Import des modules

Imports `from`

Division décimale pour `Python 2`

```
from __future__ import division
```

Interface graphique `PyQt4`

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from PyQt4 import QtNetwork
```

Fichiers d'interfaces créés avec *Qt Creator*

```
from _ui.ui_menu import Ui_Menu
from _ui.ui_module import Ui_Module
from _ui.ui_fenetrebvn import Ui_FenetreBVN
from _ui.ui_score import Ui_Score
```

Modules standars

```
from time import time, sleep, localtime
from random import randint
from os import system, popen
from math import log
```

Imports simples

Modules auxiliaires pour géner le cryptage

```
import crypterScore
import crypterTexte
```

Module pour ouvrir des pages web

```
import webbrowser
```

Modules standards

```
import sys
import os
import pickle
import binascii
import re
import unicodedata
```

Modules nécessaires à la production de l'éxecutable

```
import atexit
import platform
```

Déclaration des classes

Classe `FenetreBVNApplication`

Cette classe hérite des classes `QMainWindow` et `Ui_FenetreBVN` et permet la création du GUI et toute sa gestion.
Cette classe contient la majeure partie du programme de la fenêtre de bienvenue
Elle est directement issue de *Qt* (et donc `PyQt`)

```
class FenetreBVNApplication(QMainWindow, Ui_FenetreBVN):
    valeur_quitter = pyqtSignal(bool)
    termine = pyqtSignal()
```

Méthode d'initialisation `__init__`

Méthode permettant d'initialiser la classe

```
    def __init__(self, parent=None):
```

On hérite de la méthode `__init__` des classes parentes

```
        super(FenetreBVNApplication, self).__init__(parent)
```

On initialise les widgets décris dans le fichier auxiliaire `ui_fenetrebvn.py` créé avec *Qt Creator* et `PyQt`

```
        self.setupUi(self)
```

On affiche le logo du programme dans la zone prévue

```
        self.LabelIcone.setPixmap(QPixmap(os.getcwd() + "/img/logo1.png"))
```

On connecte les boutons à leurs slots respectifs

```
        self.BoutonContinuer.clicked.connect(self.continuer)
        self.BoutonQuitter.clicked.connect(self.quitter)
```

Méthode spéciale `keyPressEvent`

Méthode permettant de gérer les pressions de touches du clavier, prenant en paramètre `event`

```
    def keyPressEvent(self, event):
```

Si l'événement correspond à une pression de touche

```
        if type(event) == QKeyEvent:
```

Si la touche pressée est `<Escape>`

```
            if event.key() == Qt.Key_Escape:
```

On appelle la méthode `quitter`

```
                self.quitter()
```

Si la touche pressée est `<Return>` ou `<Space>`

```
            elif event.key() == Qt.Key_Return or event.key() == Qt.Key_Space:
```

On appelle la méthode `continuer`

```
                self.continuer()
```

Méthode (slot) `quitter`

Méthode permettant de quitter la fenêtre et le programme

```
    @pyqtSlot()
    def quitter(self):
```

On émet le signal `termine`

```
        self.termine.emit()
```

On émet le code de retour : `True` (quitter le programme)

```
        self.valeur_quitter.emit(True)
```

On quitte la fenêtre

```
        self.close()
```

Méthode (slot) `continuer`

Méthode permettant de quitter la fenêtre mais de continuer le programme

```
    @pyqtSlot()
    def continuer(self):
```

On émet le signal `termine`

```
        self.termine.emit()
```

On émet le code de retour : `False` (continuer le programme)

```
        self.valeur_quitter.emit(False)
```

On quitte la fenêtre

```
        self.close()
```

Classe `MenuApplication`

Cette classe hérite des classes `QMainWindow` et `Ui_Menu` et permet la création du GUI et toute sa gestion.
Cette classe contient la majeure partie du programme du menu
Elle est directement issue de *Qt* (et donc `PyQt`)

```
class MenuApplication(QMainWindow, Ui_Menu):
```

Méthode d'initialisation `__init__`

Méthode permettant d'initialiser la classe

```
    def __init__(self, parent=None):
```

On hérite de la méthode `__init__` des classes parentes

```
        super(MenuApplication, self).__init__(parent)
```

On initialise les widgets décris dans le fichier auxiliaire `ui_menu.py` créé avec *Qt Creator* et `PyQt`

```
        self.setupUi(self)
```

On affiche le logo dans la zone prévue

```
        self.LabelIcone.setPixmap(QPixmap(os.getcwd() + "/img/logo2.png"))
```

On appelle la méthode `majTextesEx` pour mettre à jour les titres des textes d'exemple

```
        self.majTextesEx()
```

On définit l'onglet actuel comme le second (textes d'exemple)

```
        self.tab_actuel = 1
```

On définit les attributs

```
        self.param = []
        self.pseudo = ""
```

On connecte les boutons à leurs slots

```
        self.BoutonCommencer.clicked.connect(self.commencer)
        self.BoutonQuitter.clicked.connect(self.quitter)
        self.BoutonAide.clicked.connect(self.ouvrirAide)
```

On connecte le changement d'onglet au slot correspondant

```
        self.TabSourceTexte.currentChanged.connect(self.tabChange)
```

On connecte les changements de texte aux slots correspondants

```
        self.SBoxMotsFR.valueChanged.connect(self.entryMotsFRCliquee)
        self.EntrySyll.textChanged.connect(self.entrySyllCliquee)
        self.CollerTexteV.textChanged.connect(self.entryCollerCliquee)
        self.NomTexteV.textChanged.connect(self.entryNomTexteCliquee)
```

Méthode spéciale `keyPressEvent`

Méthode permettant de gérer les pressions de touches du clavier, prenant en paramètre `event`

```
    def keyPressEvent(self, event):
```

Si l'événement correspond à une pression de touche

```
        if type(event) == QKeyEvent:
```

Si la touche pressée est `<Escape>`

```
            if event.key() == Qt.Key_Escape:
```

On appelle la méthode `quitter`

```
                self.quitter()
```

Si la touche pressée est `<Return>`

```
            elif event.key() == Qt.Key_Return:
```

On appelle la méthode `commencer`

```
                self.commencer()
```

Si la touche pressée est `<F1>`

```
            elif event.key() == Qt.Key_F1:
```

On appelle la méthode `ouvrirAide`

```
                self.ouvrirAide()
```

Méthode `majTextesEx`

Méthode permettant de mettre à jour les titres des textes d'exemple dans le menu

```
    def majTextesEx(self):
```

On ouvre les 3 textes, et on ne conserve que la première ligne sans le `"#"`, qui correspond au titre

```
        with open("txt/exemple1_e.txt", "r") as fichier:
            texte = crypterTexte.decrypterTexte(fichier.read())
            titre1 = texte.split("\n")[0][1:]
        with open("txt/exemple2_e.txt", "r") as fichier:
            texte = crypterTexte.decrypterTexte(fichier.read())
            titre2 = texte.split("\n")[0][1:]
        with open("txt/exemple3_e.txt", "r") as fichier:
            texte = crypterTexte.decrypterTexte(fichier.read())
            titre3 = texte.split("\n")[0][1:]
```

On affiche les nouveaux titres dans le menu

```
        self.LabelTexteEx1R.setText(titre1.decode("utf-8"))
        self.LabelTexteEx2R.setText(titre2.decode("utf-8"))
        self.LabelTexteEx3R.setText(titre3.decode("utf-8"))
```

Méthode (slot) `ouvrirAide`

Méthode permettant d'ouvrir le PDF d'aide

```
    @pyqtSlot()
    def ouvrirAide(self):
```

On ouvre le PDF d'aide avec le navigateur internet

```
        webbrowser.open("aide/aide.pdf")
```

Méthode `affFenetreBVN`

Méthode appelée avant de `show` le menu et permettant de lancer la fenêtre de bienvenue

```
    def affFenetreBVN(self):
```

On crée une nouvelle instance de la classe `FenetreBVNApplication`

```
        self.FenetreBVN = FenetreBVNApplication()
```

On connecte le signal de retour et la fermeture de la fenêtre aux méthodes `handleQuitterBVN` et `termineBVN`

