Classe SudokuGrid :

Nous avons dans un premier temps eu l'idée de faire un jeu de Sudoku. Cependant, pour rendre le jeu sympathique, plus interactif et moins redondant, il fallait que l'on trouve une manière de générer des grilles de Sudoku aléatoirement. Au départ, nous ne savions pas trop comment commencer.

Nous avons donc commencé par créer des grilles de 9 par 9 remplies de 0 à l'initialisation.

```
self.grid = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Ensuite, il faut éviter qu'il n'y ait pas de répétition de numéro sur chaque ligne et sur chaque colonne. Nous avons donc créé la méthode start(), elle fait 3 tirages aléatoires de séries de 9 chiffres qui ne se répètent pas. Elle remplit successivement les 3 blocs de 9 chiffres présents dans la diagonale. Les valeurs tirées de 1 à 9 dans ces tableaux n'ont aucune influence sur les lignes et colonnes car on vérifie déjà si le tableau de 9 chiffres n'a aucune répétition.

```
def start(self):
    self.found = 0

    for k in range(0, 3):
        # Génération d'un tableau aléatoire de taille 9 avec les nombres 1 à 9 qui ne se repetent pas
        myarray = [0, 0, 0, 0, 0, 0, 0, 0, 0]
        for l in range(0, 9):
            value = randint(1, 9)
            while (value in myarray):
                value = randint(1, 9)
            myarray[l] = value
        # Remplissage des 3 sous grilles 3x3 en diagonal
        l = 0
        for i in range(0, 3):
            for j in range(0, 3):
                self.grid[k * 3 + i][k * 3 + j] = myarray[l]
                l = l + 1
```

Après cette étape, en s'inspirant d'une vidéo YouTube qui expliquait un algorithme de résolution de Sudoku, ce qui n'est pas exactement ce dont on avait besoin car on cherche à créer une grille de Sudoku, on a donc modifié cette technique pour l'adapter à ce dont on avait besoin. Ci-après le lien de cette vidéo: https://www.youtube.com/watch?v=O2pcNxzlL0Q

La fonction solve() parcourt les lignes et les colonnes de la grille de Sudoku et si la valeur de la case est nulle, alors on teste des valeurs pour voir si elles pourraient correspondre. A cette étape, seuls 3 blocs de 9 cases sont remplis dans la diagonale.

```python
# Résolution complete grille Sudoku
def solve(self):
    # On recherche les solutions uniques
    if self.found > 1:
        return

    for y in range(0, len(self.grid)):
        for x in range(0, len(self.grid)):
            if self.grid[y][x] == 0:
                for n in range(1, 10):
                    if self.n_valide(y, x, n):
                        self.grid[y][x] = n
                        self.solve()
                        self.grid[y][x] = 0
                return

    # on recopie la solution dans la grille self.final
    for i in range(0, len(self.grid)):
        for j in range(0, len(self.grid)):
            self.final[i][j] = self.grid[i][j]
    self.found = self.found + 1
```

C'est à ce moment que l'on fait appel à la fonction n_valide() qui permet de valider si les nombres placés en paramètres sont valides. Cette fonction vérifie que les chiffres mis en paramètres ne soient présents ni sur la ligne ni sur la colonne.

```python
# On regarde si le nombre n peut convenir sur la grille
def n_valide(self, y, x, n):
    # On determine si le nombre est valide sur sa ligne
    for x0 in range(len(self.grid)):
        if self.grid[y][x0] == n:
            return False
    # On determine si le nombre est valide sur sa colonne
    for y0 in range(len(self.grid)):
        if self.grid[y0][x] == n:
            return False
    # On determine si le nombre est valide dans sa sous grille
    x0 = (x // 3) * 3
    y0 = (y // 3) * 3
    for i in range(0, 3):
        for j in range(0, 3):
            if self.grid[y0 + i][x0 + j] == n:
                return False
    return True
```

On remplit récursivement la grille de Sudoku grâce à ces deux méthodes. Pour finir, dans la méthode solve(), on affecte à la grille finale, la grille complétée,dans laquelle il y a une grille entièrement aléatoire.

Une fois que nous avons obtenu une grille entièrement aléatoire, il faut que l'on puisse retirer un certain nombre de valeurs dans la grille pour avoir un jeu de Sudoku. Nous nous sommes alors aperçu que l'on ne peut pas retirer n'importe quelles cases, car dans certains cas, le joueur pourrait mettre plusieurs valeurs différentes. On devait donc s'assurer qu'à chaque fois que l'on retirait une valeur il devait y avoir une seule possibilité de résolution du jeu. En effet, s'il y a plusieurs résolutions du jeu, notre code n'aurait plus aucun sens car le joueur pourrait placer une valeur dans une case alors que celle-ci ne correspond pas à notre grille.

Nous avons donc ensuite créé la méthode create(), qui prend en paramètre le nombre de case vide que l'on souhaiterait avoir. Cependant, pour générer une

grille avec un grand nombre de cases vides, l'algorithme est très long, c'est pour cela qu'on lui demande de faire 100 tours maximum si le nombre de case vide souhaité n'est pas atteint. Après avoir réalisé cela, nous nous sommes aperçus que 100 tours suffisait largement pour avoir des grilles pour les niveaux 1 ou 2 et que 100 tours mettait un temps correct d'attente pour avoir un nombre de cases vides bien plus conséquent.

