BE5 – INF
LE JEU DU PENDU

Rapport de Bureau d'Études

UE Informatique - INF tc2

Groupe

MIRANDA GOMES Vitor

GUEYE Khady

Chargé de BE :

CHALON René

1. Introduction

L'objectif de ce BE est de réaliser une version du jeu du Pendu en utilisant le module de python pour interfaces graphiques : TKinter.

Ce jeu consiste à essayer de découvrir un mot caché (chacune de ses lettres est remplacée par le caractère '*') avec un nombre d'essais limités (dans notre cas, 10). Pour cela, le joueur sélectionne une lettre sur le clavier virtuel, si la lettre appartient au mot à découvrir elle s'affiche en lieu du '*', sinon les formes d'un pendu s'affichent à chaque erreur, si le joueur dépasse le seuil d'erreurs le dessin du pendu est complet et il perd la partie.

Tout d'abord, nous montrerons la démarche globale des parties du code faites en autonomie, c'est-à-dire, la partie quatre de l'énoncé et le diagramme de classes concernant la solution du problème traité. À chaque section nous présentons son implémentation.

Le code est joint à ce rapport et le fichier principal est StartPage. Un exécutable devait être mis à votre disposition pour faciliter les tests mais nous ne l'avons inclus dans le fichier compressé car la taille dépasserait la limite permise sur moodle.

2. Démarche globale du code

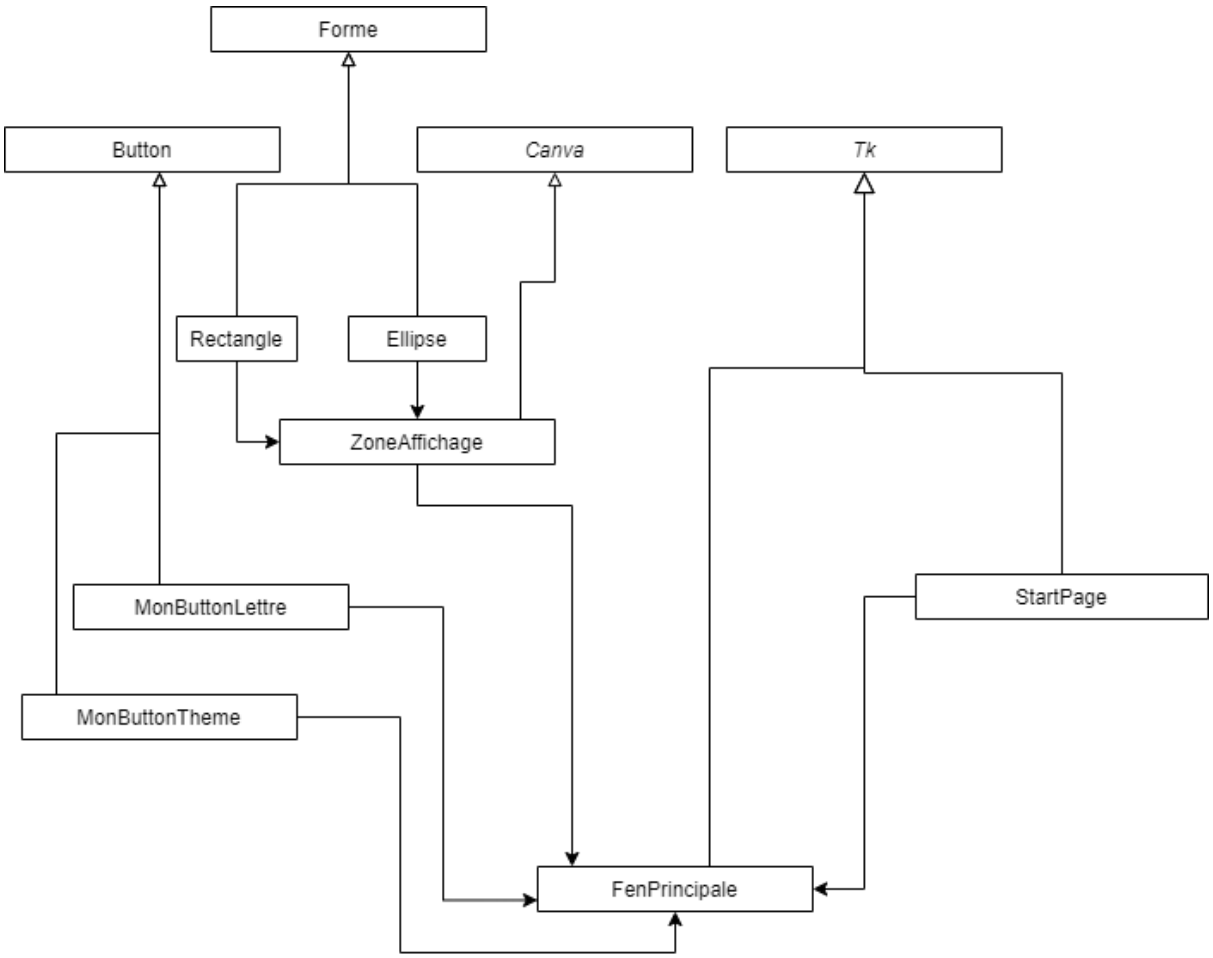
La page principale permet à l'utilisateur de gérer sa liste de joueurs (créer, supprimer, choisir pour jouer). Les joueurs créés sont enregistrés dans une base de données (pendu.db) et une fois un joueur choisi, une partie sera créée pour ce dernier, ce qui permet de sauvegarder l'historique de ses victoires et défaites.