```
        self.FenetreBVN.valeur_quitter.connect(self.handleQuitterBVN)
        self.FenetreBVN.termine.connect(self.termineBVN)
```

On affiche la fenêtre

```
        self.FenetreBVN.setWindowModality(Qt.ApplicationModal)
        self.FenetreBVN.show()
```

On met à jour les scores en appelant la fonction `crypterScore.syncDB`

```
        crypterScore.syncDB()
```

Méthode `normaliserTexte`

Méthode permettant de normaliser un texte, en le recoupant à une taille limite et en remplaçant les caractères d'espacement

```
    def normaliserTexte(self, texte):
```

On supprime les caractères d'espacement que l'on remplace par des espaces

```
        texte = (re.sub(r"[\n\t\b\a\r]", r" ", texte)).strip()
```

On remplace ensuite les espaces multiples par un espace simple

```
        texte = (re.sub(r" {2,}", r" ", texte.strip()))
```

On recoupe le texte si il est trop long

```
        if len(texte) > 4096:
            texte = texte[:4096]
```

On renvoie le texte modifié

```
        return texte
```

Méthode `cheat`

Méthode permettant de détecter si le texte choisi est conforme (assez long, vrais mots...)

```
    def cheat(self, texte):
```

On compte le nombre de mots et de caractères

```
        mots = len(texte.split(" "))
        car = len(texte.replace(" ", ""))
```

On compte le nombre de caractères différents

```
        car_d = []
        i = 0
        while len(car_d) < 5 and i < len(texte) - 1:
            c = texte[i]
            if c in car_d:
                pass
            else:
                car_d.append(c)
            i += 1
        nb_car_d = len(car_d)
```

On retourne un booléen correspondant aux conditions :
- 5 mots minimum
- 10 caractères minimum
- 5 caractères différents minimum

```
        return (mots < 5 or car < 10 or nb_car_d < 5)
```

Méthode `getParam`

Méthode permettant de récupérer les différents paramètres de l'interface

```
    def getParam(self):
```

On récupère le temps choisi que l'on exprime en secondes

```
        temps = float(self.SBoxMinutes.value() * 60 +
                      self.SBoxSecondes.value())
```

Si le temps n'est pas conforme, on utilise les valeurs par défaut

```
        if temps == 0.0:
            temps = 60.0
            self.SBoxMinutes.setValue(int(temps // 60))
            self.SBoxSecondes.setValue(int(temps % 60))
        elif temps <= 10.0 and self.pseudo.lower() != "admin":
            temps = 10.0
            self.SBoxMinutes.setValue(int(temps // 60))
            self.SBoxSecondes.setValue(int(temps % 60))
```

Si le premier onglet est l'onglet actif

```
        if self.tab_actuel == 0:
```

Si *MotsFR* est choisi

```
            if self.RadioMotsFR.isChecked():
```

On récupère la valeur pour *MotsFR*

```
                nb_mots = self.SBoxMotsFR.value()
```

On définit le mode de texte : `"mots_fr::<val>"`

```
                texte_mode = "mots_fr::{}".format(nb_mots)
```

Si *Syll* est choisi

```
            elif self.RadioSyll.isChecked():
```

On récupère la syllabe entrée

```
syll = unicode(self.EntrySyll.text()).encode("utf-8").strip()
```

Si rien n'est rentré, on utilise une valeur par défaut

```
if not syll:
    syll = "ion"
    self.EntrySyll.setText(u"ion")
```

On ouvre le dictionnaire de référence et on récupère le contenu

```
fichier_mots_brut = open("txt/dictionnaire_syll_e.txt", "r")
fichier_mots =\
    crypterTexte.decrypterTexte(fichier_mots_brut.read())
fichier_mots = fichier_mots.split("\n")
liste_mots = [elt for elt in fichier_mots if elt and
              (syll in elt)]
fichier_mots_brut.close()
```

Si moins de 25 mots du dictionnaire possèdent la syllabe entée, on utilise la valeur par défaut

```
if len(liste_mots) < 25:
    syll = "ion"
    self.EntrySyll.setText(u"ion")
```

On définit le mode de texte : `"syll::<val>"`

```
texte_mode = "syll::{}".format(syll)
```

Si le deuxième onglet est l'onglet actif

```
if self.tab_actuel == 1:
```

Si le texte d'exemple 1 est choisi

```
if self.LabelTexteEx1R.isChecked():
```

On définit le mode de texte : `"expl::1"`

```
texte_mode = "expl::1"
```

Si le texte d'exemple 2 est choisi

```
            elif self.LabelTexteEx2R.isChecked():
```

On définit le mode de texte : `"expl::1"`

```
                texte_mode = "expl::2"
```

Si le texte d'exemple 3 est choisi

```
            elif self.LabelTexteEx3R.isChecked():
```

On définit le mode de texte : `"expl::1"`

```
                texte_mode = "expl::3"
```

Si le troisième onglet est l'onglet actif

```
        if self.tab_actuel == 2:
```

Si *Texte à copier-coller* est choisi

```
            if self.CollerTexteR.isChecked():
```

On récupère le texte entré

```
                texte = unicode(self.CollerTexteV.toPlainText())\
                    .encode("utf-8").strip()
```

Si le texte normalisé n'est pas conforme

```
                if self.cheat(self.normaliserTexte(texte)):
```

On utilise des valeurs par défaut : texte d'exemple 1

```
                    texte_mode = "expl::1"
```

Si le texte est conforme

```
                else:
```

On définit le mode de texte : `"perso::entier::{{{<val>}}}"`

```
                    texte_mode = "perso::entier::{{{" + texte + "}}}"
```

Si *Texte choisi par nom* est choisi

```
            elif self.NomTexteR.isChecked():
```

On récupère le nom

```
                nom = unicode(self.NomTexteV.text()).encode("utf-8").strip()
```

On essaye

```
                try:
```

On ouvre le fichier

```
                    fichier_brut = open("txt/" + nom, "r")
```

On lit et noramlise le texte

```
                    fichier = self.normaliserTexte(fichier_brut.read())
```

On ferme le fichier

```
                    fichier_brut.close()
```

On lève une exception si le texte normalisé n'est pas conforme

```
                    if self.cheat(fichier):
                        raise TricheError
```

Si tout est passé, on défini le mode de texte : `"perso::nom::{{{<val>}}}"`

```
                    texte_mode = "perso::nom::{{{" + nom + "}}}"
```

Si une exception a été levée (il y a eu une erreur)

```
                except:
```

On utilise des valeurs par défaut : texte d'exemple 1

```
                    texte_mode = "expl::1"
```

On met à jour l'attribut-liste `param` avec les nouvelles valeurs du temps et du mode de texte

```
        self.param = [temps, texte_mode]
```

Méthode `getPseudo`

Méthode permettant de récupérer le pseudo saisi par l'utilisateur

```
    def getPseudo(self):
```

On récupère le pseudo entré

```
        pseudo = unicode(self.EntryPseudo.text()).encode("utf-8").strip()
```

Si aucun pseudo n'est entré, on choisit `"Anonyme"`

```
        if not pseudo.strip():
            pseudo = "Anonyme"
```

Si le pseudo est trop long, on le recoupe à 20 caractères

```
        if len(pseudo) > 20:
            pseudo = pseudo[:20]
```

On met à jour la valeur de l'attribut `pseudo`

```
        self.pseudo = pseudo
```

Méthode (slot) `commencer`

Méthode permettant de lancer une partie et donc d'afficher une fenêtre de *Module*

```
    @pyqtSlot()
    def commencer(self):
```

On récupère les paramètres et le pseudo

```
        self.getParam()
        self.getPseudo()
```

On créé une instance de `ModuleApplication`, en passant les paramètres et le pseudo au constructeur

```
        self.Module = ModuleApplication(self.param, self.pseudo)
```

On connecte le signal `termine` de la nouvelle fenêtre à la bonne méthode

