

Nom de votre projet :	Medieval Heroes
Membres de l'équipe :	Marylou LAPÔTRE
Membres de l'équipe :	Paul MAILLET
Membres de l'équipe :	Louis AMEDRO
Niveau d'étude :	Terminale
Etablissement scolaire :	Léonard de Vinci CALAIS
Enseignante/enseignant de NSI :	Christophe Miesczak

## > SOMMAIRE

> SOMMAIRE.....	1
> FONCTIONNEMENT GENERAL.....	2
> LES SOURCES.....	3
> LE MODULE TERRAIN.....	3
> LE MODULE ATTRIBUTS JEU.....	6
> LE MODULE SOURIS.....	7
> LE MODULE JEU.....	13
> LE MODULE PERSONNAGE.....	30
> MODULE OBJETS.....	34
> LE MODULE AFFICHER.....	37
> DOSSIER MEDIAS.....	53
> LE MODULE MUSIQUE ET SONS.....	55
> LE MODULE SAUVEGARDE.....	56
> LE MODULE ROBOT.....	64
> MEDIEVAL HEROES !!.....	75

## > FONCTIONNEMENT GENERAL

Comment fonctionne le code dans son ensemble ?

Le code du jeu Medieval Heroes est constitué de plusieurs modules interagissant entre eux :

- Le module jeu (module\_jeu.py) gère le déroulement du jeu dans son ensemble, il représente le cerveau du jeu.
- Le module terrain, lui, sert de plateau, de support pour placer, déplacer, supprimer les personnages du module personnage (module\_personnage.py) et aussi à placer des objets (module\_objets.py).
- Le module musique (module\_musique\_et\_sons.py) et le module afficher (module\_afficher.py) servent à rendre le jeu plus vivant. Les modules attributs jeu (module\_attributs\_jeu.py) et souris (module\_souris.py) permettent respectivement de garder toutes les informations du jeu et d'interagir avec le jeu.
- Le module de sauvegarde (module\_sauvegarde.py) permet de sauvegarder et charger une partie et le module robot (module\_robot.py) permet de jouer non pas contre un autre joueur, mais contre un robot.
- Pour finir, le fichier Medieval\_Heroes.py permet de lancer le jeu, il suffit alors d'appuyer sur le bouton vert ou sur F5.



Le découpage en modules a plusieurs intérêts, comme la clarté et l'organisation du code.

Cette programmation modulaire permet aussi de pouvoir utiliser facilement la programmation orientée objet (POO), au programme de terminale. Celle-ci nous sera très utile étant donné que nous souhaitons faire un jeu de plateau. Chaque personnage du jeu sera alors un objet du jeu.

D'autres points du programme de terminale seront aussi présents dans ce code tels que les piles, les files ainsi que les graphes.

Les piles seront utilisées pour la console du jeu, affichant les actions passées des joueurs dans le jeu, les files seront utilisées pour les potions et les graphes seront utilisés pour la recherche d'un chemin le plus court pour l'attaque des monstres et l'animation du déplacement d'un personnage.

Le module que nous avons privilégié pour l'animation et l'affichage du jeu est le module pygame. Nous l'avons préféré à d'autres car celui-ci nous permet une grande liberté d'action.

Ci-dessous, nous vous expliquerons plus en détails le fonctionnement des différents modules.

Nous vous recommandons d'avoir le code sous les yeux pendant votre lecture

d'un module ou d'une méthode pour mieux comprendre toute l'étendue du code !

## > LES SOURCES

Voici plus en détail l'origine des sons et des musiques du jeu :

- Pour les sons : [Bruitages & Sons, gratuits et libres de droits - LaSonotheque](#)
- Pour la musique : [https://www.youtube.com/watch?v=eZ\\_r1H9vHkI](https://www.youtube.com/watch?v=eZ_r1H9vHkI)

## > LE MODULE TERRAIN

Comme dit précédemment, le *module\_terrain* a pour but de créer une base sur laquelle on pourra insérer/supprimer les personnages du *module\_personnage*. On aura besoin d'importer le module *attributs\_jeu* pour une assertion lors de l'initialisation de la classe

```
import module_attributs_jeu
```

\* Pour l'initialisation du terrain, nous aurons besoin des attributs du jeu, (paramètre contenant beaucoup d'informations sur le jeu et appartenant à la classe *Attributs\_Jeu* du module *module\_attributs\_jeu.py*) et d'un niveau (dans notre version du jeu, il n'y en a qu'un, on mettra alors par défaut le niveau 1)

```
class Terrain():
    ...
    Une Classe pour le terrain
    ...
    def __init__(self, attributs_jeu, niveau = 1):
        ...
        initialise le terrain
        : params
            attributs_jeu (Attributs_Jeu) module
            niveau (int) valant par défaut 1
        ...
        #Assertion
        #assert isinstance(attributs_jeu, module_attributs_jeu.Attrib
        assert isinstance(niveau, int) and niveau > 0, "le niveau doi
        #Code
        self.attributs_jeu = attributs_jeu
        self.niveau = niveau
        self.grille = Terrain.attribuer_grille(self)
```

\* On initialisera ensuite la grille du terrain que l'on appellera *self.grille* à l'aide de la méthode *attribuer\_grille*.

La grille du jeu est implémentée selon le niveau, d'autres niveaux seraient donc facilement implantables dans le futur.

La grille du jeu est un tableau constitué de tableaux. Chaque élément de cette grille représente une case. La chaîne de caractères ‘ ‘ représente une case vide et la chaîne de caractères ‘X’ représente un obstacle (buisson, arbre, eau), case sur laquelle les personnages ne pourront donc pas aller.

- \* L'accesseur `acc_terrain` renvoie le contenu de la case de coordonnées `x, y` passé en paramètres. Elle sera beaucoup utilisée en dehors du module. Le mutateur `mut_terrain` permet de placer un personnage ou un coffre à un endroit de la grille, peu importe si cette case est libre ou non.
- \* Pour savoir si la case comporte ou non un obstacle, on utilisera la méthode `est_possible`.

```
def est_possible(self, x, y):
    ...
    renvoie True si la case de coordonnées (x,y) est vide et False sinon
    : params
        x, y (int)
    : return (bool)
    ...

    #assertions
    assert isinstance(x, int) and 0 <= x <= 20,'x doit être un entier compris entre 0 et 20 inclus'
    assert isinstance(y, int) and 0 <= y <= 20,'y doit être un entier compris entre 0 et 20 inclus'
    #code
    return self.acc_terrain(x, y) == ' '
```

\* Deux fonctions sont également utilisées : `cases_around` et `tuples_en_coordees`. Le fait qu'elles ne soient pas des méthodes permet de les appeler facilement depuis les autres modules (voir `module_personnage` et `module_robot`).

```
def cases_around(coordo):
    ...
    renvoie les 8 cases se situant autour de la case de coordonnées (coordo):
    : param coordo (tuple):
    : return (list of tuples)
    ...

    #Assertion
    assert isinstance(coordo, tuple), "les coordonnées doivent être dans un tuple"
    assert 0 <= coordo[0] <= 20 and 0 <= coordo[1] <= 20, "les coordonnées doivent être dans le terrain"
    #Code
```

\* La méthode `trouver_case_libre_proche` prend en paramètres deux

```
def tuples_en_coordees(coordo_perso, cases, numero_geant = None):
    ...
    change les tuples composés de -1, 1 et de 0 avec des coordonnées de cases
    : params
        coordo_perso (tuple)
        cases (list)
        numero_geant (int ou None) si int alors c'est un géant sinon personnage "normal"
    : return (list of tuples), le tableau avec les coordonnées des cases
    ...

    #Assertions
    assert isinstance(coordo_perso, tuple), "les coordonnées doivent être dans un tuple"
    assert isinstance(cases, list), "les cases sont un tableau"
    assert numero_geant == None or numero_geant in [0, 1, 2, 3], "le numéro du géant doit être soit None soit 0, 1, 2, 3"
    #Code
```

coordonnées, un `x` et un `y` et une chaîne de caractères et utilise les sous-méthodes `trouver_case` et `condition_case_geant`. Cette méthode est

utilisée uniquement dans un cas spécial : lorsqu'il faut ressusciter un personnage (voir le *module\_jeu*). Elle permet en autre de faire ressusciter un personnage le plus près possible de la case où il est mort. La chaîne passée en paramètres spécifie en étant ‘oui’ que le personnage à ressusciter est un géant (on doit alors trouver quatre cases libres et non une) sinon, en étant ‘non’.

```

def trouver_case_libre_proche(self, x, y, chaîne):
    ...
    renvoie les coordonnées de la case libre la plus proche de la case dont les coordonnées sont ceux passés en paramètres
    : params
        x, y (int)
        chaîne (str), 'non' si le personnage à ressusciter est un géant et 'non' sinon
    : return (tuple)
    ...

#Assertions
assert isinstance(x, int) and 0 <= x <= 20, 'x doit être un entier compris entre 0 et 20 inclus'
assert isinstance(y, int) and 0 <= y <= 20, 'y doit être un entier compris entre 0 et 20 inclus'
assert chaîne in ['oui', 'non'], "la chaîne doit être soit 'oui' soit 'non'"
#Code
##on vérifie la case de départ
case = (x, y)
trouve = self.est_possible(x, y) #regarde si la case passée en paramètre est libre
i = 1
if trouve and chaîne == 'oui' : #si on a trouvé une case libre mais que le personnage est un géant
    #4 cases doivent être libre
    self.condition_case_geant(x, y)
##on cherche
while not trouve : #on cherche tant qu'on n'a pas trouvé
    tab = self.trouver_case(i)
    t = 0
    while t < len(tab) and not trouve:
        n_x = x + tab[t][0]
        n_y = y + tab[t][1]
        if 0 <= n_x <= 20 and 0 <= n_y <= 20: #si la case est dans la grille
            trouve = self.est_possible(n_x, n_y)
            case = (n_x, n_y)
            if trouve and chaîne == 'oui' : #si on a trouvé une case libre mais que le personnage est un géant
                #4 cases doivent être libre
                self.condition_case_geant(n_x, n_y)
        t += 1
    i += 1
return case

def trouver_case(self, rang):
    ...
    renvoie le tableau avec des semi-coordonnées des cases éloignés de rang de la case centrale
    : param rang (int)
    : return (list)
    ...

tab = []
for i in range(-rang, rang+1):
    for j in range(-rang, rang+1):
        if abs(i) == rang or abs(j) == rang:
            tab.append((i, j))
return tab

```

Voici la sous-méthode qui vérifie trois conditions supplémentaires pour ressusciter un géant :

```

def condition_case_geant(self, x, y):
    ...
    renvoie Vrai si la case de coordonnées x, y peut être la case en haut à gauche d'un géant
    (les 3 autres cases doivent être libres)
    : params
        x, y (int)
    : return (bool)
    ...

    return (self.est_possible(x + 1, y) and
            self.est_possible(x , y + 1) and
            self.est_possible(x + 1, y + 1))

