et sinon le ou les indices désignants sa place dans le mot.

Exemple :

- `placesLettre('b','bonjour')-> [0]`
- `placesLettre('a','bonjour')-> []`
- `placesLettre('m','maman')-> [0,2]`

Question 2. Affichage

Ecrivez une fonction qui prend en paramètre un mot ***outputStr(mot : str)-> str*** et qui renvoie une chaîne de caractères qui cache les caractères du mot.

Exemple :

- outputStr('bonjour')-> '_ _ _ _ _ _ _'
- outputStr('bon')-> '_ _ _'
- outputStr('maman')-> '_ _ _ _ _'

Question 3. Programme principal

Ecrivez une fonction ***runGame()*** qui sera votre programme principal, il doit :

- contenir un ensemble de mot, sous forme de dict (dictionnaire, map) dSetWords {key : (mot,taille du mot)}
- tirer aléatoirement un mot de l'ensemble
- proposer à l'utilisateur de retrouver le mot en 5 coups, utiliser pour cela la fonction placesLettre()
- afficher le nombre de coup restant
- afficher les sorties C1, C2, C3, celles qui permettent de dessiner le pendu, pour chaque erreur
- afficher le nombre de caractère du mot sous la forme : paris = _ _ _ _ _ , utiliser la fonction outputStr()

```
#### demander a l'utilisateur de saisir un caractère
cUserChar = input("Entrer une lettre: ")
### à placer dns un bloc try catch

### exemple de dictionnaire (map : structure de type clé , valeur)
```

```
dSetWords = {
  1: ("paris",5),
  2: ("londre",6),
  3: ("madrid",6),
  4: ("new-york",8)
  5: ("berlin",6)
}
```

code qui permet de tirer aléatoirement un entier entre 1 et iSizeDict

```
import random
iSizeDict = len(dSetWords)
iRand = random.randint(1, iSizeDict)
```

exemple de sorties (vous pouvez améliorer)

```
C5 = "|---] "
C4 = "|  O "
C3 = |  T "
C2 = "|/ \ "
C1 = "|_____"
```

Question 4. Gestion des fichiers

Erivez une fonction ***buildDict(fileName : str) -> Dict*** qui prend en paramètre un nom de fichier et construit automatiquement le dictionnaire dSetWords.
Pour cela créez un fichier manuellement contenant les capitales du monde source https://fr.wikipedia.org/wiki/Liste_des_capitales_du_monde (https://fr.wikipedia.org/wiki/Liste_des_capitales_du_monde)

Attention, il va falloir appliquer des traitements :

- str.split("\t") permet de couper une chaîne de caractère à chaque tabulation
- str.lower() permet de mettre une lettre en minuscule

```
#### lire un fichier
file = open("capitales.txt", "r")
content = file.readlines()
file.close()
```

Modifiez votre programme principal pour utiliser cette nouvelle fonction.

Question 5. Niveau de difficulté

Proposez à l'utilisateur un niveau de difficulté, pour cela il est possible de trier le dictionnaire par la taille des mots :

- niveau ‘easy’ taille du mot < 7
- niveau ‘normal’ 6 < taille du mot < 9
- niveau ‘hard’ taille du mot > 8

Il faut adapter le programme principal pour prendre en compte cette évolution.

PARTIE 3 – EXERCICES BONUS

EXERCICE 4 - Aide scrabble

1. Définissez une fonction ***mot_correspond(****mot, motif****)*** qui renvoie *True* ou *False* suivant que la chaine de caractère *mot* correspond, ou pas, à la chaine de caractères *motif* donnée.