```
        self.Module.termine.connect(self.termineModule)
```

On prépare la nouvelle fenêtre, on cache le menu et on affiche la nouvelle fenêtre

```
        self.Module.setWindowModality(Qt.ApplicationModal)
        self.hide()
        self.Module.show()
```

Méthode (slot) `quitter`

Méthode permettant de quitter le menu et donc le programme

```
    @pyqtSlot()
    def quitter(self):
```

On ferme la fenêtre menu

```
        self.close()
```

Méthode (slot) `handleQuitterBVN`

Méthode prenant en paramètre la valeur de retour de la fenêtre BVN et permettant soit de continuer, soit de quitter le programme

```
    @pyqtSlot(bool)
    def handleQuitterBVN(self, valeur_quitter):
```

Si le code de retour vaut `True`

```
        if valeur_quitter:
```

On quitte le menu donc le programme

```
            self.quitter()
```

Méthode (slot) `termineBVN`

Méthode permettant de gérer la fermeture de la fenêtre BVN

```
    @pyqtSlot()
    def termineBVN(self):
```

On détruit l'instance en cours de fenêtre BVN, pour éviter les collisions

```
        del self.FenetreBVN
```

On réaffiche le menu

```
        self.show()
```

Méthode (slot) `termineModule`

Méthode prenant en paramètre le code de retour de la fenêtre du module et permettant soit de recommencer une partie identique, soit de réafficher le menu

```
    @pyqtSlot(bool)
    def termineModule(self, recommencerModule):
```

On supprime l'instance en cours de la fenêtre du module, pour éviter les collisions

```
        del self.Module
```

Si le code de retour pour recommencer vaut `True`

```
        if recommencerModule:
```

On réappelle la méthode `commencer`

```
            self.commencer()
```

Sinon (si l'utilisateur ne veut pas recommencer)

```
        else:
```

On réaffiche le menu

```
            self.show()
```

Méthode (slot) `tabChange`

Méthode permettant de gérer le changement d'onglet pour le choix du texte et prenant en paramètre le nouveau numéro de l'onglet (0, 1 ou 2)

```
    @pyqtSlot(int)
    def tabChange(self, no):
        self.tab_actuel = no
```

Si l'onglet actif est le premier

```
        if self.tab_actuel == 0:
```

On remet aux valeurs par défaut le deuxième

```
            self.LabelTexteEx1R.setChecked(True)
            self.LabelTexteEx2R.setChecked(False)
            self.LabelTexteEx3R.setChecked(False)
```

On remet aux valeurs par défaut le troisième

```
            self.CollerTexteR.setChecked(True)
            self.CollerTexteV.setPlainText("")
            self.NomTexteR.setChecked(False)
            self.NomTexteV.setText("")
```

Si l'onglet actif est le deuxième

```
        if self.tab_actuel == 1:
```

On remet aux valeurs par défaut le premier

```
            self.RadioMotsFR.setChecked(True)
            self.SBoxMotsFR.setValue(100)
            self.RadioSyll.setChecked(False)
            self.EntrySyll.setText("")
```

On remet aux valeurs par défaut le troisième

```
            self.CollerTexteR.setChecked(True)
            self.CollerTexteV.setPlainText("")
            self.NomTexteR.setChecked(False)
            self.NomTexteV.setText("")
```

Si l'onglet actif est le troisième

```
        if self.tab_actuel == 2:
```

On remet aux valeurs par défaut le premier

```
            self.RadioMotsFR.setChecked(True)
            self.SBoxMotsFR.setValue(100)
            self.RadioSyll.setChecked(False)
            self.EntrySyll.setText("")
```

On remet aux valeurs par défaut le deuxième

```
            self.LabelTexteEx1R.setChecked(True)
            self.LabelTexteEx2R.setChecked(False)
            self.LabelTexteEx3R.setChecked(False)
```

Méthode (slot) `entryMotsFRCliquee`

Méthode permettant de changer la valeur du choix quand du texte est entré dans la boîte de texte *Mots FR*

```
    @pyqtSlot()
    def entryMotsFRCliquee(self):
```

Si l'onglet actif est le premier (pour éviter de déclencher la méthode lors de la remise par défaut des onglets non actifs)

```
        if self.tab_actuel == 0:
```

On met le choix *Mots FR* à `True`

```
            self.RadioMotsFR.setChecked(True)
```

On met le choix *Syll* à `False`

```
            self.RadioSyll.setChecked(False)
```

Méthode (slot) `entrySyllCliquee`

Méthode permettant de changer la valeur du choix quand du texte est entré dans la boîte de texte *Syll*

```
    @pyqtSlot()
    def entrySyllCliquee(self):
```

Si l'onglet actif est le premier (pour éviter de déclencher la méthode lors de la remise par défaut des onglets non actifs)

```
        if self.tab_actuel == 0:
```

On met le choix *Mots FR* à `False`

```
            self.RadioMotsFR.setChecked(False)
```

On met le choix *Syll* à `True`

```
            self.RadioSyll.setChecked(True)
```

Méthode (slot) `entryCollerCliquee` Méthode permettant de changer la valeur du choix quand du texte est entré dans la boîte de texte *Texte copié-collé*

```
    @pyqtSlot()
    def entryCollerCliquee(self):
```

Si l'onglet actif est le troisième (pour éviter de déclencher la méthode lors de la remise par défaut des onglets non actifs)

```
        if self.tab_actuel == 2:
```

On met le choix *Texte copié-collé* à `True`

```
            self.CollerTexteR.setChecked(True)
```

On met le choix *Texte par nom* à `False`

```
            self.NomTexteR.setChecked(False)
```

Méthode (slot) `entryNomTexteCliquee` Méthode permettant de changer la valeur du choix quand du texte est entré dans la boîte de texte *Texte par nom*

```
    @pyqtSlot()
    def entryNomTexteCliquee(self):
```

Si l'onglet actif est le troisième (pour éviter de déclencher la méthode lors de la remise par défaut des onglets non actifs)

```
        if self.tab_actuel == 2:
```

On met le choix *Texte copié-collé* à `False`

```
            self.CollerTexteR.setChecked(False)
```

On met le choix *Texte par nom* à `True`

```
            self.NomTexteR.setChecked(True)
```

Classe `ModuleApplication`

Cette classe hérite des classes `QMainWindow` et `Ui_Module` et permet la création du GUI et toute sa gestion.
Cette classe contient la majeure partie du programme du module
Elle est directement issue de *Qt* (et donc `PyQt`)

```
class ModuleApplication(QMainWindow, Ui_Module):
    termine = pyqtSignal(bool)
```

Méthode d'initialisation `__init__`

Méthode permettant d'initialiser la classe

```
    def __init__(self, param=[60, "expl::1"], pseudo="Anonyme", parent=None):
```

On hérite de la méthode `__init__` des classes parentes

```
        super(ModuleApplication, self).__init__(parent)
```

On initialise les widgets décris dans le fichier auxiliaire `ui_module.py` créé avec *Qt Creator* et `PyQt`

```
        self.setupUi(self)
```

On part du principe que les paramètres auront été vérifiés par le menu et qu'ils sont donc exempts d'erreurs

```
self.temps_choisi = param[0]
self.mode_texte = param[1]
self.pseudo = pseudo

self.texte = ""
self.genererTexte()
```

On définit les attributs

```
self.recommencerV = False
self.pos_texte = 0
self.texte_d = self.texte[:self.pos_texte]
self.texte_g = self.texte[(self.pos_texte + 1):]
self.car_attendu = self.texte[self.pos_texte]
self.der_car_T = ""
self.dernier_juste = False
self.jeton_pauseM = True
self.temps_restant = 0.0
self.premier_lancement_timer = True
self.couleur_backup = ""
self.jeton_temps_finiM = False
self.mots_min = 0.0
self.car_min = 0.0
self.temps_ecoule = 0.0
self.score = 0.0
self.temps_score_precedant = 0.0
self.C_TEMPS = 5
self.C_SCORE = 10
self.car_justes = 0
self.car_faux = 0
self.car_faux_precedant = 0
self.reussite = 0.0
self.erreurs = 0.0
self.nombre_mots_precedant = 0
self.car_attendu_precedant = ""
self.coeff_pre = 1.0
```

On créé l'attribut `Timer`, qui est une instance du `ThreadTimer` déclaré plus haut. On lui passe en argument le temps choisi dans le fichier de configuration