```

\* Pour finir, la méthode `trouver_case_libre` renvoie, au hasard, une case libre sur le terrain (utilisée pour l'apparition des monstres et des coffres).

```
def trouver_case_libre(self, chaîne = None):
    ...
    renvoie un tuple de coordonnées pris au hasard et avec la case à ces coordonnées
    : param chaîne (str ou None), si str, vaut soit 'bas' soit 'haut' → utilisé pour
    zone spécifique
    : return (tuple)
    ...
    tab_pont = [(3, 2), (17, 2), (3, 18), (17, 18), #petit pont
                 (2, 9), (3, 9), (2, 10), (3, 10), (2, 11), (3, 11), #pont gauche
                 (17, 9), (18, 9), (17, 10), (18, 10), (17, 11), (18, 11), #pont droit
                 (9, 9), (10, 9), (11, 9), (9, 10), (10, 10), (11, 10), (9, 11), (10, 11), (11, 11)]
    ...
    ##Données
    trouve = False #par défaut, on n'a pas trouvé de case

    #Tant que la future case n'est pas libre, on choisit une nouvelle fois une case
    while not trouve :
        #Coordonnées au hasard
        x = random.randint(0, 20)
        if chaîne == 'haut':
            y = random.randint(0, 10)
        elif chaîne == 'bas' :
            y = random.randint(10, 20)
        else :
            y = random.randint(0, 20)
        trouve = self.est_possible(x, y)
        if trouve and chaîne == 'monstre' :
            trouve = not (x, y) in tab_pont #les monstres ne doivent pas apparaître
    return x, y
```

## > LE MODULE ATTRIBUTS JEU

Le module `attributs_jeu` est très important puisque celui-ci regroupe toutes les caractéristiques du jeu, d'où ses nombreux attributs. Ces attributs sont, par exemple, le temps (Jour ou Nuit) du jeu, les tableaux contenant tous les personnages et les coffres, le nombre d'actions et le nombre de tours passés, la réserve de potions des sorcières, la console, etc. Il est donc composé uniquement d'accesseurs et de mutateurs reliés aux différents attributs du jeu. Les autres modules vont alors utiliser ces différentes méthodes pour pouvoir faire évoluer le jeu. Voici un exemple d'accesseur et de mutateur qu'on peut trouver dans ce module :

- Un accesseur du nombre d'actions déjà passées :

```
def acc_nb_actions(self):
    ...
    Renvoie l'attribut nb_actions
    : return (int)
    ...
    return self.nb_actions
```

- Un mutateur qui permet de changer le temps :

```
def mut_temps(self, chaine) :
    ...
    Modifie l'attribut temps
    : param chaine (str), 'Jour' ou 'Nuit'
    : pas de return, modifie l'attribut temps
    ...
    #Assertion :
    assert chaine in ['Jour', 'Nuit'], "Le paramètre doit être"
    #Code :
    self.temps = chaine
```

## > LE MODULE SOURIS

Le *module\_souris* est très utile pour les différents événements (clique) de la souris.

```
class Souris() :
    ...
    Une classe Souris qui gère les entrées du clavier et de la souris
    ...
    def __init__(self, jeu, attributs_jeu, sauvegarde, terrain, gestionnaire_sons) :
        ...
        Initialise les attributs pour gérer les entrées.
        : params
            jeu (module_jeu.Jeu)
            attributs_jeu (module_attributs_jeu.Attributs_Jeu)
            sauvegarde (module_sauvegarde.Sauvegarde)
            terrain (module_terrain.Terrain)
            gestionnaire_sons (module_musique_et_sons.GestionnaireSon)
        ...
        #Assertions :
        assert isinstance(jeu, module_jeu.Jeu), 'jeu doit être de la classe Jeu du m
        assert isinstance(attributs_jeu, module_attributs_jeu.Attributs_Jeu), 'attri
        assert isinstance(sauvegarde, module_sauvegarde.Sauvegarde), 'terrain doit ê
        assert isinstance(terrain, module_terrain.Terrain), 'terrain doit être de la
        assert isinstance(gestionnaire_sons, module_musique_et_sons.Gestionnaire_Son
        #Attributs des paramètres :
        self.jeu = jeu
        self.attributs_jeu = attributs_jeu
        self.sauvegarde = sauvegarde
        self.terrain = terrain
        self.musique_et_sons = gestionnaire_sons
```

Cette classe prend en paramètres la plupart des modules du jeu. C'est-à-dire le jeu en lui-même, ses attributs, la Sauvegarde, le Terrain et le Gestionnaire\_Son. Le seul autre attribut qu'on a besoin est *appuye* qui sert à savoir si une touche est appuyée ou non. Par la suite, on a des accesseurs et mutateurs qui servent à accéder ou modifier un attribut ou un paramètre bien spécifique via une autre

classe.

```
def personnage_est_selectionne(self):
    ...
    Si un personnage est cliqué, alors calcule ses déplacements et ses attaques qui sont possibles
    Sinon, on fait rien
    : pas de return
    ...
    position_case = self.acc_position_case()
    #Si la case cliquée est sur le terrain :
    if 0 <= position_case[0] <= 20 and 0 <= position_case[1] <= 20:
        selection = self.terrain.acc_terrain(position_case[0], position_case[1]) #Selectionne le personnage de la case
        if selection == self.attributs_jeu.acc_selection():
            self.jeu.changer_personnage(' ') #si le joueur réclame sur le même personnage, on enlève la sélection
        else :
            self.jeu.changer_personnage(selection) #Sinon, on change le personnage.
```

Cette méthode change le personnage sur lequel le joueur a cliqué grâce à la position de la souris sur les cases du terrain si et seulement si le curseur de la souris est sur le terrain. Si le joueur a cliqué sur le même personnage, le jeu désélectionne le personnage. Sinon, le jeu change le personnage sélectionné.

```
def est_clique(self):
    ...
    Déroule les fonctions deplacement_est_clique et attaque_est_clique correctement
    : pas de return
    ...
    #Si aucun déplacement ou attaque a été effectué, alors change de personnage par
    if not (self.jeu.deplacement_est_clique() or self.jeu.attaque_est_clique()):
        self.personnage_est_selectionne()
```

Cette méthode change le personnage sur lequel le joueur a cliqué si et seulement si le joueur n'a effectué aucune action dans le jeu. C'est-à-dire déplacer ou attaquer un joueur.

```
def potion_est_clique(self, equipe_en_cours, equipe_perso):
    ...
    si une potion est cliqué, change la sélection
    : params
        equipe_en_cours (str), 'rouge' ou 'bleu'
        equipe_perso (str), 'rouge' ou 'bleu'
    : pas de return
    ...

    #Assertions
    assert equipe_en_cours in ['bleu', 'rouge'] and equipe_perso in ['bleu', 'rouge']
    #Code
    pos_cur = self.acc_position curseur()
    if self.appuye : #si le joueur a cliqué sur quelque chose
        ##potion 1
        if 33 <= pos_cur[0] <= 121 and 286 <= pos_cur[1] <= 374 :
            if equipe_en_cours == 'bleu' and equipe_perso == 'bleu' and r
                self.attributs_jeu.mut_potion_bleue_selectionnee(1)
            if equipe_en_cours == 'rouge' and equipe_perso == 'rouge' and r
                self.attributs_jeu.mut_potion_rouge_selectionnee(1)

        ##potion 2
```

Cette méthode sélectionne la potion cliquée par le joueur. Prenons exemple pour la potion d'attaque normale.

```

## potion 1
if 33 <= pos_cur[0] <= 121 and 286 <= pos_cur[1] <= 374 :
    if equipe_en_cours == 'bleu' and equipe_perso == 'bleu' and not self.attributs_jeu.acc_potions_bleues()[1].est_vide():
        self.attributs_jeu.mut_potion_bleue_selectionnee(1)
    if equipe_en_cours == 'rouge' and equipe_perso == 'rouge' and not self.attributs_jeu.acc_potions_rouges()[1].est_vide():
        self.attributs_jeu.mut_potion_rouge_selectionnee(1)

```

Cette potion est donc la potion numéro 1. Si la position du curseur sur la fenêtre (en x et y) est sur la potion, on vérifie l'équipe qui la sélectionne. Si l'équipe qui a cliqué est celle en cours, que l'équipe du personnage passé en paramètre est la même que l'équipe en cours et que la file de cette potion n'est pas vide, alors on sélectionne la potion selon l'équipe.

```

def deselectionner_bouton(self):
    """
    Désélectionne un bouton après 0.3 secondes et lance l'action du bouton
    : pas de return
    """
    #Si le joueur a appuyé sur un bouton et que le temps après avoir appuyé est de 3 secondes :
    if self.attributs_jeu.acc_bouton_clique() != None and time.time() - self.attributs_jeu.acc_temps_appui_bouton() > 0.3:

```

Cette méthode, comme son nom l'indique, désélectionne le bouton sélectionné après 3 secondes s'il y a bien un bouton qui a été cliqué. Selon le nom du bouton, l'action est différente :

- jouer : Met les différents modes de jeu à l'écran
- local : Lance une partie en initialisant correctement les attributs, et lance donc une partie par défaut Joueur contre Joueur
- robot : Lance une partie en initialisant correctement les attributs, et lance donc une partie par défaut Joueur contre Robot
- quitter : Ferme la fenêtre Pygame
- sauvegarder : Lance la procédure pour sauvegarder la partie en cours
- charger : Lance la procédure pour charger une partie extérieure
- options : Ouvre la fenêtre du menu des options
- retour\_menu : Ferme la fenêtre du menu qui s'est affiché (exemple : Retour du bouton des Options ou le choix des Modes)
- menu : Replace le joueur dans le menu du jeu

Pour finir, il n'y a donc plus de bouton cliqué. L'attribut devient None.

```

def boutons_menu_options(self) :
    """
    Gère les différents boutons du menu options
    : pas de return
    """

```

```

def boutons_menu_modes(self) :
    ...
    Gère les différents boutons du menu modes
    : pas de return
    ...

def boutons_menu(self) :
    ...
    Gère les différents boutons du menu
    : pas de return
    ...

def boutons_jeu(self) :
    ...
    Gère les différents boutons du jeu
    : pas de return
    ...

```

Pour ces méthodes, rien de plus simple. Selon l'environnement dans lequel le joueur est, si les coordonnées correspondent à un bouton, on ajoute son nom dans l'attribut *bouton\_clique* de *attributs\_jeu* puis on mémorise à quel moment ce bouton a été cliqué avec le module time.