Un *motif* est un chaîne de caractères pouvant contenir des « jokers », représentés par le caractère ".", pouvant remplacer n'importe quelle lettre. Ainsi, on aura :

```
>>> mot_correspond("tarte", "t..t.")
True
>>> mot_correspond("cheval", "c..v..l")
False
>>> mot_correspond("cheval", "c..v.l")
True
```

2. Définissez une fonction ***presente(****lettre, mot****)*** qui renvoie un entier représentant l'indice de la lettre passée en paramètre dans la chaine de caractères *mot*. La fonction renvoie -1 si la lettre n'est pas trouvée.
3. Définissez une fonction ***mot_possible(****mot, lettres****)*** qui renvoie *True* ou *False* suivant que le mot peut s'obtenir avec les *lettres* passées en paramètres. Ces lettres sont fournies à la fonction sous la forme d'une chaîne de caractères. Par exemple :

```
>>> mot_possible("lapin", "abilnpq")
True
>>> mot_possible("cheval", "abilnpq")
False
```

Contrainte: votre fonction devra appeler la fonction presente de la question 2 pour tester la présence de chaque lettre du mot dans la liste lettres.

Au Scrabble, les joueurs cherchent à former des mots avec un ensemble de lettres données. Ces lettres sont inscrites sur des jetons. Au début et tout au long de la partie, les joueurs piochent des jetons. Chacun dispose ainsi d'une "main" composée de la liste de toutes les lettres inscrites sur les jetons qu'il a piochés. Nous pourrions envisager d'avoir recours aux fonctions précédentes pour aider un joueur à construire des mots à partir des lettres dont il dispose dans sa "main". Vérifiez que votre fonction renvoie bien les valeurs correctes dans ces exemples :

```
>>> mot_possible ("chapeau", "abcehpuv")      False
>>> mot_possible ("chapeau", "abcehpuva")   True
```

Au lieu de simplement regarder si une lettre apparaît dans le mot, il faut en effet *compter* le nombre de fois où cette lettre apparaît. Si votre fonction mot_possible ne renvoie pas un résultat correct, modifiez la en conséquence.

4. Définissez la fonction ***mot_optimaux(****dico, lettres****)*** qui renvoie la liste des mots de longueur maximale présents dans la liste *dico* que l'on peut faire avec les lettres passées en paramètre dans la chaine de caractères *lettres*.

Utilisez la fonction *mots_Nlettres* de l'exercice 1 pour générer successivement des listes de mots d'une longueur donnée en commençant par la longueur maximum (nombre de lettres présentes dans la chaine *lettres*)

Testez en utilisant la fonction dictionnaire de l'exercice 1 pour créer la liste *dico* à partir du fichier littre.txt.

EXERCICE 5 - Vérification d'expressions arithmétiques

1) Définissez les fonctions suivantes :

- fonction ***ouvrante*** (*car*) admettant un paramètre *car* de type caractère et renvoyant un booléen indiquant si le caractère est une parenthèse, un crochet ou une accolade ouvrante
- fonction ***fermante*** (*car*) admettant un paramètre *car* de type caractère et renvoyant un booléen indiquant si le caractère est une parenthèse, un crochet ou une accolade fermante
- fonction ***renverse*** (*car*) admettant un paramètre *car* de type caractère et renvoyant un caractère. Selon que le paramètre soit une parenthèse, un crochet ou une accolade fermante, la fonction renvoie une parenthèse, un crochet ou une accolade ouvrante. Si le paramètre est un nombre, un espace, un opérateur ou une parenthèse,accolade ou crochet fermant, la fonction renvoie le caractère lui-même.
- fonction ***operateur*** (*car*) admettant un paramètre de type caractère et renvoyant un booléen indiquant si le caractère est un opérateur (+ ou *)
- fonction ***nombre*** (*car*) admettant un paramètre *car* de type chaîne de caractère et renvoyant un booléen indiquant si la chaine est un nombre c'est-à-dire qu'elle ne comporte que des caractères numériques (vous pourrez utiliser la méthode python *isdigit*)
- fonction ***caractere_valide***(*car*) admettant un paramètre *car* de type caractère et renvoyant vrai si le caractères est un caractère valide dans une expression arithmétique: parenthèse, crochet, accolade, chiffre, opérateur ou espace

2) Définissez une fonction ***verif_parenthese(expression)*** qui admet un paramètre *expression* de type chaîne de caractère et renvoie un booléen indiquant si l'expression arithmétique contenue dans le texte est valide (ne comportant que des caractères valides) et correctement parenthésée ou non.