```
# Création d'une grille de Sudoku avec xx trous
def create(self, nbEmptyCells):
    # On recopie la grille solution dans la grille initiale
    if self.found == 0:
        return

    # On tire aléatoirement nbEmptyCells valeurs pour faire des trous dans la grille
    # l: nombre de trous
    # nbTours: nombre de tirages aléaroires (max = 100)
    l = 0
    nbTour = 0

    while l < nbEmptyCells and nbTour < 100:
        self.found = 0
        nbTour += 1

        # on recopie dans self.gris et self.copygrid la deniere matrice de self.final
        for i in range(0, len(self.grid)):
            for j in range(0, len(self.grid)):
                self.grid[i][j] = self.final[i][j]
                self.copygrid[i][j] = self.final[i][j]
```

On tire au hasard une des 81 cases du Sudoku. On la met à 0 et on teste de nouveau si le Sudoku est réalisable avec une solution. Si c'est le cas, on retire la case tirée au hasard et on met la valeur 0 à la place dans la grille "à trous".

```
    # Identification d'une cellule aleatoire de la matrice
    value = randint(0, 80)
    x0 = (value // 9)
    y0 = (value - x0 * 9)

    originalValue = self.grid[x0][y0]
    if originalValue != 0:
        self.grid[x0][y0] = 0
        self.solve()
        # Il recomplète la grile pour s'assurer qu'il y ait une solution
         print("nbTour:%d", nbTour)
         self.affiche()
         print("self.found:%d",self.found)
        # cas une solution unique trouvee
        if self.found == 1:
            for i in range(0, len(self.grid)):
                for j in range(0, len(self.grid)):
                    self.final[i][j] = self.grid[i][j]
            l = l + 1
        else:
            for i in range(0, len(self.grid)):
                for j in range(0, len(self.grid)):
                    self.final[i][j] = self.copygrid[i][j]

# self.grid contient les zeros et a une unique solution
# on recalcule self.final pour avoir la grille solution
self.found = 0
self.solve()
```

Nous avons également créé une méthode affiche() qui permet d'afficher dans le terminal les grilles pleines et celles où l'on a retiré des valeurs afin de s'assurer que l'algorithme fonctionne bien, cela aide au débogage.

```python
def affiche(self):
    print("self.grid")
    nbzero=0
    #pour connaitre le nombre de 0 présents dans la grille self.grid
    #et l'afficher dans le terminal
    for i in range(0, len(self.grid)):
        for j in range(0, len(self.grid)):
            print(self.grid[i][j], end=" ")
            if self.grid[i][j] == 0:
                nbzero += 1
        print()
    print(nbzero)
    print()

    #pour connaitre le nombre de 0 présents dans la grille self.final
    #et l'afficher dans le terminal
    nbzero=0
    print("self.final")
    for i in range(0, len(self.final)):
        for j in range(0, len(self.final)):
            print(self.final[i][j], end=" ")
            if self.final[i][j] == 0:
                self.nbZeros += 1
        print()
    print("nbzero",self.nbZeros)
    print()
```

Certaines fonctions permettent de récupérer les différentes grilles pour les utiliser dans les classes suivantes.

```python
# Pour recuperer la grille avec les zeros
def getGrid(self):
    return self.grid

# Pour recuperer la grille solution
def getFinalGrid(self):
    return self.final
```

On a également créé une méthode reset() qui fait appel aux différentes méthodes citées précédemment pour lancer le jeu.

```python
# Pour initilaiser le jeu avec nbEmptyCells zeros (difficile d'av
def reset(self, nbEmptyCells):
    # Creation d'une grille avec la diagonlae 3x3 remplie
    self.start()
    # On résoud la grille complète de Sudoku
    self.solve()
    self.create(nbEmptyCells)
    self.affiche()
```

Cette classe a été la plus difficile à concevoir durant la création de notre jeu.

Class TabWidgetWindow :

Nous avons créé une classe TabWidgetWindow qui utilise une QTabWIdget. Elle possède 2 onglets, l'un propose un champ pour entrer le prénom, le nom et l'âge du joueur et l'autre affiche les règles du jeu ainsi qu'une question demandant si le joueur a déjà joué au Sudoku.

La méthode tab1Affiche() possède un QFormLayout et un QGridLayout ainsi que deux QLineEdit, où le joueur y rentre son nom et son prénom. Dans la troisième QLineEdit, on ne peut entrer uniquement deux chiffres pour l'âge.

