À faire :

- Commenter le code
- Remettre au clair certaines parties du code
- Faire notre menu, remodifier le GUI, le programmer, le binder
- Changer s/mots en mots/min (revoir le système de stats)
- Proposer un échantillon de textes plus étendu
- Certaines parties du code à refaire / reprécises (ex : doubles espaces)

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 math import log
import sys
```

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

```
import atexit
```

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 fonctions `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):
```

On hérite de la fonction `__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)

```
        while not self.jeton_quitter:
```

On calcule le temps restant

```
            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(0.1)
```

Si la pause est activée, on prend le temps de début de pause

```
            if self.jeton_pause:
                self.temps_debut_pause = time()
```

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

```
                while self.jeton_pause and not self.jeton_quitter:
                    sleep(0.1)
```

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):
```

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(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)
```

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 = (self.texte.replace("\n", " ")).strip()
        self.texte = self.texte.replace("  ", " ")
```

On définit les attributs

```
        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.jeton_pauseM = True
        self.temps_restant = 0.0
        self.premier_lancement_timer = True
        self.couleur_backup = ""
        self.jeton_temps_finiM = False
        self.s_mots = 0.0
        self.temps_ecoule = 0.0
```

```
        self.score = 0.0
        self.car_justes = 0
        self.car_faux = 0
        self.reussite = 0.0
        self.erreurs = 0.0
```

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 fonction en charge de le détruire proprement

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

        # On setup les widgets

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

        # Ici on bind les signaux et les slots

        self.EntryTapeCentre.textChanged.connect(self.getDerCar)
        self.BoutonStartPause.clicked.connect(self.togglePauseM)
        self.BoutonQuitter.clicked.connect(self.quitterM)
        self.Timer.temps_change_signal.connect(self.temps_change)
        self.Timer.temps_fini_signal.connect(self.temps_fini)

        # On disable tant que pas commencé

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

        self.EntryTapeCentre.setFocus()

    # Ici on créer les slots et signaux persos

    @pyqtSlot(str)
    def getDerCar(self, ligne_tapee):
        der_car_T = unicode(ligne_tapee)
        self.EntryTapeCentre.clear()
```

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

    def interpreterDerCar(self, der_car_T):
        if der_car_T != "":
            if der_car_T == self.car_attendu:
                self.decalerTexte()
                self.car_justes += 1
                self.vert()
            else:
                self.car_faux += 1
                self.rouge()

    def vert(self):
        self.LabelTapeFleche.setStyleSheet("color: green")

    def rouge(self):
        self.LabelTapeFleche.setStyleSheet("color: red")

    def decalerTexte(self):
        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):]
        if (len(self.texte) - self.pos_texte) <= 23:
            self.texte += (u" " + self.texte)
            # On met à jour les labels
        self.updateTexteLabel()

    def updateTexteLabel(self):
        texte_aff_droite = self.texte_d
        if len(texte_aff_droite) > 22:
            texte_aff_droite = texte_aff_droite[-22:]
        texte_aff_centre = self.car_attendu
        texte_aff_gauche = self.texte_g
        if len(texte_aff_gauche) > 22:
            texte_aff_gauche = texte_aff_gauche[:22]
        texte_aff_basdroite = self.texte_d
        if len(texte_aff_basdroite) > 9:
            texte_aff_basdroite = texte_aff_basdroite[-9:]
        self.LabelTexteDroite.setText(texte_aff_droite)
        self.LabelTexteCentre.setText(texte_aff_centre)
        self.LabelTexteGauche.setText(texte_aff_gauche)
        self.LabelTapeDroit.setText(texte_aff_basdroite)
```