```
self.Timer = ThreadTimer(self.temps_choisi)
```

On connecte le signal `finished` du timer à la méthode en charge de le détruire proprement

```
self.Timer.finished.connect(self.Timer.deleteLater)
```

On lance une première fois les méthodes `updateTexteLabel` et `temps_change` pour régler le GUI sur la position de départ

```
self.updateTexteLabel()
self.temps_change(self.temps_choisi)
self.meilleursScores()
```

On fixe la police des labels en police à chasse fixe (monospace)
Cela permet d'éviter l'erreur avec Windows qui ne reconnait pas la police `Monospace`

```
Police = QFont("Monospace", 30)
Police.setStyleHint(QFont.TypeWriter)
self.LabelTexteDroite.setFont(Police)
self.LabelTexteCentre.setFont(Police)
self.LabelTexteGauche.setFont(Police)
self.LabelTapeDroit.setFont(Police)
self.EntryTapeCentre.setFont(Police)
```

Quand le texte dans la boîte est changé (frappe de l'utilisateur), on appelle la méthode `getDerCar` en charge de récupérer la saisie

```
self.EntryTapeCentre.textChanged.connect(self.getDerCar)
```

Quand on clique sur le bouton start/pause, on appelle la méthode `togglePauseM` en charge du basculement start/pause

```
self.BoutonStartPause.clicked.connect(self.togglePauseM)
```

Quand on clique sur le bouton quitter, on appelle la méthode `quitterM` en charge de fermer proprement le timer avant de quitter le GUI

```
self.BoutonQuitter.clicked.connect(self.quitterM)
```

On connecte les signaux du timer `temps_change` et `temps_fini` aux méthodes du GUI associées, qui servent à interpréter quand le temps restant change et quand le temps est fini

```
        self.Timer.temps_change_signal.connect(self.temps_change)
        self.Timer.temps_fini_signal.connect(self.temps_fini)
```

On désactive les widgets tant que l'utilisateur ne clique pas sur commencer ou qu'il ne tape pas de lettre

```
        self.LabelTexteDroite.setEnabled(False)
        self.LabelTexteCentre.setEnabled(False)
        self.LabelTexteGauche.setEnabled(False)
        self.LabelTapeDroit.setEnabled(False)
        self.LabelTapeFleche.setEnabled(False)
```

On active le focus sur la boîte de texte (comme ça l'utilisateur n'a pas à cliquer dessus)

```
        self.EntryTapeCentre.setFocus()
```

Méthode `meilleursScores`

Méthode permettant d'afficher les meilleurs scores dans le tableau en bas à droite

```
    def meilleursScores(self):
```

On appelle la méthode `trouverMeilleursScores`

```
        meilleurs = self.trouverMeilleursScores()
```

On affiche les valeurs obtenues

```
        self.LabelMeilleur1.setText(unicode(str(meilleurs[0])))
        self.LabelMeilleur2.setText(unicode(str(meilleurs[1])))
        self.LabelMeilleur3.setText(unicode(str(meilleurs[2])))
```

Méthode `trouverMeilleursScores`

Méthode permettant de trouver les 3 meilleurs scores spéciaux à partir de la DB locale

```
    def trouverMeilleursScores(self):
```

On essaye

```
        try:
```

On ouvre le fichier de la DB locale

```
            fichier_db = open("score/local_db.db", "rb")
```

On récupère le contenu avec un pickler

```
            mon_pickler = pickle.Unpickler(fichier_db)
            liste = mon_pickler.load()
            fichier_db.close()
```

Si l'objet récupéré est vide ou `None`

```
            if not liste:
```

On lève l'exception `VideException`

```
                raise VideException
```

On recherche d'abord le meilleur score absolu

```
            pas_encore_trouve_best = True
            i = 0
            best = None
```

Tant que l'on ne l'a pas trouvé et que la liste n'est pas finie

```
            while pas_encore_trouve_best and i < len(liste) - 1:
```

Si le pseudo ne commence pas par `"#"`

```
                if liste[i]["pseudo"][0] != "#":
```

On considère que c'est le meilleur score

```
                    pas_encore_trouve_best = False
                    best = liste[i]["score"]
                i += 1
```

Si aucune valeur n'a été trouvée (la DB est vide ou incorrecte), on lève l'exception `VideException`

```
            if not best:
                raise VideException
```

On recherche ensuite le meilleur score dans le même mode

```
            pas_encore_trouve_best_mode = True
            i = 0
            best_mode = None
```

Tant que l'on ne l'a pas trouvé et que la liste n'est pas finie

```
            while pas_encore_trouve_best_mode and i < len(liste) - 1:
```

Si le pseudo ne commence pas par `"#"`

```
                if liste[i]["pseudo"][0] != "#":
```

Si le mode du score examiné est le même que le score réalisé

```
                    if str(liste[i]["texte_mode_enh"]) == self.modeTexteEnh():
```

On considère que c'est le meilleur score pour le mode choisi

```
                        pas_encore_trouve_best_mode = False
                        best_mode = liste[i]["score"]
                i += 1
```

Si aucune valeur n'a été trouvée (aucune partie dans le même mode dans la DB ou DB incorrecte)

```
            if not best_mode:
```

On met `"..."` à la place du meilleur score dans le mode en cours

```
                best_mode = "..."
```

On recherche enfin le meilleur score pour le pseudo en cours

```
            pas_encore_trouve_best_perso = True
            i = 0
            best_perso = None
```

Tant que l'on ne l'a pas trouvé et que la liste n'est oas finie

```
            while pas_encore_trouve_best_perso and i < len(liste) - 1:
```

Si le pseudo ne commence pas par `"#"`

```
                if liste[i]["pseudo"][0] != "#":
```

Si le pseudo du score examiné correspond au pseudo de l'utilisateur

```
                    if str(liste[i]["pseudo"]) == self.pseudo:
```

On considère que c'est le meilleur score de l'utilisateur

```
                        pas_encore_trouve_best_perso = False
                        best_perso = liste[i]["score"]
                i += 1
```

Si aucune valeur n'a été trouvée (le joueur n'a jamais joué ou la DB est incorrecte)

```
            if not best_perso:
```

On met `"..."` à la place du meilleur score du joueur

```
                best_perso = "..."
```

Enfin, on retourne les 3 valeurs de scores calculées

```
            return (best, best_mode, best_perso)
```

Si une exception a été levée (pas de DB, ou DB incorrecte, ou aucun score dans la DB)

```
        except:
```

On retourne une liste de 3 chaînes `"..."`

```
            return ("...", "...", "...")

    def modeTexteEnh(self):
        mode_texte_enh = ""
        mds = self.mode_texte.split("::")
        if mds[0] == "expl":
            mode_texte_enh = "Texte d'exemple {}"\
                .format(mds[1])
        elif mds[0] == "syll":
            mode_texte_enh = "Mots avec la syllabe -{}"\
                .format(mds[1])
        elif mds[0] == "mots_fr":
            mode_texte_enh = "Les {} mots les plus courants"\
                .format(mds[1])
        elif mds[0] == "perso":
            if mds[1] == "nom":
                mode_texte_enh = "Texte personnalisé : \"{}\""\
                    .format(mds[2][3:-3].split("/")[-1])
            elif mds[1] == "entier":
                mode_texte_enh = "Texte personnalisé : {}..."\
                    .format(mds[2][3:-3][:min(10, len(mds[2][3:-3]))])
        else:
            mode_texte_enh = "Inconnu"
        return mode_texte_enh

    def keyPressEvent(self, event):
        if type(event) == QKeyEvent:
            if event.key() == Qt.Key_Escape:
                self.quitterM()
            elif event.key() == Qt.Key_Return:
                self.togglePauseM()

    def genererTexte(self):
        mtxt_s = self.mode_texte.split("::")
        if mtxt_s[0] == "mots_fr":
            self.texte += (self.genererMotsFR(int(mtxt_s[1]))).decode("utf-8")
        elif mtxt_s[0] == "syll":
            self.texte += (self.genererSyll(mtxt_s[1])).decode("utf-8")
        elif mtxt_s[0] == "expl":
            self.texte += (self.genererExemple(int(mtxt_s[1]))).decode("utf-8")
        elif mtxt_s[0] == "perso":
            if mtxt_s[1] == "nom":
                self.texte += (self.genererNom((mtxt_s[2])[3:-3]))\
                    .decode("utf-8")
            elif mtxt_s[1] == "entier":
                self.texte += (self.genererEntier((mtxt_s[2])[3:-3]))\
```