Remarque : Ceci ne fonctionne pas avec les On/Off des options. Il change directement l'attribut du jeu en question.

```

#Sols de couleur :
if 340 < position curseur[1] < 390 :
    self.jeu.mut_sols_de_couleur(not self.jeu.acc_sols_de_couleur())

```

Exemple : L'état du bouton On/Off de l'option “sols de couleur” change par son inverse, True devient False et False devient True.

```

#Si la position de la souris est sur la barre de volume :
if 480 <= position curseur[0] <= 775 and 580 < position curseur[1] < 627 :
    self.jeu.mut_x_pointeur(position curseur[0])
    volume_sonore = (position curseur[0] - 490)/ 300 # la bonne valeur est
    if volume_sonore < 0 :
        volume_sonore = 0
    self.musique_et_sons.mut_volume(volume_sonore)

```

En ce qui concerne la barre de volume de la méthode *boutons\_menu\_options*, on vérifie si les coordonnées du curseur ne dépasse pas la barre principale. Par la suite, on change donc le volume et les coordonnées du pointeur.

```

def quitter(self, evenement) :
    ...
    Permet de quitter le jeu
    : param evenement (input)
    : pas de return
    ...
    #Croix Rouge :
    if evenement.type == pygame.QUIT :
        self.attributs_jeu.mut_continuer(False) #Arrête la boucle du jeu

```

Cette méthode est appelée plusieurs fois dans la classe. Elle permet simplement de savoir grâce à l'événement passé en paramètre si le joueur a cliqué sur la croix rouge de la fenêtre Pygame, alors on arrête la boucle du jeu et ferme donc la fenêtre Pygame.

```

def entrees_deroulement_jeu(self) :
    ...
    Gère les différentes entrées en lien avec la partie, c'est à dire la selection d'un personnage
    : pas de return
    ...
    #Si la partie n'est pas terminée, qu'il n'y a aucun déplacement en cours et qu'il n'y a aucune attaque en
    if not self.attributs_jeu.acc_partie_terminee() and not self.attributs_jeu.acc_deplacement_en_cours() and
        #Si un personnage est déjà sélectionné :
        if isinstance(self.attributs_jeu.acc_selection(), module_personnage.Personnage) :
            self.est_clique()

        #Si un coffre est sélectionné avec un personnage :
        if self.attributs_jeu.acc_coffre_selection() is not None :
            self.jeu.coffre_est_clique()
            self.attributs_jeu.mut_coffre_selection(None) #Plus aucun coffre est sélectionné

        #Sinon sélection un personnage :
    else :
        self.personnage_est_selectionne()

```

Cette fonction gère les différentes entrées pendant une partie et est reliée aux différentes actions demandées par le joueur. Tout d'abord, on vérifie bien que la partie n'est pas terminée, qu'il n'y a aucun déplacement ou aucune attaque en cours. Puis, deux choix s'offrent au joueur :

- Si le joueur a déjà sélectionné sur un personnage, alors on appelle la méthode *est\_clique* puis on vérifie si le joueur ne voulait pas ouvrir un coffre au préalable.
- Sinon, le joueur n'a pas de personnage sélectionné et donc sélectionne le personnage où le joueur a cliqué sur le terrain grâce à la méthode *personnage\_est\_selectionne*.

```

def entrees_menu(self, evenement) :
    ...
    Contrôle les entrées du clavier et de la souris et qui donne les actions demandées dans le menu.
    : param evenement (input)
    : pas de return
    ...
    #Permet de fermer la fenêtre pygame :
    self.quitter(evenement)
    #Si un des bouton de la souris est appuyé :
    if evenement.type == pygame.MOUSEBUTTONDOWN :
        #Si le clique gauche est appuyé :
        if evenement.button == 1 :
            self.mut_appuye(True) #Appuyé est "activé"
            self.boutons_menu() #Vérifie si c'est un bouton du menu (ou du menu options en particulier)
    # Sinon, les boutons de la souris sont relâché :
    else :
        self.mut_appuye(False) #Appuyé est "désactivé"

```

Cette méthode gère également les entrées, mais quand le joueur est dans le menu du jeu. Il peut donc quitter, ou avec le clique gauche de la souris, interagir avec les boutons du menu. On retrouve ici l'attribut *appuye* qui change donc d'état en fonction si le joueur a appuyé ou non sur le clique gauche de la souris.

```

def entrees_jeu(self, evenement) :
    ...
    Contrôle les entrées du clavier et de la souris et qui donne les actions demandées dans le jeu.
    : param evenement (input)
    : pas de return
    ...
    #Permet de fermer la fenêtre pygame :
    self.quitter(evenement)
    #Si un des bouton de la souris est appuyé :
    if evenement.type == pygame.MOUSEBUTTONDOWN :
        #Si le clique gauche est appuyé :
        if evenement.button == 1 :
            self.mut_appuye(True) #Appuyé est "activé"

        #Si il n'y a pas de déplacement et/ou d'attaque de personnage en cours :
        if not self.attributs_jeu.acc_deplacement_en_cours() and not self.attributs_jeu.acc_attaque_en_cours() :
            if not self.attributs_jeu.acc_menu_options() :
                self.boutons_jeu() #Vérifie si c'est un bouton du jeu
                if not self.attributs_jeu.acc_mode_robot() or (self.attributs_jeu.acc_mode_robot() and self.entrees_deroulement_jeu())

        else :
            self.boutons_menu_options()
    # Sinon, les boutons de la souris sont relâché :
    else :
        self.mut_appuye(False) #Appuyé est "désactivé"

```

Pour finir avec la classe Souris, cette méthode gère encore les entrées, mais cette fois-ci quand le joueur est dans une partie. Il peut donc quitter, ou avec le clique gauche de la souris, interagir avec les boutons du jeu (s'il n'y a pas de déplacement, d'attaque ou d'annonce de coffre en cours) et les différentes actions de la partie de la méthode *entrees\_deroulement\_jeu* (si ce n'est pas le tour du robot quand il est activé). Le joueur peut également interagir avec le menu des options si cette dernière est ouverte.

## > LE MODULE JEU

Ce module s'occupe du déroulement du jeu. Plus généralement, on peut le concevoir comme le coordinateur des différents modules, le cerveau des opérations. Il gère le tour par tour, les interactions du joueur avec le jeu, ordonne l'exécution de l'affichage et modifie les informations du jeu en fonction de l'action effectuée.

```
class Jeu() :  
    ...  
    Une classe Jeu qui gère le calcul et l'affichage pour un bon déroulement du jeu grâce aux modules importés.  
    ...  
    def __init__(self) :  
        ...  
        Initialise le jeu  
        ...  
        #Attributs Fenêtre :  
        self.ecran = pygame.display.set_mode((1300, 800)) #Affiche l'écran du jeu de dimension 1300 x 800  
        self.horloge = pygame.time.Clock() #Permet de fixer les fps du de la fenêtre (rafraîchissement et calculs par seconde)  
        ...  
        #Attributs des Importations :  
        self.attributs_jeu = module_attributs_jeu.Attributs_Jeu()  
        self.sauvegarde = module_sauvegarde.Sauvegarde(self, self.attributs_jeu)  
        self.gestionnaire_son = module_musique_et_sons.Gestionnaire_Son()  
        self.terrain = module_terrain.Terrain(self.attributs_jeu)  
        self.souris = module_souris.Souris(self, self.attributs_jeu, self.sauvegarde, self.terrain, self.gestionnaire_son)  
        self.affichage = module_afficher.Affichage(self, self.attributs_jeu, self.terrain, self.ecran, self.souris)  
        self.robot = module_robot.Robot(self, self.attributs_jeu, self.terrain)  
        ...  
        #Options :  
        self.sols_de_couleur = True  
        self.deplacements_attaques = True  
        self.option_console = True  
        self.x_pointeur = 550 #Volume
```

Tout commence par la classe Jeu qui initialise la taille de l'écran et l'horloge de la fenêtre Pygame et toutes les autres classes des autres modules. Puis elle initialise les attributs pour le menu des options qui ne sont pas réinitialisés tant que la classe Jeu (dont la fenêtre Pygame) est en cours d'exécution.

Par la suite, des accesseurs et mutateurs sont créés pour les attributs de la fenêtre et des options seulement. Cela permet d'avoir ou modifier les attributs de la classe Jeu dans d'autres modules et d'autres classes.

```
def placer(self) :  
    ...  
    Place les personnages, monstres et coffres sur le terrain  
    : pas de return  
    ...  
    for elt in self.attributs_jeu.acc_tab_personnages() + self.attributs_jeu.acc_tab_monstres() + self.attributs_jeu.acc_tab_coffres() :  
        self.terrain.mut_terrain(elt.acc_x(), elt.acc_y(), elt)
```

La méthode *placer* place les personnages, monstres et coffres sur le terrain. Elle prend donc chaque objet venant de leur tableau respectif de la classe Attributs\_Jeu et change le contenu du terrain aux coordonnées indiquées par l'objet en question.

```
def placer_par_defaut(self) :
    ...
    Place les personnages, monstres et coffres sur le terrain (modèle par défaut)
    : pas de return
    ...
```

Pour cette méthode, c'est la même que la précédente, mais elle place les personnages, monstres et coffres d'une partie par défaut. Elle initialise chaque objet correctement puis elle les met dans leur tableau correspondant de la classe Attributs\_Jeu grâce à la méthode *placer*.

### Méthodes Console :

Pour les méthodes suivantes utilisées pour la console, elles sont toutes identiques à une seule chose près : leurs phrases. Pour faire simple, chaque méthode prend des paramètres ou non selon leur besoin et ajoute grâce à la méthode *ajouter\_console* de la classe Attributs\_Jeu la phrase correspondante et une couleur dans un tableau.

```
def partie_commence_console(self) :
    ...
    Ajoute dans la console que l'équipe a changé
    : pas de return
    ...
    self.attributs_jeu.ajouter_console(['.La partie commence !', 'noir'])
```

Exemple avec cette méthode :

### Méthode *deplacer* :

La méthode *deplacer* permet de commencer un déplacement pour le personnage sélectionné.

```
self.terrain.mut_terrain(self.attributs_jeu.acc_selection().acc_x(), self.attributs_jeu.acc_selection().acc_y(), ' ')
self.attributs_jeu.mut_nouvelles_coord((x, y))
```

Pour cela, on retire tout d'abord le personnage de la grille et dans l'attribut *nouvelles\_coord* et on met ses nouvelles coordonnées.

Une fois le personnage retiré, le reste du code de la méthode concerne les calculs pour la trajectoire du personnage pour les déplacements :

```

#Récupération du bon chemin :
perso = self.attributs_jeu.acc_selection()
coordonnees = (perso.acc_x(), perso.acc_y())
if perso.acc_personnage() == 'cavalier' : #si c'est un cavalier
    dep = self.attributs_jeu.acc_deplacements_cavalier()
else:
    dep = self.attributs_jeu.acc_deplacements()

```

Tout d'abord, on récupère les déplacements du personnage pour savoir par quelle case celui-ci pourra passer.

Remarque : les déplacements du cavalier sont spéciaux : celui-ci emprunte des cases lors de son déplacement sur lesquelles il ne peut pas s'arrêter, contrairement aux autres personnages.

```

graphe = perso.construire_graphe(coordonnees, dep)
chemin = parcourir_graphe.depiler_chemin(graphe, coordonnees, (x, y))
chemin2 = []
for elt in chemin:
    chemin2.append((elt[0] * 38 + 250, elt[1] * 38))

```

On construira ensuite un graphe représentant toutes les cases de déplacements reliées entre elles. De ce graphe, on récupérera le chemin le plus court pour atteindre, depuis les coordonnées initiales, la case de destination. On convertira ensuite les coordonnées de cases obtenues en coordonnées pour l'écran

```

self.attributs_jeu.mut_coordonnees_personnage((chemin2[0]))
self.attributs_jeu.mut_personnage_en_deplacement(perso) # on décl
self.attributs_jeu.mut_deplacement_en_cours(True) # on décl

```

Pour l'animation des déplacements, on modifie des attributs du module `attributs_jeu` puis on déclenche l'animation. On augmentera ensuite le nombre d'actions effectuées par le joueur de 1.

### Méthode arreter\_animation\_deplacement :

Dans cette méthode, on vérifie si le personnage a terminé ses déplacements dans l'animation, si oui, on remet l'avancement de déplacement à zéro et on appelle la méthode `replacer`.

### Méthode replacer :

Cette méthode replace le personnage sélectionné sur la grille aux coordonnées voulues, ici `nouvelles_coord` sauvegardées dans les variables `x` et `y`.

```

perso.deplacer(x, y) # change les attr
self.terrain.mut_terrain(x, y, perso)

```

Il pourra à nouveau être considéré comme un objet sur la grille de jeu et être

perçu par les autres personnages et monstres.

On videra donc plusieurs variables pour permettre à d'autres animations de déplacement de s'effectuer.

```
self.attributs_jeu.mut_selection(None)
self.attributs_jeu.mut_personnage_en_deplacement(None)
self.attributs_jeu.mut_deplacement_en_cours(False)
```

On appellera ensuite la méthode `jouer_monstres`, car les monstres jouent dès qu'un joueur a effectué une action.