```
self.age = QLineEdit()
self.age.setValidator(QIntValidator
self.age.setMaxLength(2)
```

Puis, on a un label qui indique ce qu'il faut faire, ainsi qu'un bouton indiquant « Suivant » et qui permet d'aller à la page suivante.

Dans la deuxième page, on a une question qui est posée par l'intermédiaire d'un label, et les réponses sont formulées à l'aide de RadioButton où on demande si la personne qui s'apprête à jouer au Sudoku a déjà joué, les réponses possibles sont Oui/Non. Cependant, on ne récupère pas l'information rentrée car on ne savait pas quoi en faire, cette fonctionnalité pourrait être réutilisable dans le cadre d'une amélioration du jeu.

On utilise ici, une nouvelle fois des QFormLayout et QHBoxLayout pour placer les widgets dans la page, de manière ordonnée. On utilise également un QTextWidget dans lequel on place les consignes de jeu. Un texte qu'on rend non éditable par l'utilisation de la fonction setEnabled(False) qui empêche l'utilisateur de l'application de modifier ce qu'il y a écrit dans le fenêtre du QTextEdit().

```
texteEdit.setText(texte)
texteEdit.setEnabled(False)
texteEdit.setStyleSheet("font:10px;color:black;background-color:white")
```

Puis, on place toutes ces fonctionnalités dans la fenêtre par l'intermédiaire des fonctions addRow() et addWidget().

```
layout = QFormLayout()
joue = QHBoxLayout()
joue.addWidget(QRadioButton("Oui"))
joue.addWidget(QRadioButton("Non"))
joue.addWidget(texteEdit)
layout.addRow(QLabel("Avez vous déja joué au Sudoku ?"), joue)
```

La méthode assignSuivant() vérifie que lorsque l'on clique sur le bouton "Suivant" tous les champs sont renseignés avant d'ouvrir la page suivante et de fermer sa propre page.

On a également, 2 méthodes, regles() et labelConsignes() qui se contentent uniquement d'affecter à des strings leur valeur.

Dans l'init de cette classe, on donne un titre à la page, une taille minimale et on crée les deux pages du QTabWidget.

```python
def __init__(self, parent=None):
    super().__init__()
    self.setWindowTitle("Sudoku")
    self.setMinimumSize(550,250)

    self.regles()
    self.labelConsignes()

    self.tab1 = QWidget()
    self.tab2 = QWidget()

    self.addTab(self.tab1, "Tab 1")
    self.addTab(self.tab2, "Tab 2")

    self.tab1Affiche()
    self.tab2Affiche()
```

Classe FirstWindow :

La classe FirstWindow affiche le nom, le prénom et l'âge du joueur. Dans cette fenêtre, le joueur peut choisir le niveau dans lequel il souhaite jouer (niveau de 1 à 3). Puis un bouton Play lance la partie.

Notre méthode setBoutons() crée des QRadioButons qui correspondent aux niveaux de difficultés du jeu.

```python
def setBoutons(self):
    boutonRadio = ["niveau : 1", "niveau : 2", "niveau : 3"]
    boutons = [] #les niveaux
    for i in range(len(boutonRadio)):
        boutons.append(QRadioButton(boutonRadio[i]))
        self.grille.addWidget(boutons[i], i + 30, 0)
    facileBouton = boutons[0]
    moyenBouton = boutons[1]
    avanceBouton = boutons[2]
    facileBouton.clicked.connect(self.assignFacil)
    #facileBouton.setChecked(False)
    moyenBouton.clicked.connect(self.assignMoyen)
    #moyenBouton.setChecked(False)
    avanceBouton.clicked.connect(self.assignDifficil)
    #avanceBouton.setChecked(False)

    playBouton = QPushButton("PLAY")
    playBouton.clicked.connect(self.playPartie)
    self.grille.addWidget(playBouton)
```

Deux méthodes setIdentifiant() et saisiNiveau() créent des lables qui affichent le nom, prénom, age du joueue. Si le joueur a appuyé sur Play sans renseigner de niveau, un label s'affiche en signifiant que le niveau n'a pas encore été saisi. Pour cela, il initialise le label à non visible et lors de l'appel de la méthode playParty(), on vérifie si l'un des QRadioButton a été sélectionné.

Les méthodes, assignFacil(), assignMoyen() et assignDifficill() affirment que l'un des QRadioButton a été activé et affecte un niveau de jeu.

```python
def assignFacil(self):
    self.niveau = 1
    self.boutonCheck = True
    #print("niveau 1")
```

Pour afficher tout proprement, on crée une QGridLayout dans laquelle on place les boutons et les labels, qui sont les identifiants du joueur. Un label indique qu'il faut saisir le niveau et si le bouton Play est pressé avant d'avoir saisi le niveau, il y a un label qui apparaît indiquant que le joueur a oublié de sélectionner un niveau de jeu.

Afin de passer à la fenêtre de jeu, nous avons créé la fonction playPartie(). Cette fonction a pour but de lancer la fenêtre suivante, tout en fermant la fenêtre de choix du niveau. Avant toute chose, on souhaite vérifier que l'attribut booléen qui nous indique si le joueur a choisi un niveau soit vrai. Si cela n'est pas le cas, un label préalablement créé indiquant qu'il faut choisir un niveau avant d'aller plus loin apparaît sur notre fenêtre à l'aide de la commande setVisible(). Si la booléenne vaut vrai, alors on crée une fenêtre de classe MainWindow, classe que l'on détaillera par la suite dans le rapport. Cette fenêtre reçoit nos attributs niveau, nom, prénom et âge. On affiche cette fenêtre à l'aide de la commande show() et on ferme notre FirstWindow actuelle avec la méthode close().