   Le principe de l'algorithme à implémenter est le suivant :

   - On initialise une liste vide P (la "pile")
   - On parcourt le texte de gauche à droite. **On empile** chaque parenthèse, crochet ou accolade **ouvrante** rencontrée (méthode *append*).
   - A chaque parenthèse, crochet ou accolade fermante rencontrée, **si le caractère situé au sommet de la pile P (caractère situé dans la dernière case de P) est bien la parenthèse, crochet ou accolade ouvrante** associée à la parenthèse fermante considérée, on dépile (méthode *pop*).

   Avant de programmer la méthode, faites fonctionner l'algorithme à la main sur quelques exemples afin d'identifier notamment les conditions d'arrêt :

   (3+2) * 6-1
   ((3+2)*6-1
   (5+7]*12

Indications: Les listes python constitue une implémentation de la structure de données Pile
- création d'une pile p=[]
- test d'une pile vide: p est vide si len(p)==0
- sommet de p = p[-1]
- empiler un élément a: p.append(a)
- depiler: p.pop()

Annexes :

Exemple de manipulation de chaine de caractère

- *concaténation de chaînes : x = 'aaa' + 'bbb' donne aaabbb.*
- *x = 'abcdef' : définition de la chaîne.*
- *Les chaînes sont read-only (non mutables), donc on ne peut pas faire x[1] = 'x'*
- *print(x[2]) : 3ème caractère (indice commence à 0), ici c.*
- *print(x[0:3]) : caractères d'indices 0 à 3 - 1, ici abc.*
- *print(x[1:]) : caractères à partir de l'indice 1, ici bcdef.*
- *print(x[:3]) : caractères jusqu'à l'indice 3 - 1, ici abc.*
- *print(x[-2:]) : les 2 derniers caractères, ici ef.*
- *print(len(x)) : longueur, ici 6.*
- *'b' in x : renvoie True si b est un caractère de x.*
- *x.find('ab') : retourne le plus petit index correspondant à la chaîne "ab" (-1 si pas trouvé).*

Rappel des bonnes pratiques

- nommage des variables : la simple lecture du nom de la variable doit nous permettre de savoir
  - si possible quel type de valeur elle accueille, par exemple : int_max_list, str_user_choice
  - et sa fonction dans le programme, c'est à dire à quoi elle sert.
- annotations de types dans les en-têtes des fonctions :
  - cela augmente la lisibilité de votre code, sans pour autant en contraindre l'utilisation
  - cela vous permet de mener en amont une réflexion sur les types d'objets que vous manipulez dans vos codes
  - rappelez-vous que cela n'est pas vérifié par l'interpréteur, c'est juste une recommandation faîte au développeur pour l'invocation de la fonction, mais du coup, réfléchissez et n'écrivez pas n'importe quoi !
- ajout systématique d'une DocString : permet lors de l'affichage de l'aide sur la fonction une fois interprétée via help(maFonction) ou maFonction.__doc__ de connaitre le mode d'emploi de ladite fonction (voir PEP 256).
  - on préfèrera écrire les DocString en Anglais
  - la première phrase décrit précisément ce à quoi sert la fonction
  - on insert une section input : dans laquelle on décrit précisément chacune des entrées en spécifiant son type et son utilité
  - on insert une section ouput : dans laquelle on décrit précisément le retour de la fonction en spécifiant son type
- pour les fonctions :
  - on choisira un nom de fonction parlant
  - on évite les multiples return dans la fonction :
    - on déclare au début de la fonction, une variable type_result qui est évaluée à la valeur de retour de la fonction par défaut.
    - dans le code on value cette variable type_result
    - à la fin de la fonction on effectue un seul return de la variable type_result.