Import des modules

Import de `__future__.division` pour la division décimale même sur les `int`

```
from __future__ import division
```

Import des bibliothèques pour `PyQt` (interface graphique)

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

Import de la fenêtre graphique designée avec *Qt Creator*

```
from ui_module import Ui_Module
```

Import des bibliothèques standards de `python`

```
from time import time, sleep
from os import system, popen
from math import log
import sys
```

Import de la biblothèque `re` pour l'utilisation d'expressions régulières

```
import re
```

Import de la bibliothèque `atexit` nécessaire pour la création du `.exe`

```
import atexit
```

```
from random import randint
```

Déclaration des classes

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

```
def __init__(self, temps_choisi=60):
```

On hérite de la méthode `__init__` de la classe parente (`QThread`)

```
    QThread.__init__(self)
```

On créé les attributs de la classe

```
    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

```
def run(self):
```

On prend le temps lors du lancement et on désactive la pause

```
    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

```
        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`)

```
            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 `0,1` s

```
            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 `0,1` s

```
            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`)

```
                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)

```
        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`

```
    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`

```
    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

```
    def quitterT(self):
        self.jeton_quitter = 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()

        ##.---------------------------------------------------------------------
        # #.*A remplacer : Ceci sera ensuite remplacé par le menu !*
        # #.On ouvre le fichier de configuration `module.conf`
        # fichier_conf_brut = open("module.conf", "r")
        # #.On lit le fichier et on récupère les paramètres suivants :
        # #.- Temps choisi
        # #.- Nom (ou chemin) du fichier qui contient le texte à taper
        # fichier_conf = fichier_conf_brut.readlines()
        # self.temps_choisi = float((fichier_conf[1])[:-1])
        # nom_fichier_texte = (fichier_conf[3])[:-1]
        # #.On ouvre ensuite le fichier qui contient le texte à taper
        # fichier_texte_brut = open(nom_fichier_texte, "r")
        # self.texte = fichier_texte_brut.read().decode("utf-8")
        ##.---------------------------------------------------------------------

        # #.On enlève les retours à la ligne (remplacés par des espaces) et les
        # #.doubles espaces de ce texte, impossible ou problématiques à taper
        # #.pour l'utilisateur
        # self.texte = (re.sub(r"\n", r" ", self.texte)).strip()
        # self.texte = re.sub(r" {2,}", r" ", self.texte)
```

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.historique_tape = []
self.car_attendu_precedant = ""
self.joker = False
```

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

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

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()

    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":
                texte_temp = ((mtxt_s[2])[3:-3]).strip()
                texte_temp = (re.sub(r"\n", r" ", texte_temp)).strip()
                texte_temp = re.sub(r" {2,}", r" ", texte_temp).strip()
                if len(texte_temp) > 4096:
                    texte_temp = texte_temp[:4096]
                self.texte += (texte_temp + " ").decode("utf-8")

        assert self.texte

    def genererMotsFR(self, nb):
        fichier_mots_brut = open("dictionnaire_freq.txt", "r")
        fichier_mots = fichier_mots_brut.readlines()
        fichier_mots_brut.close()
        liste_mots = ([elt[:-1] for elt in fichier_mots if elt and
                       elt != "\n"])[: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("dictionnaire_syll.txt", "r")
        fichier_mots = fichier_mots_brut.readlines()
        fichier_mots_brut.close()
        liste_mots = [elt[:-1] for elt in fichier_mots if elt and
                      elt != "\n" 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(\"exemple{}.txt\", \"r\")".format(no))
        fichier = ([elt[:-1] for elt in fichier_brut.readlines() if elt and
                    elt != "\n"])[1]
        fichier_brut.close()
        if len(fichier) > 4096:
            fichier = fichier[:4096]
        return (fichier + " ")

    def genererNom(self, nom):
        # Fonctionne que sous linux pour vérifier l'existence du fichier
        #assert (popen("ls | grep \"{}\"".format(nom))).read()
        fichier_brut = open(nom, "r")
        # fichier = ([elt[:-1] for elt in fichier_brut.readlines() if elt and
        #             elt != "\n"])[0]
        # fichier_brut.close()
        fichier = fichier_brut.read()
        fichier = (re.sub(r"\n", r" ", fichier)).strip()
        fichier = re.sub(r" {2,}", r" ", fichier).strip()
        if len(fichier) > 4096:
            fichier = fichier[:4096]
        return (fichier + " ")
```

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 != "":
```

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.joker = False
                self.dernier_juste = True
                self.car_attendu_precedant = self.car_attendu
                self.car_justes += 1
                self.vert()
                self.decalerTexte()
            elif self.der_car_T == "$":
                self.dernier_juste = True
                self.car_attendu_precedant = self.car_attendu
                self.LabelTapeFleche.setStyleSheet("")
                self.joker = True
                self.decalerTexte()
```

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

```
            else:
                self.joker = False
                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
            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)

```
        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()
```

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 not self.joker:
        if self.dernier_juste:
            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)
```

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
            ln_tps_plus_C_div_tps = (log(self.temps_choisi) +
                                     self.C_TEMPS) / self.temps_choisi
            score_car = inv_temps_pour_car * inv_de_err_plus_un * \
                ln_tps_plus_C_div_tps * coeff * self.C_SCORE
            self.score += score_car
            self.historique_tape.append((round(temps_ecoule_l, 2),
                                         self.car_attendu_precedant
                                         .encode("utf8"),
                                         str(self.der_car_T.toUtf8()),
                                         round(score_car, 2),
                                         nombre_erreur_pour_car))
```

On affiche le score arrondi à l'entier le plus proche dans le GUI

```
            self.LabelScoreV.setText(unicode(str(int(round(self.score,
                                                           0)))))
```