Dans la fenêtre principale (après choix du joueur et début du jeu), à chaque clic le programme vérifie si la lettre appartient ou pas au mot, et en cas d'une victoire, nous ajoutons un point dans le score et on l'enregistre dans l'historique. L'utilisateur peut alors voir son nom, son score, et son historique.

Comme demandé dans l'exercice 8, il a aussi la possibilité de « tricher », retourner en arrière en cas d'erreur.

3. Diagramme de classes

Le diagramme de classes se présente comme suit, les détails seront expliqués dans les sections suivantes.



4. Partie quatre de l'énoncé

4.1. Question 7 - Apparence

Dans cette partie on a développé un code qui permet au joueur de changer le thème de l'application par un menu avec des buttons. Nous voyons la fenêtre créée dans la *figure 1*.

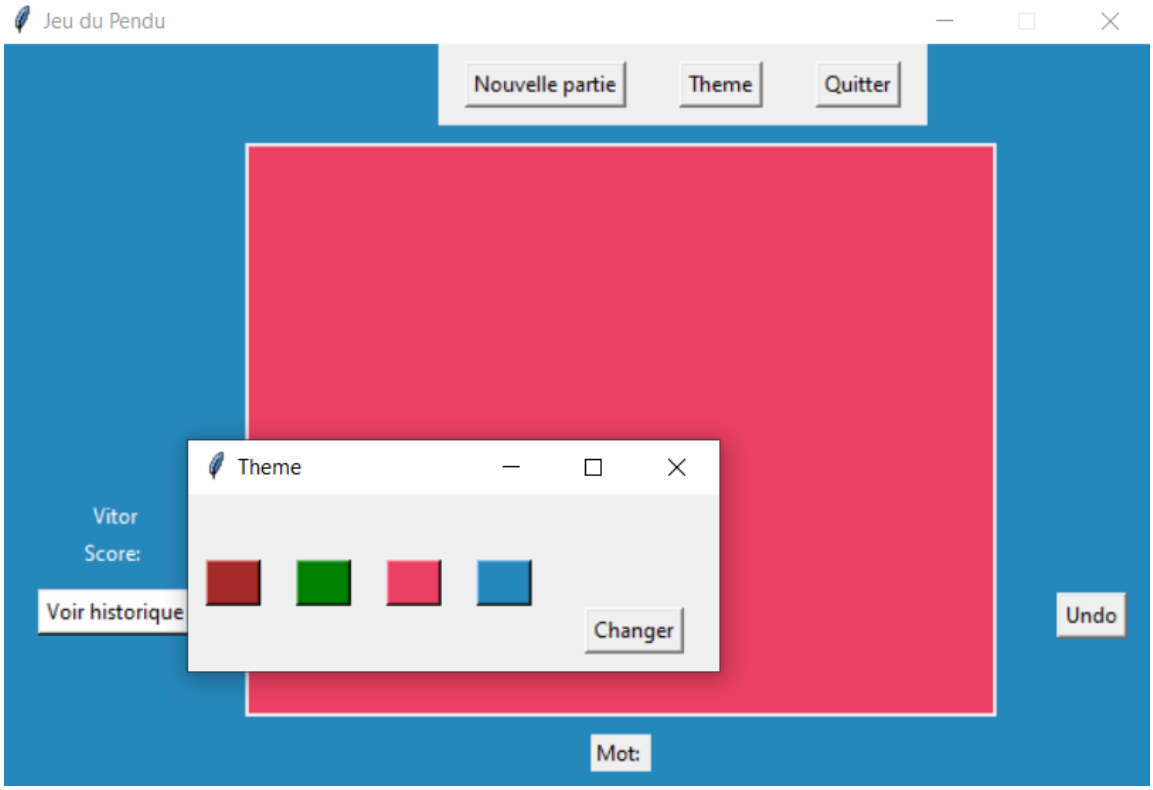


Figure 1 - La fenêtre Thème et ses options

Pour l'implémenter, nous avons défini une méthode *choixTheme (figure 2)* dans le programme principal où l'on va créer une fenêtre popup avec différentes options.

Ensuite, pour récupérer la couleur du bouton sur lequel l'utilisateur aura cliqué, nous avons créé une classe *MonBoutonTheme (figure 3)*, qui peut passer un argument (couleur) à la méthode passée en paramètre (colorSet()) et permet donc de changer la couleur en fonction du bouton sur lequel on aura appuyé.

```
121     def choixTheme(self):
122         #create popup window with different options
123         self.popup=Toplevel(self)
124         self.popup.geometry('300x100+300+300')