### Méthode `deplacer_geant` :

Le géant est un personnage spécial puisque celui-ci occupe quatre cases et non une, son déplacement diffère donc de celui des personnages dits "normaux".

Une méthode annexe a donc été nécessaire pour traiter ce cas spécifique.

Le principe de cette méthode est donc la même que celui de `deplacer` sauf qu'ici, on déplace synchroniquement chaque partie/case du géant.

```
for geant in famille:
    self.terrain.mut_terrain(geant.acc_x(), geant

for geant in famille:
    n_x = geant.acc_x() + deplacement[0]
    n_y = geant.acc_y() + deplacement[1]
    geant.deplacer(n_x, n_y) # pour le personnage
    self.terrain.mut_terrain(n_x, n_y, geant)
self.jouer_monstres() #Fait jouer les monstres
```

Une méthode annexe, `coordonnees_geant` permet de savoir dans quelle direction (haut, bas, droite ou gauche) le géant se déplace et renvoie un tuple utilisé par chaque partie du géant pour se déplacer ensemble dans un sens.

### Méthode `famille_geant` :

Cette méthode est très utile. Elle renvoie en autre l'ensemble des parties d'un géant dans un tableau lorsqu'une seule de ces parties est donné/passé en paramètre.

### Méthode `gerer_animations_attaques` :

Cette méthode gère le fonctionnement de l'affichage des attaques dans le jeu.

```
##Temps de l'animation
if self.attributs_jeu.acc_cases_potions() == []:
    nb = 20
else :
    nb = 44 #animations plus longues quand il a des bulles
```

Tout d'abord, on définit un temps pour les attaques (les potions ont plus de

temps d'animation que les attaques basiques).

```
if self.attributs_jeu.acc_attaque_temps() < nb and self.attributs_jeu.acc_attaque_en_cours() :  
    self.attributs_jeu.mut_attaque_temps(self.attributs_jeu.acc_attaque_temps() + 1) #Ajoute 1 au temps de l'attaque
```

Un attribut attaque temps sera ensuite créé afin de pouvoir gérer le temps du jeu.

Celui-ci sera augmenté jusqu'à temps que l'animation d'attaque soit terminée.

```
elif self.attributs_jeu.acc_attaque_temps() == nb :  
    self.attributs_jeu.mut_attaque_en_cours(False) #Il n'y a plus d'attaque en cours  
    self.attributs_jeu.mut_attaque_temps(0) #Le temps de l'attaque est mis à 0  
    |  
    #Pour chaque personnage dans le tableau de personnages :  
    for personnage in self.attributs_jeu.acc_tab_personnages() + self.attributs_jeu.acc_tab_monstres() :  
        #Si le personnage est endommagé :  
        if personnage.acc_endommage():  
            personnage.mut_endommage() #Change son attribut en son inverse (dans ce cas True devient False)  
        #Si le personnage est soigné :  
        elif personnage.acc_soigne():  
            personnage.mut_soigne() #Change son attribut en son inverse (dans ce cas True devient False)  
        #Si le personnage a attaqué :  
        elif personnage.acc_attaque():  
            personnage.mut_attaque() #Change son attribut en son inverse (dans ce cas True devient False)  
  
    self.attributs_jeu.mut_cases_potions([]) #la potion n'agit plus
```

Si le temps de l'animation de l'attaque est terminé, on arrête toutes les animations en changeant en False tous les booléens qui les permettaient de s'afficher.

```
if self.attributs_jeu.acc_personnage_qui_attaque() :  
    self.personnages_sont_mort()  
    self.attributs_jeu.mut_personnage_qui_attaque(False)  
    self.jouer_monstres()
```

Après la fin des animations d'attaque, plusieurs méthodes sont lancées comme personnage\_sont\_mort qui permet de vérifier si un personnage est mort et si oui de rajouter une tombe. On change ensuite l'attribut pour informer d'une attaque à False. Puis, on appelle les monstres à se déplacer suite à l'action d'attaque du joueur.

## Méthodes pour les monstres :

```
def changer_etat_monstres(self) :  
    ...  
    Change l'état des monstres qui sont dans la terre quand le soleil se couche.  
    : pas de return  
    ...  
    #Si le nombre de tour est un multiple de 4 (hors 0), le monstre sort de terre :  
    if self.attributs_jeu.acc_nombre_tour() % 4 == 0 and self.attributs_jeu.acc_nombre_tour() != 0 :  
        for monstre in self.attributs_jeu.acc_tab_monstres() :  
            #Si le monstre est dans la terre :  
            if monstre.acc_etat() == 1 :  
                monstre.mut_etat(2)
```

Cette méthode change l'état des monstres sous terre quand le soleil se couche.

Pour faire simple, quand le soleil se couche, alors pour chaque monstre sous terre de la partie, on les sort de terre pour qu'il se déplace et attaque les joueurs.

```

def ajouter_tab_monstres(self) :
    """
    Ajoute des monstres avec des coordonnées aléatoires dans le tableau en fonction de combien de Nuit sont passées.
    : pas de return
    """

    #Si l'ajout de monstres est "activé" et qu'ils n'ont pas encore été ajouté alors on ajoute des monstres
    if self.attributs_jeu.acc_nombre_tour() % 8 == 2 and self.attributs_jeu.acc_monstres_active() :
        for _ in range(self.attributs_jeu.acc_nombre_monstre_a_ajoute()) :
            #Coordonnées au hasard
            x, y = self.terrain.trouver_case_libre('monstre')
            self.attributs_jeu.ajouter_monstre(module_personnage.Monstre(x, y, self.attributs_jeu.acc_pv_monstre(), 1))
    #Augmente le nombre de monstre de 1 pour la prochaine fois s'il ne dépasse pas le nombre de 4 monstres :
    if self.attributs_jeu.acc_nombre_monstre_a_ajoute() < 4 :
        self.attributs_jeu.mut_nombre_monstre_a_ajoute(1)

    #Augmente le nombre de pv des monstre de 1 pour la prochaine fois s'il ne dépasse pas le nombre de 10 pv (seule
    if self.attributs_jeu.acc_pv_monstre() < 5 :
        self.attributs_jeu.mut_pv_monstre(1)

    #Ajout de chaque monstre du tableau des monstres sur le terrain :
    for monstre in self.attributs_jeu.acc_tab_monstres() :
        self.terrain.mut_terrain(monstre.acc_x(), monstre.acc_y(), monstre)

    #Monstres déjà déplacés
    self.attributs_jeu.mut_monstres_active(False) #Active la possibilité d'interactions

    elif self.attributs_jeu.acc_nombre_tour() % 8 != 2 and not self.attributs_jeu.acc_monstres_active() :
        self.attributs_jeu.mut_monstres_active(True)

```

Cette méthode permet d'ajouter des monstres aléatoirement sur le terrain (sauf sur les ponts) quand il reste un tour à chaque équipe avant le coucher du soleil. Après ça, selon le nombre de tours passé dans la partie, le nombre de monstres qui peuvent apparaître est différent. On choisit donc une case libre aléatoire pour chaque monstre et on l'ajoute au tableau des monstres en initialisant le monstre correctement. Le nombre de monstres, après ces ajouts, augmente pour la prochaine fois (le maximum est de quatre monstres après avoir fait quelques tests). Attention, les monstres ajoutés n'ont pas les mêmes points de vie que les précédents. On augmente donc également le nombre de points de vie (PV) pour la prochaine fois (le maximum est de 5 PV après avoir fait quelques tests). Puis, on les place sur le terrain.

Remarque : L'attribut *monstres\_active* de la classe *Attributs\_Jeu* permet de ne pas toujours ajouter des monstres au même tour et évite que les monstres n'apparaissent indéfiniment. On le désactive après des ajouts puis on le réactive au prochain tour.

```

def attaquer_monstre(self, monstre) :
    """
    Le monstre passé en paramètre attaque un personnage si c'est possible.
    : param monstre (module_personnage.Monstre)
    : return (bool), True si le monstre a attaqué, False sinon.
    """

    #Assertion :
    assert isinstance(monstre, module_personnage.Monstre), 'Le paramètre doit être un monstre'
    #Code :
    monstre.atttaquer(self.terrain) #Le monstre cherche une victime à proximité

    #Si le monstre a trouvé une victime :
    if monstre.atttaquer_ennemi_proche(self.attributs_jeu.acc_equipe_en_cours()) != None:
        victime = monstre.atttaquer_ennemi_proche(self.attributs_jeu.acc_equipe_en_cours())
    #Si la victime est le Géant :
    if victime.acc_personnage() == 'geant':
        famille = self.famille_geant(victime) #détermine la famille du géant
        #Pour chaque partie du géant :
        for geant in famille:
            geant.est_attaque('monstre')
            if not geant.acc_endommage(): #si elle n'est pas déjà attaquée par un autre
                geant.mut_endommage()
    else: #autre personnage
        victime.est_attaque('monstre') #La victime perd des pv
        if not victime.acc_endommage(): #si elle n'est pas déjà attaquée par un autre
            victime.mut_endommage() #blesse la victime

        self.attributs_jeu.mut_attaque_en_cours(True)
        self.attributs_jeu.mut_attaque_temps(0)
        return True #Le monstre a attaqué

    return False #Le monstre n'a pas attaqué

```

Cette méthode permet au monstre passé en paramètre d'attaquer un personnage si cela est possible. Tout d'abord, le monstre cherche une victime proche sur le terrain. Si le monstre a trouvé une victime, on lance la procédure d'attaque. Si la victime en question est un géant, on attaque chaque partie du géant, sinon, on attaque la victime normalement. Après ça, on dit au jeu qu'il y a une attaque en cours et lance le temps d'attaque à zéro puis renvoie True pour une attaque effectuée. Sinon, renvoie False. Le monstre n'a pas pu attaquer.

```

def jouer_monstres(self):
    """
    Déplace chaque monstre sur le plateau et gère leurs attaques
    : pas de return
    """

    #Pour chaque monstre dans le tableau des monstres :
    for monstre in self.attributs_jeu.acc_tab_monstres():
        #Si le monstre était déjà sortis de terre
        if monstre.acc_etat() != 1:
            if not self.atttaquer_monstre(monstre):
                self.attributs_jeu.ajouter_monstres_a_deplacer(monstre)
                self.attributs_jeu.mut_deplacement_en_cours(True)

```

Pour finir avec les monstres, cette méthode les fait jouer. Donc, pour chaque monstre (hors de terre) du tableau des monstres, on vérifie d'abord s'il peut attaquer. Sinon, on le fait déplacer et on indique au jeu qu'il y a un déplacement en cours.

# Les 4 méthodes qui vont suivre concernent les objets du jeu, soit les coffres et les potions.

### Méthode apparition\_coffre :

Chaque fois que le jour se lève, les 4 coffres présents sur le terrain réapparaissent sur d'autres cases, qu'ils soient ouverts ou fermés ; c'est grâce à cette méthode que cela est possible.

Pour plus d'égalité, les 4 coffres ne réapparaissent pas totalement sur des cases choisies aléatoirement, grâce à ces deux lignes de code :

```
#on ajoute deux coffres en 'bas' et deux coffres en 'haut'  
self.ajouter_coffre('haut', tab)  
self.ajouter_coffre('bas', tab)
```

et à la méthode annexe `ajouter_coffre`, 2 coffres réapparaissent en 'haut' ( $y \leq 10$ ) et 2 coffres réapparaissent en 'bas' ( $y \geq 10$ ). Cela évite que, par un hasard, il y ait 4 coffres dans une partie du terrain, ce qui avantageait une des équipes.