```
@pyqtSlot()
def togglePauseM(self):
    if self.jeton_temps_finiM:
        self.recommencer()
    elif self.premier_lancement_timer:
        self.premier_lancement_timer = False
        self.jeton_pauseM = False
        self.Timer.start()
        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)
    else:
        if not self.jeton_pauseM:
            self.pauseM()
        elif self.jeton_pauseM:
            self.reprendreM()

def pauseM(self):
    self.Timer.pauseT()
    self.BoutonStartPause.setText("Reprendre")
    self.jeton_pauseM = True
    # DISABLE
    self.LabelTexteDroite.setEnabled(False)
    self.LabelTexteCentre.setEnabled(False)
    self.LabelTexteGauche.setEnabled(False)
    self.LabelTapeDroit.setEnabled(False)
    self.LabelTapeFleche.setEnabled(False)
    self.EntryTapeCentre.setFocus()
    self.couleur_backup = self.LabelTapeFleche.styleSheet()
    self.LabelTapeFleche.setStyleSheet("")

def reprendreM(self):
    self.Timer.reprendreT()
    self.BoutonStartPause.setText("Pause")
    self.jeton_pauseM = False
    self.LabelTexteDroite.setEnabled(True)
    self.LabelTexteCentre.setEnabled(True)
    self.LabelTexteGauche.setEnabled(True)
    self.LabelTapeDroit.setEnabled(True)
    self.EntryTapeCentre.setFocus()
    self.LabelTapeFleche.setEnabled(True)
    self.LabelTapeFleche.setStyleSheet(self.couleur_backup)
```

```
@pyqtSlot()
def quitterM(self):
    self.Timer.quitterT()
    self.Timer.wait()
    self.close()

@pyqtSlot(float)
def temps_change(self, temps_restant):
    self.temps_restant = temps_restant
    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)))
    self.genererStats()

@pyqtSlot()
def temps_fini(self):
    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)
    self.jeton_temps_finiM = True
    self.setUpRecommencer()

def setUpRecommencer(self):
    self.BoutonStartPause.setText("Recommencer")

def recommencer(self):
    # A faire !!
    pass

def genererStats(self):
    # On compte le nombre de mots
    self.compterMots()
    self.compterJusteErreur()
    self.compterScore()

def compterMots(self):
    nombre_mots = len((self.texte_d).split(" ")) - 1
    self.LabelScoreV.setText(unicode(str(nombre_mots)))
```

```
            if nombre_mots <= 1:
                nombre_mots = 1
            self.temps_ecoule = self.temps_choisi - self.temps_restant
            self.s_mots = self.temps_ecoule / nombre_mots
            self.LabelSMotsV.setText(unicode(str(round(self.s_mots, 2))))

        def compterJusteErreur(self):
            somme = self.car_justes + self.car_faux
            if somme == 0:
                somme = 1
            self.reussite = self.car_justes / somme
            self.erreurs = self.car_faux / somme
            # print(erreurs)
            self.BarreReussite.setValue(round(self.reussite * 100, 0))
            self.BarreErreurs.setValue(round(self.erreurs * 100, 0))

        def compterScore(self):
            avancement = self.temps_ecoule / self.temps_choisi
            s_mots_mod = self.s_mots
            if s_mots_mod == 0:
                s_mots_mod = 1
            inv_vitesse = 1 / s_mots_mod

            # Truc rajouté en attendant : on affiche le nombre de mots par minute
            # à la place du label "Meilleurs scores"
            mots_m = (1 / s_mots_mod) * 60
            self.LabelBestT.setText(unicode(str(round(mots_m, 1))))

            erreurs_mod = self.erreurs
            if erreurs_mod < 0.001:
                erreurs_mod = 0.001
            ln_inv_erreurs = log(1 / erreurs_mod)
            ln_temps_plusC = log(self.temps_choisi / 60) + 5.5
            self.score = avancement * inv_vitesse * ln_inv_erreurs *\
                ln_temps_plusC * 100
            self.LabelScoreV.setText(unicode(str(int(round(self.score, 0)))))

    # Programme principal


    def main():  # On mettra des paramètres au main hérités du menu
        app = QApplication(sys.argv)
        myapp = ModuleApplication()
        myapp.show()
        sys.exit(app.exec_())
```

```python
if __name__ == "__main__":
    main()
```