125         self.popup.title("Theme")
126         self.b=Button(self.popup,text='Changer',command=self.popup.destroy)
127 
128 
129         #theme options
130 
131         choix1 = MonBoutonTheme(self.popup,self.colorSet,"Brown")
132         choix2 = MonBoutonTheme(self.popup,self.colorSet,"green")
133         choix3 = MonBoutonTheme(self.popup, self.colorSet,"#ec4062")
134         choix4 = MonBoutonTheme(self.popup,self.colorSet ,"#2687bc")
135 
136         #disposition
137         choix1.pack(side=LEFT, padx=10)
138         choix2.pack(side=LEFT, padx=10)
139 
140         choix3.pack(side=LEFT, padx=10)
141         choix4.pack(side=LEFT, padx=10)
142 
143 
144         self.b.pack(side=BOTTOM,padx=5, pady=10)
145 
146         #commandes
147 
148         choix1.config(command=choix1.cliquer)
149         choix2.config(command=choix2.cliquer)
150         choix3.config(command=choix3.cliquer)
151         choix4.config(command=choix4.cliquer)
152 
153     #theme change
154     def colorSet(self,color):
155         self.colorTheme=color
156         self.configure(bg=self.colorTheme)
```

Figure 2 - Implémentation de la choix de thème

Le choix effectif de la couleur du thème est appliqué avec la méthode *colorSet*, dans la ligne 154.

```
11  class MonBoutonTheme(Button):
12      def __init__(self,parent,methode,color):
13          Button.__init__(self,master=parent,bg=color,height = 1, width = 3)
14          self.methode=methode
15          self.__color=color
16 
17      def cliquer(self):
18          self.methode(self.__color)
```