```
                .decode("utf-8")

    def genererMotsFR(self, nb):
        fichier_mots_brut = open("txt/dictionnaire_freq_e.txt", "r")
        fichier_mots = crypterTexte.decrypterTexte(fichier_mots_brut.read())
        fichier_mots = fichier_mots.split("\n")
        fichier_mots_brut.close()
        liste_mots = [elt for elt in fichier_mots if elt][:nb]
        chaine = ""
        for i in range(25):
            j = randint(0, nb - 1)
            chaine += (liste_mots[j] + " ")
        return chaine

    def genererSyll(self, syll):
        fichier_mots_brut = open("txt/dictionnaire_syll_e.txt", "r")
        fichier_mots = crypterTexte.decrypterTexte(fichier_mots_brut.read())
        fichier_mots = fichier_mots.split("\n")
        fichier_mots_brut.close()
        liste_mots = [elt for elt in fichier_mots if elt and (syll in elt)]
        chaine = ""
        for i in range(25):
            j = randint(0, len(liste_mots) - 1)
            chaine += (liste_mots[j] + " ")
        return chaine

    def genererExemple(self, no):
        exec("fichier_brut = open(\"txt/exemple{}_e.txt\", \"r\")".format(no))
        fichier = crypterTexte.decrypterTexte(fichier_brut.read())
        fichier = fichier.split("\n")
        texte = ""
        for ligne in fichier:
            if ligne:
                if ligne[0] != "#":
                    texte += (ligne + "\n")
        return (self.normaliserTexte(texte) + " ")

    def genererNom(self, nom):
        fichier_brut = open("txt/" + nom, "r")
        fichier = fichier_brut.read()
        fichier_brut.close()
        return (self.normaliserTexte(fichier) + " ")

    def genererEntier(self, texte):
        return (self.normaliserTexte(texte) + " ")
```

```
    def normaliserTexte(self, texte):
        texte = (re.sub(r"[\n\t\b\a\r]", r" ", texte)).strip()
        texte = (re.sub(r" {2,}", r" ", texte.strip()))
        if len(texte) > 4096:
            texte = texte[:4096]
        return texte
```

Méthode (slot) `getDerCar`

Méthode permettant de récupérer le caractère tapé dans la boîte de texte suite à un signal `textChanged`

```
    @pyqtSlot(str)
    def getDerCar(self, ligne_tapee):
```

On récupère le caractère tapé, on vide la boîte et appelle la méthode `interpreterDerCar` pour interpréter le caractère tapé

```
        self.der_car_T = ligne_tapee
        self.interpreterDerCar()
        self.EntryTapeCentre.clear()
```

Si `jeton_pauseM` vaut `True` (le programme était en pause ou pas encore commencé et l'utilisateur a tapé une lettre), on appelle la méthode `togglePauseM` pour désactiver la pause

```
        if self.jeton_pauseM:
            self.togglePauseM()
```

Méthode `interpreterDerCar`

Méthode permettant d'interpréter le caractère tapé à la suite de la méthode `getDerCar`

```
    def interpreterDerCar(self):
```

On vérifie que le texte tapé n'est pas nul (car les méthodes `getDerCar` et `interpreterDerCar` se déclenchent après le `clear` de la boîte)

```
        if self.der_car_T != "":
```

On calcule les caractères normalisés

```
car_attendu_dec = unicodedata.normalize('NFKD', self.car_attendu)
car_attendu_norm = car_attendu_dec.encode('ascii', 'ignore')
car_attendu_norm = unicode(car_attendu_norm)
```

Si le caractère tapé est bien le caractère attendu :
On appelle la méthode `decalerTexte`, on ajoute 1 aux caractères justes et on met en vert les flèches (méthode `vert`)

```
self.der_car_T = self.der_car_T[0]
if self.der_car_T == self.car_attendu:
    self.coeff_pre = 1.0
    self.dernier_juste = True
    self.car_attendu_precedant = self.car_attendu
    self.car_justes += 1
    self.vert()
    self.decalerTexte()
# À faire
elif self.der_car_T == "$":
    self.coeff_pre = 0.0
    self.dernier_juste = True
    self.car_attendu_precedant = self.car_attendu
    self.LabelTapeFleche.setStyleSheet("")
    self.decalerTexte()
# Si caractère normalisé :
elif self.der_car_T == self.car_attendu.lower() or\
    self.der_car_T == car_attendu_norm or\
        self.der_car_T == car_attendu_norm.lower():
    self.coeff_pre = 0.25
    self.dernier_juste = True
    self.car_attendu_precedant = self.car_attendu
    self.vert()
    self.decalerTexte()
```

Sinon :
On ajoute 1 aux caractères faux et on met en rouge les flèches (méthode `rouge`)

```
else:
    self.dernier_juste = False
    self.car_faux += 1
    self.rouge()
```

Enfin, on appelle la méthode `genererStats` pour mettre à jour les statistiques

```
self.genererStats()
```

Méthode `vert`

Méthode permettant de mettre en vert les flèches (`LabelTapeFleche`)

```
    def vert(self):
```

On définit la couleur de police à `green`

```
        self.LabelTapeFleche.setStyleSheet("color: green")
```

Méthode `rouge`

Méthode permettant de mettre en rouge les flèches (`LabelTapeFleche`)

```
    def rouge(self):
```

On définit la couleur de police à `red`

```
        self.LabelTapeFleche.setStyleSheet("color: red")
```

Méthode `decalerTexte`

Méthode permettant de décaler le texte (au niveau des variables)

```
    def decalerTexte(self):
```

On avance de 1 la variable `pos_texte` ;
On actualise les variables `texte_d`, `texte_g` et `car_attendu` en fonction de la nouvelle valeur de `pos_texte`

```
        self.pos_texte += 1
        self.texte_d = self.texte[:self.pos_texte]
        self.car_attendu = self.texte[self.pos_texte]
        self.texte_g = self.texte[(self.pos_texte + 1):]
```

Si on est bientôt à cours de texte dans la partie droite (`texte_g`), on double le texte (on le reboucle sur lui-même)

```
        if (len(self.texte) - self.pos_texte) <= 23:
            self.genererTexte()
```

Enfin, on met à jour les labels

```
        self.updateTexteLabel()
```

Méthode `updateTexteLabel`

Méthode permettant de mettre à jour le texte des labels du GUI (et donc décaler le texte au niveau du GUI)

```
    def updateTexteLabel(self):
```

La variable `texte_aff_droite` correspond à `texte_d` recoupé si besoin à la longueur maximum du label (22 caractères)

```
        texte_aff_droite = self.texte_d
        if len(texte_aff_droite) > 22:
            texte_aff_droite = texte_aff_droite[-22:]
```

La variable `texte_aff_centre` correspond au caractère attendu

```
        texte_aff_centre = self.car_attendu
```

La variable `texte_aff_gauche` correspond à `texte_g` recoupé si besoin à la longueur maximum du label (22 caractères)

```
        texte_aff_gauche = self.texte_g
        if len(texte_aff_gauche) > 22:
            texte_aff_gauche = texte_aff_gauche[:22]
```

La variable `texte_aff_basdroite` correspond à `texte_d` recoupé si besoin à la longueur maximum du label (9 caractères)