```
def playPartie(self):
    if self.boutonCheck == True :
        self.window = MainWindow(str(self.niveau), self.nom, self.prenom, str(self.age))
        self.window.show()
        self.close()
    else :
        self.label2.setVisible(True)
```

Cette page a été plutôt simple à mettre en place. Notre seule réelle difficulté a été de récupérer les informations saisies dans la fenêtre précédente, mais nous avons vite trouvé la solution.

Classe Window :

Nous avons créé cette classe qui permet d'afficher dans un widget le jeu du Sudoku avec certaines cases de la grille qui sont vides. Cette classe permet également l'utilisation de boutons de 1 à 9 afin de remplir la grille de sudoku. Cette fenêtre permet également l'affichage du niveau choisi, du nombre d'erreurs et le temps que met le joueur pour résoudre le jeu.

Dans un premier temps, nous avons programmé une méthode "createBouton()" permettant via une boucle de créer des boutons de 1 à 9, que l'on place ensuite dans une QGridLayout afin de pouvoir les ordonner proprement. Après cela, cette grille est placée dans une boîte de type QVBoxLayout. Nous avons fait en sorte qu'il n'y ait pas d'espace entre les différents boutons afin de rendre l'affichage plus propre.

```python
def createBoutons(self):
    self.grilleChiffres = QGridLayout()
    boutons = []
    for i in range(9):
        unBouton = QPushButton(str(i + 1))
        # https://wiki.qt.io/Qt_for_Python_Tutorial_ClickableButton
        # https://eli.thegreenplace.net/2011/04/25/passing-extra-arguments-to-pyqt-slot
        unBouton.clicked.connect(functools.partial(self.assignValue, str(i + 1)))
        boutons.append(unBouton)
        self.grilleChiffres.addWidget(boutons[i], 2 - i // 3, i % 3)
    self.grilleChiffres.setHorizontalSpacing(0)
    self.grilleChiffres.setVerticalSpacing(0)
```

Ensuite, nous avons créé la fonction createGrille() qui initialise une QGridLayout composée de 9 lignes et 9 colonnes. Pour que le rendu soit plus propre, nous avons décidé de mettre une taille minimale à la fenêtre de notre jeu pour que tous les éléments soient correctement affichés. Nous avons également décidé de redimensionner la taille des différentes cases pour qu'elles soient carrées.

```python
def createGrille(self):
    newGrid = SudokuGrid()
    newGrid.solve()

    newGrid.reset(self.nbCasesVides)
    self.gridInit = newGrid.getGrid()
    self.gridFinal = newGrid.getFinalGrid()
    self.table = QTableWidget()
    nline = 9
    ncol = 9
    self.table.setRowCount(nline)
    self.table.setColumnCount(ncol)
    # permet d'éviter d'afficher les nom d'axes horizontaux et verticaux
    self.table.horizontalHeader().hide()
    self.table.verticalHeader().hide()
    self.table.horizontalScrollBar().setDisabled(True)
    self.table.verticalScrollBar().setDisabled(True)
    self.table.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)
    self.table.setVerticalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)
    for line in range(ncol):
        self.table.setRowHeight(line, 20)
    for col in range(nline):
            self.table.setColumnWidth(col, 20)
    for line in range(9):
        for col in range(9):
            self.table.setItem(line, col, QTableWidgetItem())
```

Nous avons mis beaucoup de temps à comprendre qu'il fallait mettre des QTableWidgetItem dans les différentes cases afin de pouvoir y mettre des valeurs à l'initialisation et aussi pendant le jeu.

Nous faisons ensuite appel à la class SudokuGrid qui permet d'avoir une grille peine et une grille partiellement pleine. Dans un premier temps, nous avons créé la méthode putValues() qui initialise les cases en blanc et les rend sélectionnables mais non éditables. On fait en sorte que les cases soient centrées et remplies par une string vide. On fait tout cela à l'initialisation afin de réutiliser cette méthode lorsqu'on clique sur le bouton replay. Ensuite, une fois que tout cela est fait, on met les valeurs de la grid récupérée dans toutes les cases de la QTableWidget. On n'affiche que les valeurs qui sont différentes de 0, car le 0 signifie à notre programme que la case sera à compléter par le joueur.