Figure 3 - Définition de la classe MonBoutonTheme

4.2. Question 8 - Undo

Ici nous avons implémenté une méthode *undo (figure 4)* qui permet de revenir d'un ou plusieurs coups en arrière en effaçant la dernière forme ajoutée.

```
265     def undo(self):
266         if self.__nbManque>=1:
267             self.__nbManque=self.__nbManque-1
268             #masquage de la pendule
269             self.forme[self.__nbManque].setState("hidden")
```

Figure 4 - La méthode undo

Pour le bouton on l'a simplement fait avec la ligne 65 du code *(figure 5)*, son positionnement se fait à la ligne 94 du code.

```
64          #undo button
65          self.__undo = Button(self, text='Undo')
```

Figure 5 - Création du bouton Undo

Le bouton exécute la commande self.undo comme vu ci-après.

```
113         self.__undo.config(command=self.undo)
```

Figure 6 - Commande du bouton Undo

4.3. Bonus - Score Joueur et Historique

Nous devons ici implémenter un système de sauvegarde des échecs et des succès d'un joueur (identifié par un pseudo demandé au joueur en début de partie). Afin de résoudre ce problème nous avons créé une fenêtre d'introduction au jeu *(figure 7)*, elle a été faite avec un code séparé du code principal, il s'appelle *StartPage.py*.