### Méthode ajouter\_coffre :

C'est grâce à cette méthode que sont ajoutés les nouveaux coffres sur le terrain dès que le jour se lève. Elle prend en paramètres une chaîne de caractères : 'haut' ou 'bas' pour savoir dans quelle partie du terrain on doit ajouter des coffres et un tableau contenant les tuples des coordonnées des anciens coffres. Le `tab_coordees` contient les positions pré définies pour les coffres : ce sont des cases du terrain assez difficiles d'accès ou alors en plein milieu du terrain pour inciter les joueurs à se battre pour atteindre le coffre.

Cette méthode permet également de ne pas faire réapparaître les coffres sur les mêmes cases : cela permet de prendre les joueurs par surprise. De plus, si aucune des cases pré définies n'est libre, alors les coffres apparaissent aléatoirement sur le terrain.

```

trouve = False #par défaut, on n'a pas encore trouvé de case pour l'i = 0
while not trouve and i < len(tab_cordonnees): #tant qu'on n'a pas trouvé
    trouve = self.terrain.est_possible(tab_cordonnees[i][0], tab_cordonnees[i][1])
    i += 1
if trouve : #si une case est vide
    coffre = module_objets.Coffre(tab_cordonnees[i-1][0], tab_cordonnees[i-1][1])
    self.attributs_jeu.ajouter_tab_coffres(coffre) #ajout d'un nouveau coffre
    self.terrain.mut_terrain(tab_cordonnees[i-1][0], tab_cordonnees[i-1][1])
else :
    ###Coordonnées au hasard
    x, y = self.terrain.trouver_case_libre(chaine)
    coffre = module_objets.Coffre(x, y)
    self.attributs_jeu.ajouter_tab_coffres(coffre)
    self.terrain.mut_terrain(x, y, coffre) #ajout dans le terrain

```

### Méthode ouverture\_coffre :

Cette méthode est appelée lorsqu'un joueur clique sur un coffre ; une action particulière est donc réalisée suivant le contenu du coffre en question.

Remarque : le contenu du coffre est défini selon un entier naturel compris entre 1 et 10 inclus. Pour plus de détails, veuillez lire la documentation du *module\_objets*.

Par exemple, si le contenu du coffre est 10, cela signifie que tous les personnages appartenant à l'équipe adverse (par rapport au personnage qui a ouvert le coffre) voient leurs dégâts d'attaque augmenter de 2 :

```

#####
### AUGMENTATION DES DÉGÂTS DE TOUS LES PERSONNAGES ADVERSES
#####

else : #10
    equipe = self.attributs_jeu.acc_equipe_en_cours()
    for perso in ['monstre', 'mage', 'paladin', 'geant', 'sorciere', 'valkyrie',
                  'bleu'] :
        if equipe == 'bleu' :
            nouvelle_attaque = module_personnage.DIC_ATTAQUES_ROUGE[perso] + 2 #l
            module_personnage.mut_dic_attaques(perso, 'rouge', nouvelle_attaque)
        else :
            nouvelle_attaque = module_personnage.DIC_ATTAQUES_BLEU[perso] + 2 #le
            module_personnage.mut_dic_attaques(perso, 'bleu', nouvelle_attaque) #

```

### Méthode ouverture\_potion :

À l'instar de la méthode *ouverture\_coffre*, cette méthode s'occupe de réaliser l'action adéquate lorsqu'un joueur veut jeter une potion grâce à la sorcière. Selon le type de la potion (1, 2, 3 ou 4, voir *module\_objets* pour plus de détails), la sorcière attaque différemment le personnage choisi. L'attaque dépend

également de l'étendue de la potion, en effet celle-ci peut affecter plus d'une case (voir *module\_objets*).

Plus généralement, la sorcière peut soit attaquer un personnage (le nombre de pv de dégats infligés est choisi aléatoirement), soit guérir un personnage de son équipe (le nombre de pv du personnage soigné ne peut pas dépasser le nombre de pv qu'il avait en début de jeu, en effet, le personnage est soigné et non dopé), soit tuer un personnage adverse (un one-shoot en quelque sorte) ou soit faire changer d'équipe un personnage appartenant uniquement à l'équipe adverse (le rallier).

Voici par exemple la partie du code lorsque la potion sélectionnée est la 3 (mort du personnage visé) :

Remarque : la variable “cases” contient toutes les cases affectées par la potion.

```
#####
#### MORT DU PERSONNAGE
#####
elif potion.acc_contenu() == 3:
    if a_attaque :
        for case in cases :
            perso = self.terrain.acc_terrain(case[0], case[1])
            perso.est_attaque('sorciere', perso.acc_pv()) #on reti
            perso.mut_endommage()
            self.attaque_sorciere_console(equipe)
```

Ici, “if attaque” permet de savoir si une potion a bien été lancé, la variable *perso* récupère le personnage attaqué, s’ensuit une baisse de ses pv pour atteindre 0 (soit sa mort) et la mutation de son attribut *endommage*. La dernière ligne du code permet de signaler dans la console qu’un personnage est mort, tué par la sorcière.

**Méthodes Clique :**

```

def deplacement_est_clique(self) :
    ...
    Déplace le personnage en question sur le cercle selon s'il est un personnage "classique" ou s'il est un Géant.
    : return (bool) True si un déplacement a été effectué, False sinon
    ...
    position_case = self.souris.acc_position_case()

    #Si le joueur a cliqué sur un personnage de l'équipe en cours et s'il a cliqué sur un cercle de déplacement :
    if self.attributs_jeu.est_meme_equipe() and position_case in self.attributs_jeu.acc_deplacements():

        #Si le personnage est un Géant, alors déplace le Géant :
        if self.attributs_jeu.acc_selection().acc_personnage() == 'geant':
            self.deplacer_geant(position_case[0], position_case[1])

        #Sinon, le personnage est "classique" :
        else :
            self.deplacer(position_case[0], position_case[1])

        self.deplacement_console(self.attributs_jeu.acc_selection(), position_case) #Ajoute une phrase de déplacement
        self.effacer_actions()
        self.attributs_jeu.mut_nombre_action(self.attributs_jeu.acc_nombre_action() + 1) #Augmente le nombre d'actions
    return True

return False

```

Cette méthode permet de savoir si le joueur a effectué un déplacement avec un personnage ou pas. Si personnage sélectionné est de l'équipe en cours et si le joueur a cliqué sur un cercle de déplacement, alors on lance la procédure de déplacement selon si le personnage sélectionné est un géant ou non. Puis, on ajoute dans la console que le personnage en question s'est déplacé, on efface les actions et on ajoute un au nombre d'actions faites par l'équipe en cours. Après ça, la méthode renvoie True. Sinon, il n'y a pas eu de déplacement et la méthode renvoie False.

Pour la méthode *jouer\_son\_attaque*, elle est lancée simplement le son correspondant au personnage qui attaque.

```

def attaque_est_clique(self):
    ...
    attaque le personnage en question selon si il est "normal" ou si il est un geant.
    : return (bool) True si une attaque a été effectuée, False sinon
    ...

```

Cette méthode permet de savoir si le joueur a effectué une attaque avec un personnage ou pas et ressemble donc à la méthode *deplacement\_est\_clique*. Néanmoins, différentes actions sont réalisées par le jeu selon le personnage qui attaque (soit le personnage sélectionné) et le personnage qui subit (personnage cliqué par le joueur).

```

perso_qui_attaque = self.attributs_jeu.acc_selection()
#Si la sorcière attaque
if perso_qui_attaque.acc_personnage() == 'sorciere' :
    a_attaque = self.ouverture_potion(position_case[0], position_case[1])

#Si le géant attaque (attaque tous les personnages autour de lui)
elif perso_qui_attaque.acc_personnage() == 'geant' :
    for case in self.attributs_jeu.acc_attaques():
        perso = self.terrain.acc_terrain(case[0], case[1])
        perso.est_attaque('geant')
        perso.mut_endommage()
    self.attaque_console(self.attributs_jeu.acc_selection(), personnage_qui_subit) #Ajout

#Sinon, le personnage est "classique" :
else :
    personnage_qui_subit.est_attaque(self.attributs_jeu.acc_selection().acc_personnage())
    personnage_qui_subit.mut_endommage()

    self.attaque_console(self.attributs_jeu.acc_selection(), personnage_qui_subit) #Ajout

```

Si le personnage qui attaque est une sorcière, on effectue l'attaque avec la potion sélectionnée. Sinon si le personnage est un géant, le géant attaque donc chaque personnage ennemi autour de lui et ajoute dans la console que le géant a attaqué. Sinon, le personnage n'est ni une sorcière, ni un géant et donc attaque normalement avec une annonce de son attaque dans la console du jeu.

```

#Si le personnage_qui_subit est le Géant :
if personnage_qui_subit.acc_personnage() == 'geant' :
    famille = self.famille_geant(personnage_qui_subit)
    #Pour chaque partie du géant :
    for geant in famille :
        geant.est_attaque(self.attributs_jeu.acc_selection().acc_personnage())
        geant.mut_endommage()

```

Puis, on passe au personnage qui subit. Si ce personnage est un géant, alors toutes les parties du géant prennent des dégâts.

```

if a_attaque :
    #Attributs jeu
    self.attributs_jeu.mut_nombre_action(self.attributs_jeu.acc_nombre_action() + 1)
    self.attributs_jeu.acc_selection().mut_attaque() #le personnage a attaqué
    self.attributs_jeu.mut_attaque_en_cours(True)
    self.attributs_jeu.mut_attaque_temps(0)

    self.changer_personnage(' ') #Enlève le personnage sélectionné par le joueur (rien s
    self.effacer_actions()
    self.attributs_jeu.mut_personnage_qui_attaque(True)
    return True

```

S'il y a bien eu une attaque effectuée, alors on augmente de 1 le nombre d'actions de l'équipe en cours, on indique au jeu qu'il y a une attaque en cours et

lance le temps de l'attaque à zéro. Pour finir, enlève le personnage sélectionné, efface les actions, indique au jeu qu'il y a un personnage qui attaque et renvoie True.

```
def coffre_est_clique(self, coffre = None):
    ...
    si un coffre est cliqué et qu'il y a un joueur de l'équipe en cours autour du coffre, celui-ci s'ouvre
    : pas de return
    ...

    #vérification d'un joueur autour
    if coffre == None :
        coffre = self.attributs_jeu.acc_coffre_selection()
    perso_selectionne = self.attributs_jeu.acc_selection()
    #si le coffre n'a pas déjà été ouvert et personnage sélectionné de la bonne équipe
    if not coffre.acc_est_ouvert() and isinstance(perso_selectionne, module_personnage.Personnage) and perso_selectionne != None :
        present, case = coffre.est_present_autour(self.terrain, perso_selectionne)
        #si oui, ouverture du coffre
        if present :
            coffre.ouverture()
            self.ouverture_coffre(coffre, case)
            self.coffre_console(coffre.acc_contenu())

        if not self.attributs_jeu.acc_annonce_coffre() :
            self.attributs_jeu.mut_annonce_coffre(True)

        self.attributs_jeu.mut_nombre_action(self.attributs_jeu.acc_nombre_action() + 1)
        self.jouer_monstres() #on fait jouer les monstres
```

Cette dernière méthode de clique permet de savoir si le joueur veut ouvrir un coffre avec un personnage qui est proche de ce coffre ou pas. Si le coffre en question n'est pas ouvert, que le personnage n'est pas un monstre et que ce personnage est dans l'équipe en cours, on lance la procédure d'ouverture de coffre. Si le personnage sélectionné est autour du coffre, alors on ouvre le coffre et annonce dans la console son contenu. Par ailleurs, s'il n'y a pas d'annonce de coffre en cours, alors on indique au jeu qu'il y a une annonce de coffre. Pour finir, on augmente de 1 le nombre d'actions de l'équipe en cours et on fait jouer les monstres.