```
        texte_aff_basdroite = self.texte_d
        if len(texte_aff_basdroite) > 9:
            texte_aff_basdroite = texte_aff_basdroite[-9:]
```

Ensuite, on met à jour les labels avec les nouvelles valeurs des variables évoqués ci-dessus

```
        self.LabelTexteDroite.setText(texte_aff_droite)
        self.LabelTexteCentre.setText(texte_aff_centre)
        self.LabelTexteGauche.setText(texte_aff_gauche)
        self.LabelTapeDroit.setText(texte_aff_basdroite)
```

Méthode (slot) `togglePauseM`

Méthode permettant d'activer/désactiver la pause

```
    @pyqtSlot()
    def togglePauseM(self):
```

Si le temps est fini, le bouton start/pause permet recommencer (méthode `recommencer`)

```
        if self.jeton_temps_finiM:
            self.recommencer()
```

Si le temps n'est pas fini, et que c'est le premier lancement :

```
        elif self.premier_lancement_timer:
```

On désactive la pause (`jeton_pauseM`) et le drapeau de premier lancement (`premier_lancement_timer`)

```
            self.premier_lancement_timer = False
            self.jeton_pauseM = False
```

Pour le premier caractère, on prend un temps arbitraire de 0,5 s

```
            self.temps_score_precedant = time() - 0.5
```

On lance ensuite le timer pour la première fois ;

```
            self.Timer.start()
```

On change le texte du bouton start/pause ;
On active les différents labels désactivés lors du lancement ; Et on active le focus sur la boîte de texte

```
            self.BoutonStartPause.setText(u"Pause")
            self.LabelTexteDroite.setEnabled(True)
            self.LabelTexteCentre.setEnabled(True)
            self.LabelTexteGauche.setEnabled(True)
            self.LabelTapeDroit.setEnabled(True)
            self.EntryTapeCentre.setFocus()
            self.LabelTapeFleche.setEnabled(True)
```

Si le temps n'est pas fini et que ce n'est pas le premier lancement :

```
        else:
```

Si `jeton_pause_M` vaut `False` (pas de pause), on lance la pause (méthode `pauseM`)

```
            if not self.jeton_pauseM:
                self.pauseM()
```

Sinon (pause active), on désactive la pause (méthode `reprendreM`)

```
            elif self.jeton_pauseM:
                self.reprendreM()
```

Méthode `pauseM` Méthode permettant de mettre en pause le GUI

```
    def pauseM(self):
```

On appelle la méthode `pauseT` du timer pour le mettre en pause

```
        self.Timer.pauseT()
```

On change le texte du bouton start/pause et on active la pause en passant `jeton_pauseM` à `True`

```
        self.BoutonStartPause.setText(u"Reprendre")
        self.jeton_pauseM = True
```

On désactive ensuite les différents labels en gardant le focus sur la boîte de texte

```
        self.LabelTexteDroite.setEnabled(False)
        self.LabelTexteCentre.setEnabled(False)
        self.LabelTexteGauche.setEnabled(False)
        self.LabelTapeDroit.setEnabled(False)
        self.LabelTapeFleche.setEnabled(False)
        self.EntryTapeCentre.setFocus()
```

Enfin, on récupère la couleur actuelle des flèches que l'on sauvegarde, on on met les flèches en couleur par défaut (noir)

```
        self.couleur_backup = self.LabelTapeFleche.styleSheet()
        self.LabelTapeFleche.setStyleSheet("")
```

Méthode `reprendreM`

Méthode permettant de reprendre après une pause du GUI

```
    def reprendreM(self):
```

On appelle la méthode `reprendreT` du timer pour enlever la pause du timer

```
        self.Timer.reprendreT()
```

On change le texte du bouton start/pause et on désactive la pause en passant `jeton_pauseM` à `False`

```
        self.BoutonStartPause.setText(u"Pause")
        self.jeton_pauseM = False
```

On réactive les labels précédemments désactivés durant la pause en gardant le focus sur la boîte de texte

```
        self.LabelTexteDroite.setEnabled(True)
        self.LabelTexteCentre.setEnabled(True)
        self.LabelTexteGauche.setEnabled(True)
        self.LabelTapeDroit.setEnabled(True)
        self.EntryTapeCentre.setFocus()
        self.LabelTapeFleche.setEnabled(True)
```

On remet la couleur que les flèches avaient lors de la mise en pause grâce à la valeur sauvegardée

```
        self.LabelTapeFleche.setStyleSheet(self.couleur_backup)
```

Méthode (slot) `quitterM`

Méthode permettant de quitter proprement le programme en fermant d'abord le timer

```
    @pyqtSlot()
    def quitterM(self):
```

On appelle la méthode `quitterT` du timer pour le fermer, et on attend qu'il se ferme

```
        self.Timer.quitterT()
        self.Timer.wait()
        self.termine.emit(self.recommencerV)
```

Enfin, on ferme le programme

```
        self.close()
```

Méthode (slot) `temps_change`

Méthode permettant de mettre à jour le temps affiché lors de l'émission du signal `temps_change_signal`

```
    @pyqtSlot(float)
    def temps_change(self, temps_restant):
```

On récupère la valeur de `temps_restant` portée par le signal qui appel ce slot (cette méthode)

```
        self.temps_restant = temps_restant
```

On met alors à jour l'affichage du temps restant et la barre d'avancement

```
        self.LabelRestantV.setText(unicode(
                                   "{} / {}"
                                   .format(round(self.temps_restant, 1),
                                           round(self.temps_choisi, 1))))
        self.BarreAvancement.setValue(int(round((self.temps_restant /
                                                 self.temps_choisi) * 100, 0)))
```

Méthode (slot) `temps_fini`

Méthode appelée lorsque le temps est fini et permettant de paramètrer le GUI pour un enventuel nouveau lancement (si l'utilisateur recommence)

```
    @pyqtSlot()
    def temps_fini(self):
```

On désactive tous les labels et on remet la couleur des flèches par défaut (noir)

```
        print("Temps fini")
        self.LabelTexteDroite.setEnabled(False)
        self.LabelTexteCentre.setEnabled(False)
        self.LabelTexteGauche.setEnabled(False)
        self.LabelTapeDroit.setEnabled(False)
        self.EntryTapeCentre.setEnabled(False)
        self.LabelTapeFleche.setStyleSheet("")
        self.LabelTapeFleche.setEnabled(False)
```

On met la variable `jeton_temps_finiM` à `True` et on appelle la méthode `setUpRecommencer` pour repréparer le GUI pour un nouveau lancement

```
        self.jeton_temps_finiM = True
        self.setUpRecommencer()

        print("Appel gerer score")
        self.gererScore()
```

Méthode `setUpRecommencer`

Méthode permettant de reparamétrer le GUI pour un nouveau lancement

```
    def setUpRecommencer(self):
```

On change le texte du bouton start/pause

```
        self.BoutonStartPause.setText(u"Recommencer")
```

Méthode `recommencer`

Méthode permettant de recommencer

```
    def recommencer(self):
```

On passe la valeur de `recommencerV` à `True`
On appelle ensuite la méthode `quitterM` permettant de quitter

```
        self.recommencerV = True
        self.quitterM()
```

Méthode genererStats

Méthode permettant de générer les statistiques

```
    def genererStats(self):
```

On appelle les différentes méthodes en charge des statistiques

```
        self.temps_ecoule = self.temps_choisi - self.temps_restant
        if not self.temps_ecoule == 0:
            self.compterMots()
            self.compterCar()
            self.compterJusteErreur()
            self.compterScore()
```

Méthode compterMots

Méthode permettant de compter le nombre de mots tapés et de calculer ensuite le temps moyen mis pour taper un mot (`mots_min`)

```
    def compterMots(self):
```

On calcule le nombre de mots tapés à partir de la valeur de `texte_d`

```
        texte_mod = self.texte_d.replace("'", " ")
        nombre_mots = len((texte_mod).split(" ")) - 1
```

On définit le nombre de mots tapés comme étant supérieur à 1

```
        if nombre_mots > self.nombre_mots_precedant:
```

On change l'ancienne valeur de `nombre_mots_precedant`