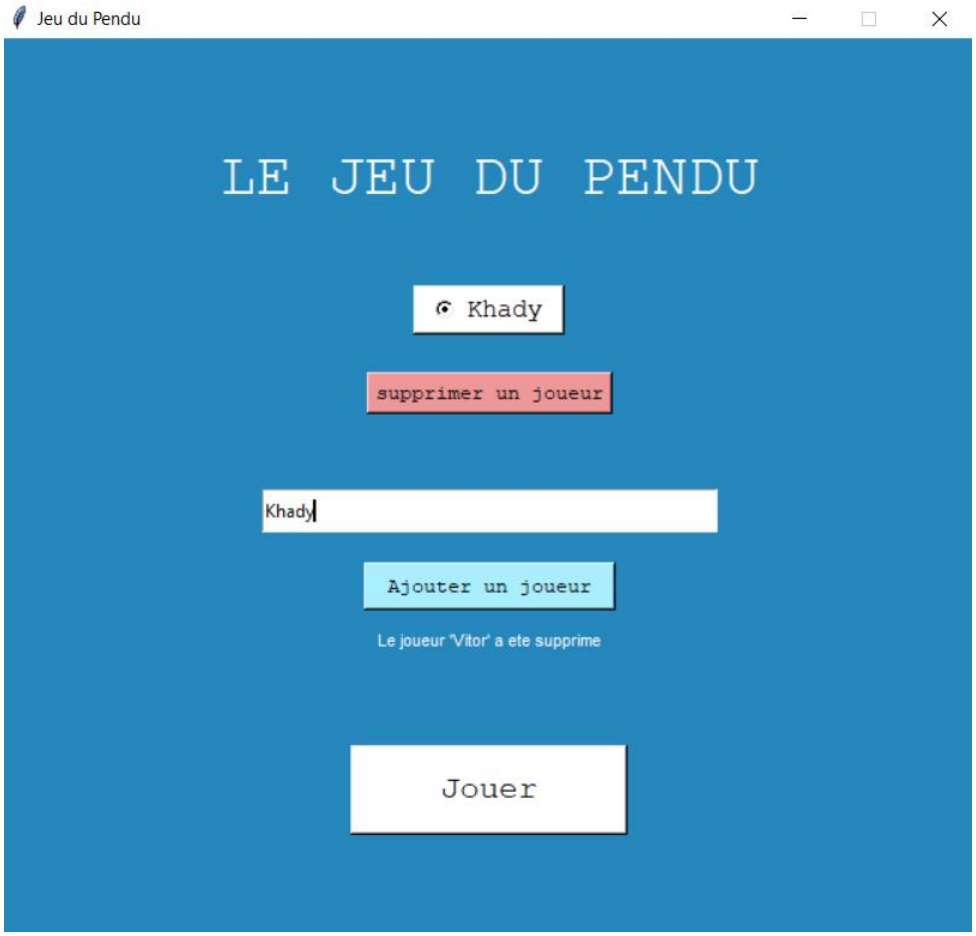


Figure 7 - Fenêtre d'introduction au jeu

Vous pourrez tester les détails de cette fenêtre (ajout double d'utilisateur, suppression, limitation du nombre de joueurs) dans le programme exécutable fourni. Puisque cette partie n'a pas été précisée dans l'énoncé, nous n'en ferons pas la description dans ce rapport. Nous avons jugé nécessaire l'implémentation de cette

partie car une création de joueur implique une suppression et une certaine capacité de gestion d'utilisateur.

4.3.1 Base de données *pendu.db*

Ici, le but principal est de stocker les données du joueur dans une base de données que nous avons appelée *pendu.db*, elle a été créée pour que nous puissions analyser la performance du joueur en faisant un score et un historique des parties. En ce qui concerne sa structure nous avons deux tables, une appelée Joueur (Idjoueur, pseudo) et autre Partie (Idpartie, Idjoueur, mot, score).

Il faut remarquer que la clé *Idjoueur* apparaît dans les deux tables en étant la clé primaire de la table *Joueur*, il faut la définir comme une clé étrangère dans la table 'partie' avec l'attribut ON DELETE CASCADE pour que la suppression d'un joueur entraine aussi la suppression de son historique de parties.

Pour la connexion avec la base de donnée on a le code affiché dans la *figure 8* et il faut pour cela importe sqlite3 comme module python (*figure 9*).

```
27          #database connection
28          self.bd="pendu.db"
29          try :
30              self.conn=sqlite3.connect(self.bd)
31              self.curseur=self.conn.cursor()
32          except Exception as err:
33              print("erreur de connexion à la base de donnee",err)
```

Figure 8 - Connection avec la base de donnée

```
7   import sqlite3
8   from tkinter import *
9   from FenPrincipale import *
```

Figure 9 - Appel des classes et méthodes du programme principal

Pour récupérer les noms qui existent déjà dans la base de données nous avons créé la méthode *getPlayers (figure 10)* qui la retourne. Pour cela nous avons défini une fonction appelée *assign (figure 11)* afin de récupérer les premiers éléments de l'objet afin de pouvoir utiliser la fonction map après.

```
118     def getPlayers(self):
119         self.requete="SELECT Pseudo FROM joueur"
120         self.curseur.execute(self.requete)
121         players=self.curseur.fetchall()
122         players=list(map(self.assign,players))
123         return players
```

Figure 10 - La méthode getPlayers

```
113     def assign(self,x):
114         x=x[0]
115         return x
```

Figure 11 - La méthode assign

4.3.2. Ajouter un nouveau joueur

Au niveau de l'interface de la fenêtre d'introduction, nous avons créé un input pour que le joueur puisse ajouter son nom *(figure 10)* et nous avons mis une limite maximum de trois joueurs. Tous s'affichent au-dessus de la fenêtre d'introduction *(figure 11)*.

```
65          #pseudo input
66          textvalue = StringVar()
67          textvalue.set("joueur")
68          self.entree = Entry(self.pseudoFrame, textvariable=textvalue, width=50)
69          self.entree.pack(side=TOP,pady=20,ipady=5)
70
```

