TP5

April 10, 2021

Objectifs

Cette série regroupe les fonctions, les conteneurs et les boucles en Python. Bien que l'objectif principale est de comprendre la manipulation des fonctions, elle présente aussi un complément d'objectifs de la série de TP4. Elle traite des problèmes qui demandent des reflexions de la part des étudiants, mais aussi, demandent la bonne compréhension des conteneurs et des boucles.

Exercice 1

L'objectif de cet exercice est d'utiliser les techniques et les méthodes associées aux listes dans l'algorithme de tri par sélection. Alors, définissez la fonction **triSelect** qui permet de mettre en oeuvre l'algorithme de tri par sélection tout en exploitant la technique de slicing (tranche), la méthode index et la fonction min(). Votre solution doit avoir une seule boucle.

Exercice 2

L'objectif de cet exercice est d'étudier la situation de deux segments de droite. Dans notre, cas un segment est constituer de points apparents et de points discrets. Un point apparent est un point dont ces coordonnées sont de type entier. Les autres point du segment son des points discrets. Un segment est, bien sûr, une portion d'une droite d'équation **y=ax+b**. Dans notre cas les coéfficients de la droite, c'est à dire, 'a' et 'b' sont de type entier.

1. Définissez la fonction **segment** qui renvoie un dictionnaire ayant tois couples (clé, valeur) : ('a', la veleur de a), ('b', la vaelur de b) et ('interval', l'interval de point où varie x). Par exemple, pour a=2, b=3 et l'interval de x est [1,3], le dictionnaire sera {'a':2, 'b': 3, 'interval':(1,3)}. Attention, a et b sont des entiers et x un tuple regroupant deux valeurs entières exprimant les deux extrimités de l'interval fermé. La fonction **segment** admettra 4 paramètres, a, b et les limites de l'interval : extr1 et extr2.

2. Définissez une fonction **lambda** nommée **f** qui implémente la fonction d'une droite **y=ax+b**. Cette fonction admettra comme paramètre un dictionnaire ayant le forme de celui renvoyé par la fonction *segment*. La fonction **f** renvoie une liste de couple de valeurs entières. Cette liste représente les coordonnées (abscisse, ordonnée) des points apparents.

3. Définissez une fonction **parallel** qui renvoie True si deux segments sont parallèles sinon renvoie False. Elle admet deux paramètres de type dictionnaire ayant chacun le forme de celui renvoyé par la fonction *segment*. Rappelons que deux droites sont parallèles si les valeurs absolues de leurs coefficients |a| sont égaux. Pour avoir la valeur absolue d'un nombre vous pouvez utilisez la fonction prédéfinie **abs()**.

4. Deux segments parallèles peuvent être totallement ou partiellement confondus. Définissez, alors, une fonction **PourConf** qui renvoie le pourcentage en terme de nombre de points apparents communs entre deux segments. La formule est : nombre de points apparents

communs divisé par la somme des nombres de points apparents des deux segments. Sinon, elle renvoie 0.

5. Si les deux segment ne sont ni parallèles ni confondue, alors ils pevent se couper en un point. Définissez la fonction **InterPointApparent** qui renvoie le point apparent d'intersection entre deux segments s'il existe, sinon renvoie 0.

6. Le point d'intersection entre deux segments, dans notre cas, peut être un point discret. Nous souhaitons chercher les points apparents de chaque segements les plus proches au point d'intersection discret, bien sûr s'il existe. Pour calculer l'intesection entre deux segments [A,B] et [C,D] il faut :

   A. Calculez le déterminant $\mathbf{det = (xB - xA)(yC - yD) - (xC - xD)(yB - yA)}$

   B. Calculez $\mathbf{t1 = ((xC - xA)(yC - yD) - (xC - xD)(yC - yA))/det}$

   C. Calculez $\mathbf{t2 = ((xB - xA)(yC - yA) - (xC - xA)(yB - yA))/det}$

   D. Les deux segments se coupent ssi t1 et t2 appartiennent simultanément à [0, 1].

   Nous nous inspirerons de cette méthode afin de déterminer les points apparents de chaque segments les plus proches au point discret d'intersection. Pour ce faire, nous allons fixer les deux extrimés d'un segment et parcourir les points apparents de l'autre segment deux à deux. Si par exemple, notre liste de points apparent est la suivante [(-3, -7), (-2, -4), (-1, -1), (0, 2), (1, 5), (2, 8), (3, 11)] alors le parcours se fait de cette manière : (-3, -7), (-2, -4), ensuite, (-2, -4), (-1, -1), ensuite, (-1, -1), (0, 2), ensuite, (0, 2), (1, 5), ensuite, (1, 5), (2, 8), enfin, (2, 8), (3, 11). A chaque itération il faut calculer t1 et t2. Une fois t1 et t2 appartiennent simultanément à [0, 1] enregistrer les deux points apparentes, quitter le parcours du segment et refaire le même processus avec l'autre segment. Attention, il ne faut pas perdre du temps avec le deuxième segment si vous avez parcouru le premier sans trouvez de valeurs pour t1 et t2 qui appartiennent simultanément à [0, 1]. Définissez la fonction **interPointDisc** qui renvoie les points apparents les plus proches au point d'intersection discret entre deux segments, s'il existe. Sinon, elle renvoie 0.