```python
def putValues(self):
    for line in range(9):
        for col in range(9):
            it = self.table.item(line, col)
            it.setText("")
            it.setBackground(QColor(256, 256, 256))
            it.setForeground(QColor(0, 0, 0))
            it.setFlags(it.flags() & ~Qt.ItemIsEditable)
            it.setTextAlignment(Qt.AlignCenter)
            if self.gridInit[line][col] != 0:
                it.setText(str(self.gridInit[line][col]))
                it.setBackground(QColor(100, 110, 10))
                it.setForeground(QColor(0, 0, 0))
                it.setFlags(it.flags() & ~Qt.ItemIsSelectable)
                it.setTextAlignment(Qt.AlignCenter)
    self.update()
```

Ensuite, on crée une fonction qui s'appelle onClickedRow() qui affecte à un attribut, self.selectedCelle, les valeurs de la cellule sélectionnée. Et on initialise cet attribut à [-1, -1] pour qu'il n'y ait pas de bug avant qu'une cellule soit sélectionnée.

On a également créé une fonction assignValues() dans laquelle on place en paramètre le nombre sélectionné. Dans un premier temps, on place la cellule sélectionnée en ligne et colonne. Pour ajouter des valeurs dans la grille de jeu, on teste si la case sélectionnée est bien dans la grille, puis on regarde si la case est bien vide et enfin si la valeur qu'on essaye de lui donner est bien la valeur correspond à la grille complétée. Si c'est le cas, on change la couleur de la cellule, on la rend non sélectionnable et on met la valeur dedans. Si la valeur n'est pas bonne, on augmente le nombre d'erreurs du joueur. On regarde ensuite le nombre de cases complétées afin de vérifier si toutes les cases sont complétées et lancer une fonction s'occupant de la fin du jeu.

```python
def assignValue(self, valeurbouton):
    row = self.selectedCell[0]
    column = self.selectedCell[1]  # bug si aucune case n'est sélectionnée
    if row >= 0 and row < 9 and column >= 0 and column < 9:
        it = self.table.item(row, column)
        valeurCellule = it.text()
        print(valeurCellule)
        if valeurCellule == "" and str(self.gridFinal[row][column]) == valeurbouton:
            it.setText(valeurbouton)
            it.setBackground(QColor(150, 110, 100))
            it.setTextAlignment(Qt.AlignCenter)
            it.setFlags(it.flags() & ~Qt.ItemIsSelectable)
        elif valeurCellule == "":
            self.nbErreurs()
        nb = 0
        for line in range(9):
            for col in range(9):
                it = self.table.item(line, col)
                valeurCellule = it.text()
                if valeurCellule != "":
                    nb += 1
        self.nb = nb
        print("nombre =", self.nb)
        self.table.update()
        QApplication.processEvents()
        if self.nb == 81 :
            self.endGame()
```

Ensuite, on crée une méthode ide() qui affiche toutes ces caractéristiques de jeu. Elle les met dans une grille pour que les objets soient ordonnés, puis l'ajoute dans une boîte pour l'affichage final. On à également un bouton help qui autorise un nombre maximal de trois aides par partie. On a rajouté une icône ainsi que le nombre d'aides disponibles. Si le joueur demande une aide, la méthode askHelp réduit le nombre d'aides restantes et inflige une pénalité de 10 secondes au joueur.

```
def ide(self):
    self.labelNiveau = QLabel()
    self.labelNiveau.setText("Niveau : " + str(self.niveau))
    self.labelNiveau.setAlignment(Qt.AlignCenter)
    self.labelNiveau.setStyleSheet("background :orange; color : black; font : 12pt;")

    self.helpButton = QPushButton()
    self.helpButton.setText("3")
    self.helpButton.setIcon(QIcon('help_icon.png'))
    self.helpButton.clicked.connect(self.askHelp)

    self.grilleIde = QGridLayout()
    self.grilleIde.addWidget(self.labelNiveau,0,0)
    self.grilleIde.addWidget(self.helpButton,1,0)
```

```
def askHelp(self):
    row = self.selectedCell[0]
    column = self.selectedCell[1]
    if row >= 0 and row < 9 and column >= 0 and column < 9:
        it = self.table.item(row, column)
        valeurCellule = it.text()
        print(valeurCellule)
        if valeurCellule == "":
            if self.nbHelp > 0:
                self.nbHelp -= 1
                self.helpButton.setText(str(self.nbHelp))
                self.time += 100  # si on fait appel à de l'aide : pénalité de 10secondes
                self.assignValue(str(self.gridFinal[row][column]))
                if self.nbHelp == 0:
                    self.helpButton.setEnabled(False)
```

Pour faire le chrono, on utilise QTimer. On a donc créé plusieurs méthodes, à savoir start(), pause(), reset() et showTime() et méthode setChrono().

```
timer = QTimer(self)
timer.timeout.connect(self.showTime)
timer.start(100)#pour qu'on puisse avoir le temps en seconde *10
```

```
def showTime(self):
    if self.flag :
        self.time += 1
    self.texte = str(self.time / 10)#1 chiffre après la virgule(et le temps en seconde)
    self.label.setText("Chrono :" + self.texte)

def start(self):
    self.flag = True

def pause(self):
    self.flag = False

def reset(self):
    self.flag = True
    self.time = 0
    self.playGame = True
```

La fonction replayPartie() permet de relancer une partie, elle remet tout à zéro et renvoie une nouvelle grille de jeu. Enfin, on cache le widget contenant les meilleurs scores avec la méthode setVisible(False) afin de retrouver une fenêtre de jeu identique à celle du lancement du jeu.