## Méthodes Événements :

```
def effacer_actions(self) :
    ...
    Efface les indications de déplacement et d'attaques du personnage sélectionné
    : pas de return
    ...

    self.attributs_jeu.mut_deplacements([]) #Enlève les déplacements
    self.attributs_jeu.mut_deplacements_coord([])
    self.attributs_jeu.mut_deplacements_cavalier([])
    self.attributs_jeu.mut_attaques([]) #Enlève les attaques
```

Comme écrit sur l'image, la méthode efface toutes les indications de déplacements et d'attaques de leurs attributs venant de la classe Attributs\_Jeu.

```

def changer_personnage(self, selection) :
    ...
    Enlève le personnage sélectionné et donc le change
    : param selection (?)
    : pas de return
    ...
    if isinstance(selection, module_objets.Coffre):# si c'est un coffre
        self.attributs_jeu.mut_coffre_selection(selection)
    else: #sinon
        self.attributs_jeu.mut_selection(selection)

    #Si clique un personnage ou déplacement/attaque de l'équipe qui joue :
    if selection != '' and isinstance(selection, module_personnage.Personnage) and selection.acc_personnage() != 'monstre' :
        selection.cases_valides_deplacement(self.terrain) #déplacements
        selection.cases_valides_attaques(self.terrain) #attaques

    #Sinon (si clique sur autre chose) :
    else :
        self.effacer_actions()

```

Cette méthode permet donc de changer le personnage, monstre ou case vide par celui passé en paramètre. Plusieurs choix s'offrent au joueur. Si cette sélection est un coffre, alors on le sélectionne comme un coffre et non comme un personnage (ou monstre) en changeant l'attribut *coffre\_selection* du module Attributs\_jeu. Sinon, on le sélectionne en changeant l'attribut *selection* du module Attributs\_jeu .

Si, par la suite, la sélection passée en paramètre est un personnage, on cherche ses cases de déplacements et d'attaques valides. Sinon, on efface les actions.

```

def changer_equipe(self):
    ...
    Change l'équipe en cours et enlève le personnage sélectionné
    : pas de return
    ...

    dic_equipes = {'bleu' : 'rouge', 'rouge' : 'bleu'}
    self.attributs_jeu.mut_equipe_en_cours(dic_equipes[self.attributs_jeu.acc_equipe_en_cours()])
    self.effacer_actions()
    self.changer_personnage(' ') #Enlève le personnage sélectionné
    self.attributs_jeu.mut_nombre_action(0) #Met le nombre d'action à 0
    self.attributs_jeu.augmente_nombre_tour() #Augmente le nombre de tour de 1
    self.equipe_console(self.attributs_jeu.acc_equipe_en_cours()) #Indique dans la console (du jeu

```

Comme indiqué dans le nom, cette méthode change l'équipe en cours par l'autre équipe. Par ailleurs, elle efface les actions, désélectionne le personnage, remet le nombre d'actions à zéro et indique dans la console qu'il y a eu un changement d'équipe.

```

def personnages_sont_mort(self):
    ...
    Vérifie si des personnages ou monstres sont mort (pv <= 0).
    Si c'est le cas, on les supprime du tableau correspondant et on les enlève du terrain
    : pas de return
    ...

```

Cette méthode enlève tous les personnages morts en les supprimant du tableau

des personnages, en ajoutant une tombe à leur emplacement et en le sauvegardant dans l'attribut *dernier\_personnage\_mort* suivi de l'équipe en cas de réanimation lors d'une ouverture de coffre.

Remarque : La tombe du géant est placée au milieu des quatre cases

```
def qui_gagne(self, bleu_present, rouge_present) :  
    """  
    Renvoie l'équipe qui a gagné  
    : params  
        bleu_present (bool)  
        rouge_present (bool)  
    : return (str)  
    """  
  
    #Assertions  
    assert isinstance(bleu_present, bool) and isinstance(rouge_present, bool)  
    #Code  
    gagnant = None  
    if bleu_present and not rouge_present :  
        gagnant = 'bleu'  
    elif not bleu_present and rouge_present :  
        gagnant = 'rouge'  
    elif not bleu_present and not rouge_present :  
        gagnant = 'monstres'  
    return gagnant
```

Cette méthode permet d'indiquer clairement quelle équipe a gagné la partie.

```

def est_fini(self):
    ...
    Renvoie True s'il reste que des personnages d'une même équipe, False sinon
    : return (bool)
    ...
    tab = self.attributs_jeu.acc_tab_personnages()
    indice = 0
    bleu_present = False
    rouge_present = False

    #Tant qu'un personnage de chaque équipe est vivant dans le tableau des personnages
    while not (bleu_present and rouge_present) and indice < len(tab):
        personnage = tab[indice]
        #Si le personnage est de l'équipe bleu, alors il y a au moins un personnage
        if personnage.acc_equipe() == 'bleu' :
            bleu_present = True
        #Sinon, le personnage est de l'équipe rouge, alors il y a au moins un personnage
        else :
            rouge_present = True

        indice += 1
    #Si il reste une seule équipe en jeu, alors change la partie terminée en True et
    if not (bleu_present and rouge_present) :
        self.attributs_jeu.mut_partie_terminee(not (bleu_present and rouge_present))
        equipe_qui_gagne = self.qui_gagne(bleu_present, rouge_present)
        self.attributs_jeu.mut_equipe_gagnante(equipe_qui_gagne)
        self.equipe_gagnante_console()

    return self.attributs_jeu.acc_partie_terminee()

```

Cette méthode indique au jeu si la partie est finie ou pas. Donc, tant qu'il y a un personnage par équipe qui est en vie, on ne lance pas la procédure de fin de partie et renvoie False ou si plus aucun personnage n'est en vie, cela signifie que les monstres ont gagné ! Sinon, indique au jeu que la partie est fini, l'équipe qui a gagné et annonce dans la console l'équipe gagnante. Renvoie donc True par la même occasion.

### Méthode Réinitialiser :

```

def reinitialiser_attributs(self, par_defaut = False, mode_robot = False) :
    ...
    Réinitialise quelques attributs du jeu quand le joueur charge une partie
    ...

```

Cette méthode réinitialise quelques attributs de la classe Jeu quand le joueur charge ou lance une partie. Si le paramètre “par\_defaut” est True, lance la partie normalement avec la méthode “placer\_par\_defaut” et une annonce de début de partie. Sinon, on renvoie tous les attributs. Si le paramètre “mode\_robot” est True, on indique que le robot est activé.

```

def restaurer_attributs_importation(self, tab) :
    ...
    Restaure les anciens attributs en cas de chargement
    :param tab (list of attributs)
    :return (Attributs_Jeu), pour la classe Sauvegarde
    ...
    #Assertion :
    assert isinstance(tab, list) and len(tab) == 5
    assert isinstance(tab[0], module_attributs_jeu.Attributs_Jeu)
    assert isinstance(tab[1], module_terrain.Terrain)
    assert isinstance(tab[2], module_souris.Souris)
    assert isinstance(tab[3], module_afficher.Affichage)
    assert isinstance(tab[4], module_robot.Robot),
    #Code :
    self.attributs_jeu = tab[0]
    self.terrain = tab[1]
    self.souris = tab[2]
    self.affichage = tab[3]
    self.robot = tab[4]

    return self.attributs_jeu

```

Cette méthode est appelée uniquement en cas d'erreur de chargement de fichier et donc restaure les attributs placés dans le tableau passé en paramètre et on les réaffecte à la classe Jeu.

### Méthode Boucle :

```

def boucle(self) :
    ...
    Ici on effectue tous les calculs et affichages nécessaires au jeu
    : pas de return
    ...

```

Ceci est la dernière méthode qui regroupe toutes les autres méthodes utile pour un bon déroulement du Jeu complet. C'est-à-dire la musique, les entrées, le menu, le jeu (la partie), les animations, les affichages et la fréquence d'horloge de la fenêtre Pygame. Si cette boucle est stoppée, alors cela ferme la fenêtre Pygame.

```

def jouer(self):
    """
    Initialise la fenêtre et lance le jeu avec un taux de 60 rafraîchissements/calculs par seconde
    : pas de return
    """
    pygame.init() #Initialise Pygame
    pygame.display.set_caption('Medieval Heroes') #Le nom de la fenêtre sera "Medieval Heroes"
    pygame.mouse.set_visible(False) #La souris n'est pas visible quand elle est sur la fenêtre Pygame
    pygame.display.set_icon(pygame.image.load('medias/medieval_heroes.png')) #Ajoute l'icône à la fenêtre :
    self.boucle() #Lance la boucle du jeu

```

Quand le jeu sera lancé avec le fichier “Medieval\_Heroes.py”, il importe le module\_jeu, initialise la classe Jeu et appelle cette méthode qui crée la fenêtre avec le nom de “Medieval Heroes”, met en invisible la souris, ajoute l’icône de la fenêtre et lance la boucle ! Le jeu peut démarrer !

## > LE MODULE PERSONNAGE

Ce module permet la création des personnages et des monstres. Il existe 12 types différents de personnages, tous créés à l'aide de ce module : la sorcière, le cavalier, le géant, le monstre, le mage, le cracheur de feu, la valkyrie, l'archère, le paladin, le poulet, le barbare et l'ivrogne. Chaque personnage possède ses propres caractéristiques : le nombre de pv au début du jeu, comment ils se déplacent, comment ils attaquent et combien de dégâts d'attaque ils infligent. Par conséquent, chaque personnage possède son propre visuel (voir module afficher).