7. Pour tester votre solution, exécutez les lignes de codes suivantes.

[ ]:
```
def etude2Segments(seg1,seg2):
    if parallel(seg1,seg2):
        temp=confondus(seg1,seg2)
        if temp==0:
            print("Les deux segment sont strictement parallèles")
        else:
            print("Les deux segment sont confondus de {}%".format(temp))
    else :
        temp=interPointApparent(seg1,seg2)
        if temp!=0:
            print("L'intersection des deux segments est :",temp)
        else:
            temp=interPointDisc(seg1,seg2)
            if temp!=0:
```

```
                print("L'intersection des deux segments se trouve entre les␣
↪points :",temp)
            else:
                print("Les deux segments ne se coupent pas et ne sont pas␣
↪parallèle")
```

[ ]:
```
seg1=segment(3,2,-3,14)
seg2=segment(-2,1,-5,14)
seg3=segment(3,2,0,4)
seg4=segment(1,2,-3,14)
```

[ ]:
```
etude2Segments(seg1,seg3)
```

[ ]:
```
etude2Segments(seg1,seg2)
```

[ ]:
```
etude2Segments(seg3,seg2)
```

[ ]:
```
etude2Segments(seg1,seg4)
```

Exercice 3

Dans cet exercice nous souhaitons mettre en place un correcteur de ponctuation de la langue française. Pour ce faire, nous commencerons par étendre les méthodes sur les chaines de caractères par une fonction **findAll**. Ensuite, nous définisserons deux fonctions. La première consiste à la vérification de la ponstuation d'un texte rédigé en français et la deuxième localisera les erreurs. Enfin, nous définisserons notre fonction permettant la correction automatique des erreurs de ponctuation.

1. Pour les chaines de caractères nous distinguons les méthodes **find** et **index** qui renvoient la position de la première occurence d'une chaine de caractères dans un texte, bien sûr, si elle existe. Définissez, alors, en Python la fonction **findAll** qui permet de renvoyer toutes les positions d'une chaine de caractère dans un texte, sinon renvoie "-1".
2. Redéfinissez la fonction **findAll** en ajoutant deux paramètres obtionnels **start** par défaut égale à 0 et **stop** par défaut égale à l'indice du dernier caractère du texte.
3. Définissez une fonction **isCorCapPonc**, tout simplement elle permet de vérifier si un texte respecte les règles suivantes:
    1. La première lettre du texte est en majuscule.
    2. On ne met pas d'espace avant les symboles de ponctuations ".", ",", "!" et "?"
    3. On doit mettre une espace après les symboles de ponctuations ".", ",", "!", "?", ":" et ";"
    4. La lettre juste après les symboles de ponctuations ".", "!" et "?" doit être en majuscule.
    5. On doit mettre une espace avant ":" et ";"
    6. La lettre juste après les symboles de ponctuations ",", ":" et ";" doit être en minuscule.

Remarque :Vous pouvez utiliser les méthode str.isupper() (vrai si le caractère ou la chaine est en majuscule sinon faux) et str.islower() (vrai si le caractère ou la chaine est en minuscule sinon faux).

4. Définissez une fonction **locErreur** qui renvoie un dictionnaire de clé le rang de la régle (A,B,C,D,E ou F) et de valeur la liste des positions où la régle n'est pas respectées dans le texte. Si la règle est respecté pour tout le texte alors la valeur sera "Vérifiée".

Attention : la détection de certaines erreurs ne pourra avoir lieu qu'après la correction des autres. Aussi, les listes des positions des erreurs doivent être triées dans l'ordre croissant (vous pouvez utiliser la méthode liste.sort() qui permet de trier les éléments d'une liste) afin de garantir le bon déroulement du processus de correction automatique.

5. Définissez une fonction **CorCapPonc** renvoie le texte corrigé, c'est à dire, respectant les règles ci-dessus.

6. Pour tester vos solutions, exécutez les lignes de codes suivantes :

[ ]:
```
ch="nous souhaitons tester ce paragraphe , pourcela , Nous avons commis␣
 ↪plusieurs fauts de ponctuations .voyons alors notre correcteur est il␣
 ↪capable de les corrigées ?bien sûr au niveau de détection:Il détecte␣
 ↪certaines erreurs et d'autres aprés la corrections."
print(ch)
print("-----------------------------------------------")
print(isCorCapPonc(ch))
print("-----------------------------------------------")
erreurs=locErreur(ch)
print(erreurs)
print("-----------------------------------------------")
ch_Corrigee=CorCapPonc(ch,erreurs)
print(ch_Corrigee)
print("-----------------------------------------------")
print(isCorCapPonc(ch_Corrigee))
```