Figure 10 - Input du nom du joueur

Figure 11 -L'affichage des noms des joueurs

Ensuite, la méthode *newPlayer (figure 12)* va récupérer l'entrée saisie et vérifier l'existence de l'utilisateur puis l'ajouter dans la base de données s'il n'existe pas. Pour cela, nous avons mis des différentes sorties (0, 1, 2 et 3) pour chaque situation et on les traite dans la méthode suivante *ajouterJoueur (figure 13).*

```
151     def newPlayer(self):
152         self.joueur=self.entree.get()
153 
154         try:
155             #verification de l'existence du joueur
156             self.requete="SELECT IdJoueur FROM joueur WHERE Pseudo='{}'".format(self.joueur)
157             self.curseur.execute(self.requete)
158             self.pseudo=self.curseur.fetchall()
159             #si le client est deja la
160             if self.pseudo!=[]:
161                 return 2
162 
163             #si c'est un nouveau joueur
164             if len(self.players)<self.playerlimit:
165                 self.curseur.execute("INSERT INTO joueur(Pseudo) VALUES ('{}')".format(self.joueur))
166                 self.conn.commit()
167             else:
168                 return 3
169 
170         except:
171             return 0
172 
173         #si la requete reussi et qu'un nouveau client est ajouté
174         return 1
```

Figure 12 - La méthode newPlayer

Pour donner un exemple, si le nom existe déjà dans la base de données on obtient un 2 comme sortie, et la méthode *ajouterJoueur* faire afficher le message: "Cet utilisateur existe déjà veuillez le sélectionner".

```
128     def ajouterJoueur(self):
129         result=self.newPlayer()
130         if result==2:
131             self.message.config(text="Cet utilisateur existe deja veuillez le selectionner")
132             self.message.pack(pady=10)
133         if result==3:
134             self.message.config(text="Vous ne pouvez pas creer plus de 3 joueurs,\n il faut en supprimer un")
135             self.message.pack(pady=5)
136         if result==0:
137             self.message.config(text="ohh!! l'ajout du joueur a echoué, reessaie pour voir!")
138             self.message.pack(pady=10)
139         if result==1:
140             #on affiche le nouveau joueur créé dans la liste des joueurs disponibles.
141             #la variable self.joueur a été créée dans la fonction newplayer et elle contient l'entrée de l'utilisateur
142             self.message.config(text="Super! l'ajout a reussi")
143             self.message.pack(pady=10)
144             #player list update
145             self.players.append(self.joueur)
146             #player button list update
147             self.playerButtonListUpdate()
```

Figure 13 - La méthode ajouteJoueur

4.3.3. Affichage des joueurs

La méthode suivante *(figure 14)* fait l'affichage de la liste de joueurs disponibles, comme la ligne 55 est assez longue on la reproduit entièrement juste après la figure :

```
49          #show existing players
50          self.players=self.getPlayers()
51          self.playersButton=[]
52          #player buttons create
53          self.value=StringVar()
54          for i in range(len(self.players)):
55              self.playersButton.append(Radiobutton(self.frame2,text=self.players[i],v
56              self.playersButton[i].pack(side=LEFT,ipady=1,ipadx=10,padx=20,pady=5)
```

Figure 14 - Méthode pour actualiser l'affichage des joueurs

Ligne 55 :

self.playersButton.append(Radiobutton(self.frame2,text=self.players[i],variable=self.value, value=i,relief=RAISED,font=("Courier", 13), bg="white"))

La boucle *for* ici a pour objectif de créer le bouton en fonction de chaque nom de joueur contenu dans la liste des joueurs. La méthode suivante *(figure 15)* actualise l'affichage de la liste de joueurs disponibles, comme la ligne 94 est assez longue on la reproduit entièrement jusqu'après la figure:

```
86      #cette methode actualise l'affichage de la liste des joueurs disponibles
87      def playerButtonListUpdate(self):
88          #old players button list hide
89          for j in range(len(self.playersButton)):
90              self.playersButton[j].pack_forget()
91          #new player button list show
92          self.playersButton.clear()
93          for i in range(len(self.players)):
94              self.playersButton.append(Radiobutton(self.frame2,text=self.players[i],v
95              self.playersButton[i].pack(side=LEFT,ipady=1,ipadx=10,padx=20,pady=5)
```

Figure 15 - Méthode pour actualiser l'affichage des joueurs

Nous avons eu besoin de recréer la liste des boutons car à la suppression d'un joueur, l'indexage change et il faudrait que les boutons et la liste des joueurs aient le même indexage.