```
def replayPartie(self):
    newGrid = SudokuGrid()
    newGrid.solve()
    newGrid.reset(self.nbCasesVides)
    self.gridInit = newGrid.getGrid()
    self.gridFinal = newGrid.getFinalGrid()
    self.putValues()
    self.update()
    self.reset()
    self.resetError()
    self.nbHelp = 3
    self.helpButton.setEnabled(True)
    self.helpButton.setText("3")
    self.windowScore.setVisible(False)
```

La fonction setScore() a pour but d'initialiser les différents meilleurs scores à 0 et d'assigner aux informations sur les meilleurs scores une string vide lorsque l'on lance le jeu. Enfin, on crée un attribut de type fenêtre des meilleurs scores, une classe que l'on détaillera plus tard, et on lui assigne les informations sur les high-score.

```
def setScore(self):
    self.bestTime = [0,0,0]
    self.scoreEasy = ["Facile","","","","","",""]
    self.scoreMedium = ["Moyen","", "", "", "", "", ""]
    self.scoreHard = ["Difficile","", "", "", "", "", ""]
    self.windowScore = WindowHighScore(self.scoreEasy,self.scoreMedium,self.scoreHard)
```

La fonction putScore() modifie les informations sur le meilleur score. Elle modifie le tableau contenant le nom, le prénom, l'âge, le nouveau meilleur temps ainsi que le nombre d'erreurs, en fonction du niveau actuel.

```
def putScore(self):
    if self.niveau == "1":
        self.scoreEasy[1] = str(self.bestTime[0]/10)
        self.scoreEasy[2] = str(self.erreurs)
        self.scoreEasy[3] = self.prenom
        self.scoreEasy[4] = self.nom
        self.scoreEasy[5] = str(self.age)
    if self.niveau == "2":
        self.scoreMedium[1] = str(self.bestTime[1]/10)
        self.scoreMedium[2] = str(self.erreurs)
        self.scoreMedium[3] = self.prenom
        self.scoreMedium[4] = self.nom
        self.scoreMedium[5] = str(self.age)
    if self.niveau == "3":
        self.scoreHard[1] = str(self.bestTime[2]/ 10)
        self.scoreHard[2] = str(self.erreurs)
        self.scoreHard[3] = self.prenom
        self.scoreHard[4] = self.nom
        self.scoreHard[5] = str(self.age)
```

Afin de pouvoir modifier le niveau de difficulté directement depuis la barre de menu, nous avons codé des fonctions qui permettent de changer le niveau actuel. Au départ, nous pensions faire une seule fonction qui choisissait quel niveau assigné, cependant il n'est pas possible d'ajouter des arguments lorsque l'on appelle une fonction avec une action de MenuBar. Nous avons donc fait 3 fonctions identiques à quelques détails près.
Ces méthodes assignent à l'attribut self.niveau une nouvelle valeur, change le nombre de cases à cacher, met à jour le label affichant le niveau, puis relance une nouvelle partie.

```
def niveau1(self):
    self.niveau = "1"
    self.nbCasesVides = 10
    self.labelNiveau.setText("Niveau : " + str(self.niveau))
    self.replayPartie()

def niveau2(self):
    self.niveau = "2"
    self.nbCasesVides = 30
    self.labelNiveau.setText("Niveau : " + str(self.niveau))
    self.replayPartie()

def niveau3(self):
    self.niveau = "3"
    self.nbCasesVides = 70
    self.labelNiveau.setText("Niveau : " + str(self.niveau))
    self.replayPartie()
```

La méthode endGame() est appelée par la fonction assignValues() lorsque toutes les cases ont été complétées. Cette fonction commence par arrêter le chronomètre. Ensuite, elle regarde si le meilleur score actuel vaut 0, ce qui veut dire que quel que soit le temps, il sera le nouveau record pour ce niveau de difficulté, ou si le temps de ce niveau est meilleur que le meilleur score enregistré. Dans ces deux cas, le temps obtenu est inséré comme meilleur score ainsi que les informations sur le joueur. De plus, le widget contenant les meilleurs scores est actualisé et il s'affiche dans notre fenêtre de jeu à l'aide de la fonction addWidget().