```
            self.nombre_mots_precedant = nombre_mots
```

On calcule le nombre de mots tapés par minite et on l'affiche dans l'interface

```
            self.mots_min = nombre_mots / (self.temps_ecoule / 60.0)
            self.LabelMotsMinV.setText(unicode(str(round(self.mots_min, 1))))
```

Méthode `compterCar`

Méthode permettant de compter et d'afficher le nombre de caractères tapés à la minute (`car_min`)

```
    def compterCar(self):
```

On calcule le nombre de caractères tapés depuis la valeur `texte_d`

```
        nombre_car = len(self.texte_d)
        self.car_min = nombre_car / (self.temps_ecoule / 60.0)
```

On affiche le nombre de caractères tapés par minute, arrondi à l'entier le plus proche

```
        self.LabelCarMinV.setText(unicode(str(int(round(self.car_min, 0)))))
```

Méthode `compterJusteErreur`

Méthode permettant de calculer et d'afficher dans les barres horizontales le pourcentage de caractères justes et faux (réussite et erreurs)

```
    def compterJusteErreur(self):
```

On définit la somme, supérieure à 1, des caractères justes et faux

```
        somme = self.car_justes + self.car_faux
        if somme == 0:
            somme = 1
```

On calcule les ratios de caractères justes et faux en fonction de la somme calculée précedemment

```
        self.reussite = self.car_justes / somme
        self.erreurs = self.car_faux / somme
```

Enfin, on affiche dans l'interface (sur les barres horizontales) les deux valeurs que l'on vient de calculer

```
        self.BarreReussite.setValue(round(self.reussite * 100, 0))
        self.BarreErreurs.setValue(round(self.erreurs * 100, 0))
```

Méthode `compterScore`

Méthode permettant de calculer le score selon la nouvelle formule
À détailler

```
    def compterScore(self):
        if self.dernier_juste:
```

On recherche le type de caractères

```
            if self.der_car_T == " ":
                coeff = 0.4
            elif re.match(r"[a-z]", self.der_car_T):
                coeff = 0.5
            elif re.match(r"[A-Z]", self.der_car_T):
                coeff = 0.8
            elif re.match(r"[²&é\"'(-è_çà)=,;:!<]", self.der_car_T):
                coeff = 1
            elif re.match(r"[0-9°+?./§>]", self.der_car_T):
                coeff = 1.2
            else:
                coeff = 1.6
```

On calcule le temps et les différents facteurs

```
            temps_score = time()
            temps_ecoule_l = temps_score - self.temps_score_precedant
            inv_temps_pour_car = 1 / (temps_ecoule_l)
            self.temps_score_precedant = temps_score
            nombre_erreur_pour_car = self.car_faux -\
                self.car_faux_precedant
            self.car_faux_precedant = self.car_faux
            inv_de_err_plus_un = 1 / (nombre_erreur_pour_car + 1)
            ln_tps_plus_C_div_tps = (log(self.temps_choisi) +
                                     self.C_TEMPS) / self.temps_choisi
```

On calcule le score final et on l'affiche

```
            score_car = inv_temps_pour_car * inv_de_err_plus_un * \
                ln_tps_plus_C_div_tps * coeff * self.C_SCORE * self.coeff_pre
            self.score += score_car
            self.LabelScoreV.setText(unicode(str(int(round(self.score,
                                                           0)))))
```

```
def gererScore(self):
    print("Début Gérer score")
    print("Gestion temps")
    dh = localtime()
    AAAA = str(dh[0])
    print("Année")
    while len(AAAA) < 4:
        AAAA = "0" + AAAA
    MM = str(dh[1])
    print("Mois")
    while len(MM) < 2:
        MM = "0" + MM
    print("jour")
    JJ = str(dh[2])
    while len(JJ) < 2:
        JJ = "0" + JJ
    print("Heure")
    hh = str(dh[3])
    while len(hh) < 2:
        hh = "0" + hh
    print("minute")
    mm = str(dh[4])
    while len(mm) < 2:
        mm = "0" + mm
    print("seconde")
    ss = str(dh[5])
    while len(ss) < 2:
        ss = "0" + ss
    print("Fin gestion temps debut gestion mode")
    mds = self.mode_texte.split("::")
    if mds[0] == "expl":
        mode_texte_enh = "Texte d'exemple {}"\
            .format(mds[1])
    elif mds[0] == "syll":
        mode_texte_enh = "Mots avec la syllabe -{}"\
            .format(mds[1])
    elif mds[0] == "mots_fr":
        mode_texte_enh = "Les {} mots les plus courants"\
            .format(mds[1])
    elif mds[0] == "perso":
        if mds[1] == "nom":
            mode_texte_enh = "Texte personnalisé : \"{}\""\
                .format(mds[2][3:-3].split("/")[-1])
        elif mds[1] == "entier":
            mode_texte_enh = "Texte personnalisé : {}..."\
                .format(mds[2][3:-3][:min(10, len(mds[2][3:-3]))])
```

```
        else:
            mode_texte_enh = "Inconnu"
        print("Fin mode début dico")
        dico_score = {"pseudo": self.pseudo,
                      "score": int(round(self.score, 0)),
                      "cpm": round(self.car_min, 1),
                      "mpm": round(self.mots_min, 1),
                      "temps": int(self.temps_choisi),
                      "d_h": "{}-{}-{} {}:{}:{}".format(AAAA, MM, JJ, hh, mm,
                                                        ss),
                      "texte_mode": self.mode_texte,
                      "texte_t": self.texte_d.encode("utf-8"),
                      "texte_mode_enh": mode_texte_enh}
        print("--> Dico généré")
        self.stockerLocalScore(dico_score)
        print("--> Local Score fait")
        crypterScore.crypterScoreAttente(dico_score)
        print("--> En Attente Score fait")
        adresse = crypterScore.envoyerScoreAttente()
        print("--> Envoyer fait")
        self.affFenetreScore(dico_score, adresse)

    def stockerLocalScore(self, dico_score):
        dico_score_raccourci = {"pseudo": dico_score["pseudo"],
                                "score": dico_score["score"],
                                "cpm": dico_score["cpm"],
                                "mpm": dico_score["mpm"],
                                "temps": dico_score["temps"],
                                "d_h": dico_score["d_h"],
                                "texte_mode_enh": dico_score["texte_mode_enh"]}
        try:
            fichier_db = open("score/local_db.db", "rb")
            mon_pickler = pickle.Unpickler(fichier_db)
            try:
                liste = mon_pickler.load()
            except:
                liste = []
            finally:
                fichier_db.close()
        except:
            liste = []
        liste.append(dico_score_raccourci)
        liste = sorted(liste, key=lambda dico: dico["score"], reverse=True)
        fichier_db = open("score/local_db.db", "wb")
        mon_pickler = pickle.Pickler(fichier_db)
        mon_pickler.dump(liste)
```

```
        fichier_db.close()

    def termineScore(self):
        del self.Score
        self.show()

    def affFenetreScore(self, dico_score, adresse):
        print("INIT")
        self.Score = ScoreApplication(dico_score, adresse)
        print("Créée")
        self.Score.termine.connect(self.termineScore)
        print("Bind signal")
        self.Score.setWindowModality(Qt.ApplicationModal)
        print("Modal")
        self.hide()
        print("Module caché")
        self.Score.show()
        print("Affiché")
```

Classe `ThreadTimer`

La classe `ThreadTimer`, héritée de `QThread`, permet de lancer un timer en thread d'arrière plan, qui fonctionne tout seul (standalone)
On peut intéragir avec le timer grâce aux méthodes `pauseT`, `reprendreT` et `quitterT`

```
class ThreadTimer(QThread):
```

En-tête de la classe

On créé ici les signaux `pyqtSignal` permettant d'intéragir avec le GUI Ces signaux seront ensuite connectés au GUI avec la méthode `connect`