Le module personnage commence par la fonction `trouve_famille_geant` très utile pour savoir toutes les cases occupées par le géant visé. Le fait qu'elle ne soit pas une méthode permet de l'appeler facilement depuis les autres modules (tout comme les deux fonctions du `module_terrain`)

S'ensuivent trois dictionnaires : un pour les pv de chaque type de personnage, un pour les dégâts d'attaque de l'équipe rouge et un pour les dégâts d'attaque de l'équipe bleu. Ceux-ci s'accompagnent d'un mutateur (utilisé dans l'ouverture des coffres, voir module jeu). L'initialisation de la classe Personnage se trouve juste après :

```

class Personnage():
    ...
    une classe pour les personnages du jeu
    ...

def __init__(self, personnage, equipe, x, y):
    ...
    initialise le personnage
    : params
        personnage (str)
        equipe (str), une des deux équipes, 'bleu' ou 'rouge'
        x (int), 0 <= x <= 20
        y (int), 0 <= y <= 20
    ...

#assertions
assert personnage in ['monstre', 'mage', 'paladin', 'geant', 'sorciere', 'valkyrie', 'archere', 'poulet']
assert equipe in ['bleu', 'rouge', 'neutre'], "l'équipe est soit bleu, soit rouge ou neutre (monstre)"
assert 0 <= x <= 20, "x ne doit pas être hors de la grille !"
assert 0 <= y <= 20, "y ne doit pas être hors de la grille !"
#code
self.personnage = personnage
self.equipe = equipe
self.x = x
self.y = y
self.pv = DIC_PV[personnage]
self.endommage = False
self.attaque = False #par défaut, le personnage n'attaque personne
self.soigne = False

```

En autre, les différents attributs sont : le personnage (représenté par une chaîne de caractères) les coordonnées (deux entiers x et y), l'équipe du personnage (une chaîne de caractères entre rouge, bleu et neutre (pour les monstres)), ses pv (entiers inférieurs ou égal à 60) et trois attributs booléens : endommage pour savoir si le personnage est attaqué, attaque pour savoir si le personnage est en train d'attaquer et soigne pour savoir si celui-ci est soigné (voir la potion de soin dans le *module\_objets*). Cette initialisation est suivie des différents accesseurs et mutateurs et de deux méthodes importantes pour les déplacements et attaques du personnage :

\* cases\_valides\_attaques :

```

def cases_valides_attaques(self, terrain):
    ...
    améliore les cases d'attaques
    Les cases valides sont les cases avec un personnage dont les
    : param terrain (Terrain)
    : return (list)
    ...

#Assertion
assert isinstance(terrain, module_terrain.Terrain), "le terrain doit être une instance de Terrain"
#Code
cases = self.cases_attaques() #les cases d'attaques par défaut
attaques_valides = [] #les cases valides finales
for attaque in cases: #on regarde chaque case
    perso = terrain.acc_terrain(attaque[0], attaque[1])
    if isinstance(perso, Personnage) and (not perso.acc.equipe == self.equipe):
        attaques_valides.append(attaque)

```

\* cases\_valides\_deplacements :

```

def cases_valides_deplacement(self, terrain):
    ...
    améliore les cases de déplacements
    Les cases valides sont :
    - Les cases vides, sans personnages ni obstacles
    - Des cases accessibles depuis le personnage sans sauts
    : param terrain (Terrain)
    : pas de return
    ...
    #Cases atteignables sans obstacles ni personnages
    dep_base = self.cases_deplacements()
    dep_sans_obstacles = self.cases_sans_obstacles(terrain, dep_base)
    dep_valides = self.cases_finales(dep_sans_obstacles)

    #Changement dans déplacements
    terrain.attribut_jeu.mut_deplacements(dep_valides) #change les cases r

```

Toutes deux utilisent plusieurs sous-méthodes pour éviter de se surcharger. En somme, elle modifie les attributs *attaques* et *déplacements* du module *attributs\_jeu*, ceux-ci deviennent des tableaux contenant les coordonnées (tuples) des cases où peuvent respectivement se déplacer et attaquer les personnages, et modifie également *coordo\_depla*, utilisé pour l'affichage. Pour le déplacement des personnages, l'utilisation d'un graphe a été nécessaire (voir *module\_graphe.py*). Cela permet au personnage d'atteindre la case où il souhaite aller sans détours, ce qui permet l'optimisation de l'affichage de son déplacement, mais aussi de savoir si une case est atteignable par un chemin.

Deux autres classes s'ajoutent à la classe Personnage : la classe Geant et la classe Monstre. Toutes deux ont été nécessaires, car ces deux types de personnage ont des particularités ; ces classes héritent donc de la classe mère qu'est la classe Personnage avec toutefois des attributs et des méthodes supplémentaires :

- Le géant a la particularité assez visible d'occuper 4 cases et non une seule. Il possède donc un attribut supplémentaire : *numero\_geant*, (0 si case en haut à gauche, 1 si case en haut à droite, 2 si case en bas à gauche et 3 si case en bas à droite). Son déplacement a donc été plus délicat : il fallait déplacer chaque case du géant synchroniquement. Son déplacement était possible si et seulement si deux cases et non une étaient libres, d'où la méthode *deplacements\_couples* qui rassemblent les cases de déplacements du géant par paire.

```

class Geant(Personnage):
    ...
    une classe pour un géant
    ...
    def __init__(self, equipe, x, y, numero_geant, pv = None):
        initialise une classe pour un géant, hérite de la classe Personnage
        : params
            equipe (str), une des deux équipes, 'bleu' ou 'rouge'
            x (int), 0 <= x <= 20
            y (int), 0 <= y <= 20
            numero_geant (int), 0 <= numero_geant <= 3
        ...
        #assertion
        assert isinstance(numero_geant, int) and -1 <= numero_geant <= 3 ,
        #code
        super().__init__('geant', equipe, x, y, pv)
        self.numero_geant = numero_geant

```

- Pour finir, le monstre. Le monstre est le seul personnage qu'aucun joueur ne peut contrôler : il choisit d'après un algorithme quelle victime attaquer et se déplace ensuite pour l'atteindre. Celui-ci a besoin de nombreux attributs pour l'affichage (voir module afficher) et d'un attribut etat, un entier valant soit 1 si le monstre se situe sous la terre (cela permet au joueur de savoir à l'avance où vont apparaître les monstres et agir en conséquence) soit 2 si le monstre est sorti de terre. Comme indiqué dans le *module\_jeu*, les monstres apparaissent sous l'état 1 deux tours avant la nuit et grandissent une fois la nuit tombée. Plus le jeu avance, plus ils sont nombreux et difficiles à battre. Ils ne meurent que si un personnage les attaque et les tue. En plus de ses accessoires et mutateurs, le monstre a besoin de méthodes lui permettant de trouver la prochaine victime qu'il va attaquer et d'attaquer celle-ci lorsqu'il se trouve juste à côté. Son déplacement se base, tout comme les personnages, sur l'utilisation d'un graphe. Faute d'avoir trouvé plus efficace, la méthode *construire\_graphe* construit un graphe à l'aide de l'ensemble du terrain et trouve le chemin le plus court pour atteindre sa victime

```

class Monstre(Personnage):
    ...
    une classe pour un monstre
    ...
    def __init__(self, x, y, pv, etat):
        initialise une classe pour un monstre
        : params
            x, y (int) avec 0 <= x, y <= 20
            0 <= pv
            etat (int), 1 = dans la terre / 2 = hors de la terre
        ...
        #assertions
        assert etat in [1, 2], "l'état doit être soit 1 soit 2 !"
        assert 0 <= pv, "le monstre doit avoir des pv positifs !"
        #code
        super().__init__('monstre', 'neutre', x, y)
        self.pv = pv
        self.etat = etat
        self.tab_victime = []
        #pour l'affichage
        ...

```

## > MODULE OBJETS

Ce module sert à la création de deux objets : les coffres et les potions. La classe Coffre s'initialise de la façon suivante :

```
class Coffre():
    ...
    une classe pour les coffres
    ...
    def __init__(self, x, y):
        ...
        initialise la classe
        : params
        x, y (int)
        ...
        self.x = x
        self.y = y
        self.contenu = self.definir_contenu()
        self.est_ouvert = False
        self.avancement_ouverture = 1
        self.dic_contenu = {1 : 'bonus de vie pour le perso
                           2 : 'changement de personnage'
                           3 : "augmente de 5 les dégâts"
                           4 : "ressuscite le dernier perso
                           5 : "ressuscite le dernier perso
                           6 : 'ajoute une potion de vie
                           7 : 'ajoute une potion de mort
                           8 : "ajoute une potion de chan
                           9 : 'ajoute une potion de mort
                           10 : "augmente de 1 les dégâts
                           }
```

Les attributs des coffres sont ses coordonnées (deux entiers, x et y), son avancement d'ouverture (entier entre 1 et 2, utilisé pour l'affichage), un attribut booléen `est_ouvert` qui permet de savoir si le coffre est ouvert ou non et enfin, un attribut `contenu` (un entier entre 1 et 10). Tous les coffres ne peuvent avoir qu'un contenu, contenu choisi au hasard parmi 10, chaque contenu de coffre ayant une chance d'apparition différente.

Voici les bonus offerts par les coffres (60% de chance d'apparition) :

- Un bonus de vie de 10 pv pour le personnage qui ouvre le coffre (9%)
- Un changement de personnage pour celui qui ouvre le coffre (ne peut pas devenir un géant) (8%)
- Une augmentation de 5 des dégâts d'attaque du personnage qui ouvre le coffre (7%)
- Une résurrection du dernier personnage mort de l'équipe du personnage qui a ouvert le coffre. S'il n'y a aucun personnage à ressusciter, le coffre change de contenu (8%)
- Ajout de deux potions de soin à la réserve de la sorcière de l'équipe du personnage qui a ouvert le coffre (14%)
- Ajout d'une potion de mort à la réserve de la sorcière de l'équipe du

personnage qui a ouvert le coffre (8%)

- Ajout d'une potion de changement d'équipe à la réserve de la sorcière de l'équipe du personnage qui a ouvert le coffre (6%)

Et voici les malus (40% chances d'apparition) :

- Une résurrection du dernier personnage mort de l'équipe adverse. S'il n'y a aucun personnage à ressusciter, le coffre change de contenu (18%)
- Ajout d'une potion de mort à la réserve de la sorcière de l'équipe adverse (9%)
- Augmentation de 2 des dégâts d'attaque de tous les personnages adverses (13%)

La méthode `est_present_autour` renvoie True si le personnage passé en paramètre se situe sur une des 8 cases autour du coffre et False sinon.

Voici l'initialisation de la classe Potion :

```
class Potion():
    ...
    une classe pour une potion
    ...
    def __init__(self, contenu):
        ...
        initialise une potion mystère
        : param contenu (int)
            • 1 : réduit les pv de l'ennemi
            • 2 : guérit le coéquipier
            • 3 : tue instantanément
            • 4 : l'ennemi change d'équipe
        ...
        #assertions
        assert isinstance(contenu, int) and contenu in [1, 2, 3, 4],
        #code
        self.contenu = contenu
        self.etendue = self.etendue_potion()
```

Il y a quatre types de potions différentes :

- les potions d'attaque qui retirent des pv au(x) personnage(s) visé(s) (bleu, numéro : 1)
- les potions de soin qui ajoute des pv au(x) personnage(s) visé(s) (vert, numéro : 2). Néanmoins, le nombre de pv d'un personnage ne peut pas dépasser le nombre de pv qu'il avait au début de la partie.
- les potions de mort qui tue le personnage visé (rouge, numéro : 3)
- les potions de changement d'équipe qui font changer d'équipe le personnage visé (jaune, numéro : 4)

Ces potions ont aussi des étendues déterminées aléatoirement par la méthode `etendue_potion`. Si la potion est une potion d'attaque ou de soin, dans 60% des cas, elle n'agit que sur la case choisie, dans 25% des cas, elle agit sur deux cases, dans 10% des cases, elle agit sur 5 cases (la case choisie et les cases en haut, en bas, à droite et à gauche) et enfin, dans 5% des cas, elle agit sur 9 cases (la case choisie plus les 8 cases autours). Les potions de mort et de changement d'équipe, trop puissantes, n'agissent que sur la case choisie.

Par défaut, chaque sorcière possède une infinité de potions d'attaque (sinon il

est possible que la sorcière se retrouve sans potion), 3 potions de soin, 1 potion de mort et 1 potion de changement d'équipe. L'unique façon de gagner des potions est d'ouvrir des coffres. Pour faciliter les choses, s'il arrive que deux sorcières se retrouvent dans la même équipe, leurs potions sont mises en commun : il n'y a qu'une seule réserve de potions.