```
def endGame(self):
    self.pause()
    if self.niveau == "1":
        if self.bestTime[0] == 0:
            self.bestTime[0] = self.time
            self.putScore()
        else:
            if self.time < self.bestTime[0]:
                self.bestTime[0] = self.time
                self.putScore()
    if self.niveau == "2":
        if self.bestTime[1] == 0:
            self.bestTime[1] = self.time
            self.putScore()
        else:
            if self.time < self.bestTime[1]:
                self.bestTime[1] = self.time
                self.putScore()
    if self.niveau == "3":
        if self.bestTime[2] == 0:
            self.bestTime[2] = self.time
            self.putScore()
        else:
            if self.time < self.bestTime[2]:
                self.bestTime[2] = self.time
                self.putScore()
    self.windowScore = WindowHighScore(self.scoreEasy,self.scoreMedium,self.scoreHard)
    self.boite.addWidget(self.windowScore)
```

Nous avons créé un menu déroulant qui permet au joueur de changer lui-même la couleur du fond d'écran du jeu, en fonction des différentes couleurs proposées. Pour cela, on utilise la fonction comboBox() et changeColor(). Dans comboBox(), on utilise un QComboBox non éditable, pour que le joueur ne puisse pas écrire dans cette comboBox. On crée également une liste de strings qui sont les différentes couleurs proposées. Ensuite, par la méthode addItems(), on place la liste de strings dans la comboBox puis on place lacomboBox dans l'attribut self.grilleIde pour afficher la comboBox à l'endroit où on le souhaite. Puis, pour pouvoir changer la couleur décidée par le joueur, on fait appel à la méthode changeColor() qui récupère la couleur sélectionnée et la réutilise dans la méthode setStyleSheet() pour changer la couleur de la fenêtre principale.

```
def comboBox(self):
    self.comboBox = QComboBox()
    self.comboBox.setEditable(False)
    listColors = ["white", "blue", "green", "cyan", "yellow", "brown", "red","pink", "purple"]
    self.comboBox.addItems(listColors)
    self.grilleIde.addWidget(self.comboBox, 1, 1)
    self.comboBox.activated.connect(self.changeColor)
```

```
def changeColor(self):
    couleur = self.comboBox.currentText()
    self.setStyleSheet("background : " + couleur)
```

Les difficultés de cette classe ont été le nombre de fonctionnalités ainsi que des classes que nous ne connaissions pas telles que QTableWidget, QTimer que nous avons beaucoup utilisé au cours de la création de notre jeu.

Classe Main Window:

Nous souhaitions ajouter à notre jeu une barre de menu en haut de la fenêtre. Cependant, nous nous sommes heurtés à un problème, notre fenêtre contenant le sudoku était de classe QWidget et seule une classe de type QMainWindow permet de recevoir une barre de menu. Nous avons donc essayé de rajouter une barre de menu sur notre fenêtre de type QWidget, cependant malgré toutes nos tentatives, cela ne fonctionnait pas. Nous avons alors repris ce que nous avons vu en cours et nous avons cherché comment convertir le plus simplement possible notre fenêtre de type QWidget vers une fenêtre de type QMainWindow pouvant recevoir des menus. Il a donc fallu créer une classe de type QMainWindow, aux dimensions similaires à notre ancienne fenêtre. Ensuite, il fallait pouvoir incorporer la classe de type QWidget dans notre QMainWindow. Pour cela, on crée un attribut de type QWidget contenant le jeu du sudoku dans notre fenêtre principale MainWindow. On assigne ce widget comme widget central à l'aide de la commande setCentralWidget(). On récupère en héritage les variables qui nous seront utiles telles que le nom, le prénom, l'âge et on les rentre dans notre attribut QWidget.

```
class MainWindow(QMainWindow):
    def __init__(self,niveau, nom, prenom, age):
        super().__init__()
        self.setWindowTitle("Sudoku")  # nom de la fenetre
        self.setMinimumSize(540,540)
        """pal = QPalette()
        pal.setColor(self.backgroundRole(), QColor("red"))
        self.setPalette(pal)"""
        self.nom = nom
        self.prenom = prenom
        self.age = age
        self.main_widget = Window(niveau,nom,prenom,age)
        self.main_widget.start()
        self.setCentralWidget(self.main_widget)
        self.setMenuBar()
```

Sur cette classe, nous avons ainsi rajouté une barre de menu contenant plusieurs fonctionnalités. Pour cela, on crée un objet de type menuBar auquel on ajoute plusieurs menus ou actions de type QAction avec la méthode addMenu() et addAction(). On relie ensuite ces actions à des fonctions à l'aide de la formule triggered.connect().

```
def setMenuBar(self):
    self.menu_bar = self.menuBar()
    self.menu_bar.setNativeMenuBar(False)

    game = self.menu_bar.addMenu("Game")
    niveau = game.addMenu("Level")
    niv1 = QAction("Level 1", self)
    niv1.triggered.connect(self.main_widget.niveau1)
    niveau.addAction(niv1)
    niv2 = QAction("Level 2", self)
    niv2.triggered.connect(self.main_widget.niveau2)
    niveau.addAction(niv2)
    niv3 = QAction("Level 3", self)
    niv3.triggered.connect(self.main_widget.niveau3)
    niveau.addAction(niv3)

    restart = QAction("Restart", self)
    restart.triggered.connect(self.main_widget.replayPartie)
    game.addAction(restart)
    quitter = QAction("Quitter", self)
    quitter.triggered.connect(self.close)
    game.addAction(quitter)
```

Au sein de cette classe, on a créé une seconde méthode. Celle-ci permet d'afficher une fenêtre contenant les meilleurs scores précédemment obtenus sur notre jeu. Nous avons donc ajouté à notre classe des attributs contenant plusieurs string avec les noms, prénoms, âge, temps, nombre d'erreurs et niveau des meilleurs scores précédemment obtenus. Ces attributs sont vides lors de l'initialisation. Pour afficher nos meilleurs scores, nous déclarons simplement une variable fenêtre de type Window HighScore que l'on expliquera plus tard. On lui fournit nos tableaux contenant les informations sur le meilleur score et on affiche cette variable.