Ligne 94 :

self.playersButton.append(Radiobutton(self.frame2,text=self.players[i],variable=self.value, value=i,relief=RAISED,font=("Courier", 13), bg="white"))

4.3.4. Effacer un joueur

Pour effacer un joueur nous avons créé la méthode *deletePlayer*. Avant tout, il faut le sélectionner dans le menu au-dessus de la page, ainsi on récupère son index dans la liste et il faut cliquer sur le bouton "supprimer un joueur" *(figure 16)*, sa définition est affichée dans la figure 17. Comme la ligne 59 est assez longue, on la montre entièrement juste en-dessous de la figure 17.

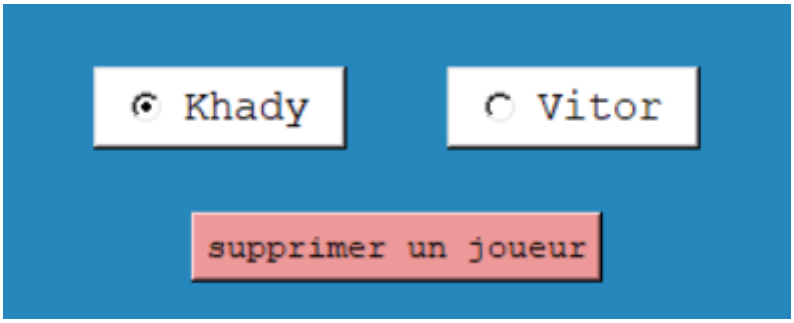


Figure 16 - Bouton "supprimer un joueur"

```
58      #player delete button
59      delete=Button(self.frame,text="supprimer un joueur",font=
60
61      self.pseudoFrame=Frame(self.frame,bg="#2687bc")
62      self.pseudoFrame.pack(pady=30)
```

Figure 17 - Définition du bouton "supprimer un joueur"

Ligne 59:

delete=Button(self.frame,text="supprimer un joueur",font=("Courier", 10), bg="#ee9999",command=self.deletePlayer).pack()

```
178     def deletePlayer(self):
179         try:
180             toDeleteIndex=int(self.value.get())
181         except:
182             self.message.config(text="Il faut selectionner le joueur à supprimer d'abord")
183             self.message.pack(pady=10)
184
185         toDelete=self.players[toDeleteIndex]
186         #requete de suppression
187         self.requete="DELETE FROM joueur WHERE Pseudo='{}'".format(toDelete)
188         self.curseur.execute(self.requete)
189         self.conn.commit()
190
191         #players list update
192         self.players.pop(toDeleteIndex)
193
194         #players button list update (on refait l'affichage car les indexes on maintenant cha
195         self.playerButtonListUpdate()
196
197         #message update
198         self.message.config(text="Le joueur '{}' a ete supprime".format(toDelete))
199         self.message.pack(pady=10)
```

Figure 18 - La méthode deletePlayer

4.3.5. Commencer la partie

Pour commencer la partie avec le joueur choisi, nous avons créé la méthode *startnext* avec la commande destroy pour la page d'introduction.

```
 99      def startnext(self):
100          try:
101              playerIndex=int(self.value.get())
102              player=self.players[playerIndex]
103              app = FenPrincipale(player)
104              StartPage.destroy(self)
105          except:
106              self.message.config(text="Il faut selectionner un joueur d'abord")
107              self.message.pack(pady=10)
```

Figure 19 - Méthode qui initialise le jeu

Cette méthode va alors détruire la page d'accueil et appeler la FenPrincipale en lui passant le nom du joueur.

4.3.6. Score et historique

Nous avons développé un système d'affichage du score du joueur ainsi que son historique. Le score est actualisé à chaque fin de partie.

