

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 / repréciser (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 désignée 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 interagir avec le timer grâce aux fonctions `pauseT`, `reprendreT` et `quitterT`

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

En-tête de la classe

On crée ici les signaux `pyqtSignal` permettant d'interagir 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 crée 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'exécution 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 signal `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 décrits 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 crée 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 lance une première fois les fonctions updateTexteLabel et temps_change pour régler le GUI sur la position de départ

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

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

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

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

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

Quand on clique sur le bouton quitter, on appelle la fonction `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 fonction 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

```
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 (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 fonction `interpreterDerCar` pour interpréter le caractère tapé

```

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

```

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

```

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 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`)

```

if der_car_T == self.car_attendu:
    self.decalerTexte()
    self.car_justes += 1
    self.vert()

```

Sinon :

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

```

else:
    self.car_faux += 1
    self.rouge()

```


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.texte += (u" " + self.texte)
```

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
```

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("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 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 fonction `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("Pause")
self.jeton_pauseM = False
```

On réactive les labels précédemment 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.quitTimer()
self.Timer.wait()

```

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 appelle 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)))

```

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

```

self.genererStats()

```

Méthode (slot) temps_fini

Méthode appelée lorsque le temps est fini et permettant de paramétrer le GUI pour un éventuel 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 fonction `setUpRecommencer` pour reparamétrer 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("Recommencer")

```

Méthode `recommencer`

Méthode permettant de recommencer

```

def recommencer(self):

```

À faire

```

pass

```

Méthode `genererStats`

Méthode permettant de générer les statistiques

```

def genererStats(self):

```

On appelle les méthodes `compterMots`, `compterJusteErreur` et `compterScore` en charge des calculs des statistiques

```

self.compterMots()
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 (`s_mots`)

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

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

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

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

```
if nombre_mots <= 1:
    nombre_mots = 1
```

Enfin, on calcule le temps par mot moyen (`s_mots`), et on affiche cette valeur dans le GUI

```
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))))
```

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 précédemment

```
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 formule impliquant la vitesse, la précision, l'endurance, le temps choisi et l'avancement

```
def compterScore(self):
```

On calcule l'avancement comme étant le rapport du temps écoulé sur le temps total

```
avancement = self.temps_ecoule / self.temps_choisi
s_mots_mod = self.s_mots
```

On calcule l'inverse de la vitesse comme étant l'inverse de `s_mots` différent de 0 (pour éviter la division par 0)

```
if s_mots_mod == 0:
    s_mots_mod = 1
inv_vitesse = 1 / s_mots_mod
```

On calcule également le nombre de mots par minute
On l'affiche à la place de l'ancien label "Meilleurs scores"

```
mots_m = (1 / s_mots_mod) * 60
self.LabelBestT.setText(unicode(str(round(mots_m, 1))))
```

On calcule le logarithme népérien de l'inverse du ratio d'erreurs supérieur à 0.001

```
erreurs_mod = self.erreurs
if erreurs_mod < 0.001:
    erreurs_mod = 0.001
ln_inv_erreurs = log(1 / erreurs_mod)
```

On calcule la somme du logarithme népérien du temps choisi et d'une constante qui vaut ici 5.5 (à éventuellement modifier)

```
ln_temps_plusC = log(self.temps_choisi / 60) + 5.5
```

Enfin, on calcule le produit de tous ces facteurs, on multiplie le résultat par 100 et on affiche la valeur arrondie à l'entier le plus proche de ce score

```
self.score = avancement * inv_vitesse * ln_inv_erreurs * \
    ln_temps_plusC * 100
self.LabelScoreV.setText(unicode(str(int(round(self.score, 0)))))
```

Programme principal

Fonction main

Fonction prenant en argument (futur) les valeurs choisies dans le menu, et permettant de créer l'interface et de la lancer

```
def main():
```

On crée une application *Qt* `QApplication`, pour porter notre GUI

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

On crée notre GUI comme étant une instance de la classe `ModuleApplication` décrite plus haut

```
myapp = ModuleApplication()
```

On affiche notre GUI et on connecte sa fermeture à la fermeture du programme

```
myapp.show()
sys.exit(app.exec_())
```

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` défini plus haut avec des paramètres (futurs) par défaut

```
main()
```