```
def showHighScore(self):
    self.windowScore = WindowHighScore(self.main_widget.scoreEasy, self.main_widget.scoreMedium, self.main_widget.scoreHard)
    self.windowScore.setMinimumSize(600, 200)
    self.windowScore.show()
```

Classe Window HighScore :

La classe Window HighScore reçoit plusieurs tableaux contenant des chaînes de caractères et les assigne comme attribut de cette classe.

```
class WindowHighScore(QWidget):
    def __init__(self,scoreEasy,scoreMedium,scoreHard):
        super().__init__()
        self.setWindowTitle("High Score")
        #self.setMinimumSize(650, 250)
        self.scoreEasy = scoreEasy
        self.scoreMedium = scoreMedium
        self.scoreHard = scoreHard
        self.madeBox()
        self.putScore()
```

Nous avons ensuite créé deux méthodes qui permettent de créer une QTableWidget et de la remplir avec les informations sur les meilleurs scores. La première fonction, madeBox(), crée une Table de taille 4 par 6 et désactive le fait de pouvoir se déplacer sur la table puisque notre fenêtre est adaptée afin de recevoir le tableau en entier. Nous parcourons ensuite les cases afin de leur assigner un item de type QTableWidgetItem et de modifier la taille des cases pour qu'elles soient adaptées à nos chaînes de caractères. On parcourt une deuxième fois les cases afin de leur assigner une string vide pour leur initialisation, ainsi que pour les désactiver la fonctionnalité permettant de modifier les items.

```
def madeBox(self):
    self.boite = QVBoxLayout()
    self.table = QTableWidget(4, 6)
    self.table.horizontalHeader().hide()
    self.table.verticalHeader().hide()
    self.table.horizontalScrollBar().setDisabled(True)
    self.table.verticalScrollBar().setDisabled(True)
    self.table.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)
    self.table.setVerticalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)
    self.boite.addWidget(self.table)
    self.setLayout(self.boite)
    for line in range(0, 4):
        for col in range(0, 6):
            self.table.setItem(line, col, QTableWidgetItem())
            self.table.setRowHeight(line, 20)
            self.table.setColumnWidth(col, 80)
    for line in range(0, 4):
        for col in range(0, 6):
            item = self.table.item(line, col)
            item.setText("")
            item.setFlags(item.flags() & ~Qt.ItemIsEditable)
```

Notre deuxième fonction a pour but de rentrer les valeurs dans notre table. Pour cela, on crée une boucle qui rentre les titres des colonnes dans notre première ligne. Puis, on crée de nouvelles boucles qui affichent, dans les différentes lignes, les informations sur les meilleurs scores que l'on a déjà déclaré comme attribut.

```
def putScore(self):
    param = ["Niveau:", "Temps:", "Erreurs:", "Prénom:", "Nom:", "Age:"]
    for i in range(0, 6):
        item = self.table.item(0, i)
        item.setText(param[i])
        item.setTextAlignment(Qt.AlignCenter)
    for i in range(0, 6):
        item = self.table.item(1, i)
        item.setText(self.scoreEasy[i])
        item.setTextAlignment(Qt.AlignCenter)
    for i in range(0, 6):
        item = self.table.item(2, i)
        item.setText(self.scoreMedium[i])
        item.setTextAlignment(Qt.AlignCenter)
    for i in range(0, 6):
        item = self.table.item(3, i)
        item.setText(self.scoreHard[i])
        item.setTextAlignment(Qt.AlignCenter)
```

Enfin, nous avons ajouté une fonction writeScore() qui permet de rentrer les meilleurs résultats dans un fichier nommé score.txt à l'aide de la méthode write(). Nous avons aussi rajouté une méthode readScore() qui permet de lire le fichier texte. Cela nous permet de sauvegarder les meilleurs scores même lorsque l'on relance notre jeu.

```
def readScore():
    scoreAll=[]
    try:
        with open("score.txt", "r") as filin:
            for ligne in filin:
                print(ligne)
                scoreN = group(ligne.strip())
                scoreAll.append(scoreN)
            filin.close()
    except:
        scoreAll = [['Facile', '', '', '', '', ''],
                    ['Moyen', '', '', '', '', ''],
                    ['Difficile', '', '', '', '', '']]
        print(scoreAll)

    return scoreAll
```

```
def writeScore(scoreTab):
    with open("score.txt", "w") as filout:
        for scoreN in scoreTab:
            score = concatenate(scoreN) +"\n"
            filout.write(score)
        filout.close()
```