```
    temps_fini_signal = pyqtSignal()
    temps_change_signal = pyqtSignal(float)
    finished = pyqtSignal()
```

Méthode d'initialisation `__init__` Méthode permettant d'initialiser la classe "‘python def __init__(self, temps_choisi=60): "‘ On hérite de la méthode**init**de la classe parente (**QThread**) "‘python QThread.__init__(self) "‘ On créé les attributs de la classe "‘python self.temps_choisi = temps_choisi self.temps_depart = 0.0 self.temps_inter = 0.0 self.temps_ecoule = 0.0 self.temps_restant = 0.0 self.jeton_quitter = False self.jeton_pause = True self.temps_debut_pause = 0.0 self.temps_fin_pause = 0.0 "‘ ###Méthode principale**run**Cette méthode correspond au corps du thread, qui est appelée lors du**start()**, et dont la fin correspond à la fin de l'execution du thread "‘python def run(self): "‘ On prend le temps lors du lancement et on désactive la pause "‘python self.temps_depart = time() self.jeton_pause = False "‘ Tant que**jeton_quitter**est**False**(tant que l'on ne veut pas quitter) On calcule le temps restant "‘python while not self.jeton_quitter: self.temps_inter = time() self.temps_ecoule = self.temps_inter - self.temps_depart self.temps_restant = self.temps_choisi - self.temps_ecoule "‘ Si il est négatif, on le met à**0**, on met une dernière fois à jour le temps (signal**temps_change**), et on envoie un signal pour dire que le temps est fini (signal**temps_fini**) Enfin, on termine la méthode (**return**) "‘python if self.temps_restant <= 0.0: self.temps_restant = 0.0 self.temps_change_signal.emit(self.temps_restant) self.temps_fini_signal.emit() return "‘ Sinon, on met à jour le temps (signal**temps_change**), et on fait hiberner le programme pendant**0,01**s "‘python else: self.temps_change_signal.emit(self.temps_restant) sleep(0.01) "‘ Si la pause est activée, on prend le temps de début de pause Ensuite, tant que la pause est activée et que le timer ne doit pas être quitté, le programme hiberne par pas de 0,01 s "‘python if self.jeton_pause: self.temps_debut_pause = time() while self.jeton_pause and not self.jeton_quitter: sleep(0.01) "‘ Quand on sort de la boucle (pause terminée), on prend le temps de fin de pause, on calcule le temps passé en pause, et on ajoute cette durée au temps de lancement (**temps_depart**) "‘python self.temps_fin_pause = time() self.temps_pause_ecoule = (self.temps_fin_pause - self.temps_debut_pause) self.temps_depart += self.temps_pause_ecoule "‘ Quand la boucle est cassée (quand**jeton_quitter**vaut**True**), on émet un signale**finished**(utile pour la destruction au bon moment du thread) "‘python self.finished.emit() "‘ ###Méthode de pause**pauseT**Cette méthode permet de mettre en pause le thread, en modifiant la valeur de l'attribut**jeton_pause**de**False**à**True**"‘python def pauseT(self): self.jeton_pause = True "‘ ###Méthode de pause**reprendreT**Cette méthode permet de reprendre le thread après une pause, en modifiant la valeur de l'attribut**jeton_pause**de**True**à**False**"‘python def reprendreT(self): self.jeton_pause = False "‘ ###Méthode permettant de quitter le timer**quitterT**Cette méthode permet de quitter le thread en modifiant la valeur de l'attribut**jeton_quitter**de**False**à**True**Cela casse la boucle principale de la méthode**run**du thread "‘python def quitterT(self): self.jeton_quitter = True "‘ ##Classe**ScoreApplication**Cette classe hérite des classes**QWidget**et**Ui_Score**et permet la création du GUI et sa gestion Cette classe contient la majeure partie du programme de la fenêtre des scores Elle est directement issue de *Qt* et (et donc**PyQt**) "‘python class ScoreApplication(QWidget, Ui_Score): termine = pyqtSignal() "‘ ###Méthode d'initialisation**init**Méthode permettant d'initialiser la classe "‘python def __init__(self,

```
    def genererHTMLDB(self, dico_score):
```

On récupère les pseudo et score courants pour afficher en rouge la ligne correspondant à la partie en cours

```
        pseudo_cur = dico_score["pseudo"]
        score_cur = dico_score["score"]
```

On définit le code HTML de la page

```
        c = ""
        c +=\
            """<html lang="fr">
    <head>
        <meta charset="UTF-8" />
        <style>
            *
            {
                font-family: "Comic sans MS", Verdana, sans-serif;
                font-size: 14px;
            }

            #normal
            {

            }

            #best
            {
                background-color: yellow;
            }

            #self
            {
                background-color: red;
            }

            #cat
            {
                background-color: gray;
            }
        </style>

        <title>Scores IGGF</title>
    </head>
```

```
    <body>
"""
        c +=\
            """         <table>
"""
```

On ouvre la DB locale
La DB locale est forcément définie car elle vient d'être créée si elle n'existait pas lors de la fin de la partie

```
        fichier_db = open("score/local_db.db", "rb")
```

On récupère le contenu sous forme d'une liste de dictionnaires grâce à un pickler

```
        mon_pickler = pickle.Unpickler(fichier_db)
        liste = mon_pickler.load()
        fichier_db.close()
```

On gère le type de ligne

```
        pas_encore_trouve_best = True
```

Pour chaque élément dans la liste

```
        for elt in liste:
            type_ligne = "normal"
```

Si l'élément commence par un `"#"`, alors la ligne correspond à une catégorie

```
            if str(elt["pseudo"])[0] == "#":
                type_ligne = "cat"
```

On prend le premier score n'étant pas une catégorie, que l'on définit comme le record

```
            elif pas_encore_trouve_best:
                type_ligne = "best"
                pas_encore_trouve_best = False
```

Sinon si le score et le pseudo de la ligne correspondent à ceux de la partie qui vient d'être faîte, on définit le score comme le sien

```
            elif (str(elt["pseudo"]) == str(pseudo_cur) and
                  str(elt["score"]) == str(score_cur)):
                type_ligne = "self"
```

Sinon, la ligne est du type normal

```
            else:
                type_ligne = "normal"
```

On ajoute alors au fichier HTML la ligne correspondante dans le tableau

```
            nl = """            <tr id={}>
                <td>{}</td>
                <td>{}</td>
                <td>{}</td>
                <td>{}</td>
                <td>{}</td>
                <td>{}</td>
                <td>{}</td>
            </tr>
"""\
                .format(str(type_ligne),
                        str(elt["pseudo"]),
                        str(elt["score"]),
                        str(elt["cpm"]),
                        str(elt["mpm"]),
                        str(elt["temps"]),
                        str(elt["d_h"]),
                        str(elt["texte_mode_enh"]))
            c += nl
```

Enfin, on finit d'écrire le fichier HTML

```
        c += """        </table>
    </body>
</html>"""
```

On écrit alors le code HTML dans un fichier

```
        fichier_html = open("score/fichier_html.html", "w")
        fichier_html.write(c)
        fichier_html.close()
```

Et on affiche le code HTML dans la zone prévue

```
        self.ViewScore.setHtml(c.decode("utf-8"))
```

Méthode (slot) `quitter`

Méthode permettant de quitter la fenêtre des scores

```
    @pyqtSlot()
    def quitter(self):
```

On émet le signal `termine`

```
        self.termine.emit()
```

On ferme la fenêtre

```
        self.close()
```

Programme principal

Fonction `main`

Fonction ne prenant pas d'arguments et permettant de créer l'interface et de la lancer

```
def main():
```

On créé une application *Qt* `Qapplication`, pour porter notre GUI

```
    app = QApplication(sys.argv)
```

On créé notre GUI comme étant une instance de la classe `MenuApplication` décrite plus haut

```
    myapp = MenuApplication()
    # #.On affiche notre GUI et on connecte sa fermeture à la fermeture du
    # #.programmme
    myapp.affFenetreBVN()
    app.exec_()

    del myapp
    del app
```

Test de lancement standalone

Test permettant de lancer le programme si il est exécuté directement tout seul, sans import

```
if __name__ == "__main__":
```

On appelle la fonction `main` définit plus haut

```
    main()
```