Figure 20 – Score et bouton historique

Les boutons et labels pour l'historique et le score sont créés comme suit.

```
#affichage du nom du joueur et du score enregistré
self.joueurLabel=Label(joueurFrame,text=self.joueur,bg="#2687bc",fg="white")
self.joueurScoreLabel=Label(joueurFrame,text="Score: ",bg="#2687bc",fg="white")
#bouton pour afficher l'historique
self.boutonHistorique=Button(joueurFrame,text="Voir historique",command=self.historiq
```

Figure 21 – creation label Score et bouton historique

On va d'abord récupérer le score du joueur dans la base de données en faisant la somme de ses victoires (1) et défaites (0).

```
#cette methode permet de recuperer l'IdJoueur du joueur ayant ce nom dans la table joueur
def getJoueurId(self):
    try:
        self.requete="SELECT IdJoueur FROM joueur WHERE Pseudo='{}'".format(self.joueur)
        self.curseur.execute(self.requete)
        joueurId=self.curseur.fetchone()
        return joueurId[0]
    except:
        return -1

def getJoueurScore(self):
    try:
        self.requete="SELECT SUM(score) FROM Partie WHERE IdJoueur='{}'".format(self.joue
        self.curseur.execute(self.requete)
        score=self.curseur.fetchone()
        print(score[0])
        if score[0]!=None:
            score=score[0]
        else:
            score=0

        print(score)
        return score
    except:
        print("erreur")
        return -1
```

Figure 22 – Récupération du score à partie de l'ID du joueur

Puis nous affichons le score dans la méthode « lancer » qui permet de commencer une partie

```
#affichage du score
self.joueurScoreLabel.config(text='Score: '+str(self.joueurScore))
```

Figure 23 – Affichage du score

En fin de partie le nouveau score est alors enregistré :

```
if gagne==True:
    self.__lmot.config(text=self.__mot+'-Bravo!! Vous avez gagne')
    score=1
else:
    self.__lmot.config(text=self.__mot+'-Vous avez perdu, le mot etait:'+self.__mot)
    score=0

#sauvegarde du score dans la base de donnée
self.saveGame(score)

#update affichage du score
self.joueurScore=self.getJoueurScore()
self.joueurScoreLabel.config(text='Score: '+str(self.joueurScore))
```

Figure 24 – Sauvegarde et affichage du nouveau score

La méthode saveGame est définie ci-dessous :

```
#cette methode se charge d'enregistrer le score du joueur apres le jeu
def saveGame(self,score):
    try:
        self.curseur.execute("INSERT INTO Partie(IdJoueur,Mot,score) VALUES ('{}','{}','{}')".format(self.joueurId,self.__mot,score))
        self.conn.commit()
    except:
        print("echec")
        return -1
```

Figure 25 – Sauvegarde du nouveau score et mot

L'historique est géré à l'aide de la méthode historique présentée ci-après :

```
def historique(self):
    #create popup window with game history
    self.historique=Toplevel(self)
    self.historique.title("Historique des parties")

    try:
        self.requete="SELECT * FROM Partie WHERE IdJoueur='{}'".format(self.joueurId)
        self.curseur.execute(self.requete)
        histoArray=self.curseur.fetchall()
    except:
        print("oups")
        return 0

    #creation du tableau de l'historique
    tableau = Treeview(self.historique, columns=('Mot', 'Resultat'))
    tableau.column("Resultat", width=20)
    tableau.column("Resultat", width=70)

    tableau.heading('Mot', text='Mot')

    tableau.heading('Resultat', text='Resultat')

    tableau['show'] = 'headings' # sans ceci, il y avait une colonne vide à gauche qui a pour rôle d'af

    tableau.pack(padx = 10, pady = (0, 10))
    print(histoArray)

    for enreg in histoArray:
        tableau.insert('', 'end', iid=enreg[0], values=(enreg[2], enreg[3]))


    self.quit=Button(self.historique,text='Fermer',command=self.historique.destroy).pack(side=BOTTOM,pa
```

Il récupère les informations concernant le joueur de cette partie dans la base de données puis l'affiche dans un tableau géré par Treeview.

Conclusion

Nous avons donc développé une application comme il a été demandé et ce BE nous a permis de mieux appréhender la gestion d'une interface graphique sur python, en plus de consolider les notions acquises en gestion de base de données et modélisation.