## > DOSSIER GRAPHE

Ce dossier est composé de trois fichiers pythons : *module\_lineaire*, *parcourir\_graphe* et *module\_graphe\_dic*. Voici l'utilité de chacun :

- Le *module\_graphe\_dic* est un module utilisé uniquement pour les déplacements des personnages et la recherche de victime pour les monstres. Il permet de construire un graphe à l'aide d'un dictionnaire. La classe utilisée est la classe *Graphe\_non\_oriente\_dic* qui hérite de la classe *Graphe\_oriente\_dic* avec la méthode *ajouter\_arete* qui diffère puisque les arêtes vont dans les deux sens.

```
class Graphe_non_oriente_dic(Graphe_oriente_dic) :  
    ...  
    une classe pour un graphe non-orienté avec une matrice adjacente  
    ...  
    def __init__(self):  
        ...  
        initialise une classe pour un graphe non-orienté  
        ...  
        super().__init__()
```

La clé du dictionnaire est un sommet du graphe (ici, le sommet sera un tuple correspondant aux coordonnées d'une case du terrain) à laquelle est associée un tableau contenant les sommets adjacents/voisins au sommet en question (les cases voisines de la case-clé).

- *module\_lineaire* est utilisé pour la création des Piles et des Files, utilisées pour l'affichage de la console et les potions. Leurs méthodes principales sont : *est\_vide*, *empiler/enfiler* et *depiler/defiler*. La classe File a néanmoins la méthode *acc\_longueur* qui permet de savoir la longueur de la file (méthode utilisée pour savoir combien de potions possède une sorcière).

Extrait de la classe Pile :

```

class Pile():
    ...
    implémentation d'une pile avec des maillons
    ...
    def __init__(self):
        ...
        initialise une classe pour une pile
        ...
        self.sommet = None

    def est_vide(self):
        ...
        renvoie True si la pile est vide et False sinon
        : return (bool)
        ...
        return self.sommet == None

    def empiler(self, valeur):
        ...
        ajoute la valeur en tête de pile

```

Extrait de la classe File :

```

class File():
    ...
    une classe pour les Files
    ...
    def __init__(self):
        ...
        construit une file vide de longueur 0 possédant une tête et une queue
        ...
        self.tete = None # premier maillon de la queue
        self.queue = None # dernier maillon de la queue
        self.longueur = 0 # la longueur de la file

    def acc_longueur(self):
        ...
        renvoie l'attribut longueur
        : return (int)
        ...
        return self.longueur

    def est_vide(self):
        ...
        renvoie True si la file est vide et False sinon
        : return (bool)

```

## > LE MODULE AFFICHER

Dans le module afficher, de nombreux modules seront nécessaires car de nombreuses informations seront à prendre en compte pour l'affichage.

```

import pygame, random, module_attributs_jeu, module_clavier_souris, module_terrain, module_personnage
from graphe import module_lineaire

```

Le module de l'affichage peut se découper en trois parties :

### 1 - Les initialisations :

Toutes les images sont des attributs du module d'affichage et sont mises sous la forme de dictionnaires ou de tableaux ou simplement chargées individuellement.

Le fait qu'elles soient des attributs évite de devoir charger une nouvelle fois les images.

Sur un ordinateur avec de faibles performances, le chargement des images peut prendre une seconde environ.

```
def __init__(self, attributs_jeu, terrain, ecran, clavier_souris):
    ...
    Initialise l'affichage
    : params
        attributs_jeu (module.attributs_jeu.Attributs_Jeu)
        terrain (module_terrain.Terrain)
        ecran (pygame.display)
        clavier_souris (module_clavier_souris.Clavier_Souris)
    ...
    #Assertions :
    assert isinstance(attributs_jeu, module_attributs_jeu.Attributs_Jeu), 'attr
    assert isinstance(terrain, module_terrain.Terrain), 'terrain doit être de l
    assert isinstance(clavier_souris, module_clavier_souris.Clavier_Souris), 'c

    #Attributs des Paramètres :
    self.attributs_jeu = attributs_jeu
    self.terrain = terrain
    self.ecran = ecran
    self.clavier_souris = clavier_souris
```

*self.ecran* est un attribut très important dans *module\_afficher.py* puisqu'il s'agit de la surface sur laquelle toutes les images seront projetées, superposées les unes sur les autres. *self.ecran* est une surface de 1300\*800 pixels

## 2 - Les méthodes :

### **Méthode afficher curseur :**

La première méthode concerne l'affichage du curseur(*afficher curseur(self)*). Dans celle-ci, le curseur est affiché en fonction de l'état de la souris (appuyé ou non). Si le joueur appuie, l'image de la souris change et devient un doigt abaissé. On peut savoir si la souris est appuyé grâce à la méthode *acc\_appuye()* dans le *module\_souris*.

### **Méthode afficher\_fond :**

*afficher\_fond(self)* est une autre méthode qui permet, elle, d'afficher le fond d'écran du jeu visible sur la page d'accueil.

### **Méthode afficher\_boutons\_menu :**

`afficher_boutons_menu(self)` est une méthode permettant d'afficher les boutons dans le menu qui est la page d'accueil du jeu. La méthode `acc_bouton_clique(self)` du module attributs\_jeu est utilisée pour déterminer sur quel bouton le joueur clique. Si le bouton n'est pas cliqué, il affichera son image originelle, mais si le bouton est cliqué, il affichera sa deuxième couleur, plus claire pour signaler au joueur qu'il a bien cliqué sur ce bouton. Dans le cas où le bouton est cliqué, l'image plus claire s'affichera jusqu'à ce que la condition ne soit plus valide, soit après quelques secondes.

```
bouton_quitter = self.boutons['quitter_menu'][0]
texte_quitter = police.render("Quitter", 1, (179, 12, 36))

self.ecran.blit(bouton_quitter, (450, 580))
self.ecran.blit(texte_quitter, (597, 600))
```

Voici un exemple, de comment l'affichage d'un bouton est géré (sans les conditions) : tout d'abord, on prend dans `self.boutons` le dictionnaire des images des boutons, puis l'image que l'on souhaite ; ensuite, on crée le texte en rapport avec l'image choisie et enfin, on renseigne les informations dans la parenthèse dans cet ordre : "texte", taille, (couleur\_en\_rgb).

Avec `self.ecran.blit`, on va ensuite projeter les objets dans l'écran. D'abord l'image du bouton puis le texte, l'ordre est important, sinon le texte est en dessous du bouton et donc non/peu visible. On renseignera aussi les coordonnées (x, y dans lesquelles elles sont placées).

### Méthode `afficher_boutons_jeu` :

La méthode `afficher_boutons_jeu` est très similaire aux autres méthodes qui affichent des boutons, celle-ci sert à afficher les boutons dans le jeu en lui-même, une fois une partie lancée.

### Méthode `afficher_boutons_options` :

Le fonctionnement de la méthode `afficher_boutons_options` est assez semblable à `afficher_boutons_menu` car ils ont le même système pour les boutons. On notera une petite différence pour l'affichage de la barre de son : le pointeur de celle-ci aura une coordonnée x qui varie en fonction des réglages du joueur. Si le volume se rapproche de 0, le pointeur sera donc près de x = 475 et inversement.

```

#Barre son :
bouton_option_son = self.boutons['barre_son']
self.ecran.blit(bouton_option_son, (475, 550))

pointeur_barre = self.boutons['pointeur_son']
self.ecran.blit(pointeur_barre, (self.attributs_jeu.acc_x_pointeur(), 550))

```

### Méthode afficher\_menu\_options :

*afficher\_menu\_options* est une méthode qui charge un cadre et du texte pour le menu des options, elle appelle ensuite la fonction qui affiche les boutons des options.

### Méthode afficher\_menu\_modes:

*afficher\_menu\_modes* est une méthode qui affiche les différents modes de jeu mise à disposition du/des joueur(s) :

\* On place d'abord les images de cadres dans l'affichage :

```

#Cadres :
self.ecran.blit(self.local, (250, 250))
self.ecran.blit(self.robot, (770, 250))

```

\* Puis on initialise, les textes :

```

#Textes :
texte_joueur_bleu = police.render('Joueur', 1, (42, 51, 176))
texte_joueur_rouge = police.render('Joueur', 1, (237, 28, 36))
texte_vs = police.render('VS', 1, (152, 82, 51))
texte_robot = police.render('Robot', 1, (237, 28, 36))

```

\* On les pose ensuite sur l'écran :

```

self.ecran.blit(texte_joueur_bleu, (280, 410))
self.ecran.blit(texte_vs, (373, 440))
self.ecran.blit(texte_joueur_rouge, (400, 470))

self.ecran.blit(texte_joueur_bleu, (800, 410))
self.ecran.blit(texte_vs, (895, 440))
self.ecran.blit(texte_robot, (930, 470))

```

On place ensuite un bouton “local” et un bouton “robot” qui permettent aux joueurs de choisir le mode de jeu voulu.

Pour finir, dans cette méthode, on place un bouton retour au menu qui permet aux joueurs de retourner au menu.

### Méthode afficher\_filtre :

```
temps = self.attributs_jeu.acc_temps()
```

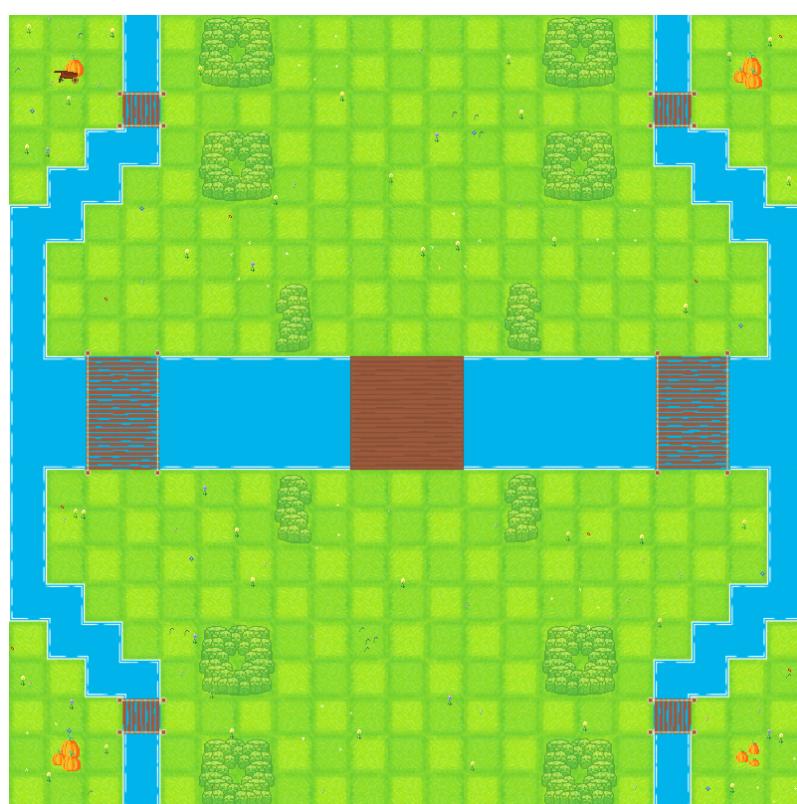
L'attribut du jeu *temps* nous permet de savoir si il fait 'Jour' ou 'Nuit' avec une chaîne de caractères.

Avec cela, on affiche ou pas un carré noir avec plus ou moins de transparence qui va se poser sur le terrain pour faire un effet de nuit.

L'attribut *self.transition* permet d'avoir une apparence progressive du carré en baissant, à chaque appel de la fonction, la transparence du carré de 1.

### Méthode afficher\_terrain :

La méthode *afficher\_terrain* permet d'afficher le terrain de jeu ci-dessous :  
(niveau 1)



Le terrain a une dimension de 800 pixels sur 800 et est constitué de 21 cases sur 21, chaque case faisant quasiment 38 pixels sur 38. Les arbres seront affichés dans une autre méthode car les personnages peuvent passer sous les côtés feuillus des arbres.

### Méthode afficher\_bandes :

`afficher_bandes` permet, elle, d'afficher les images des barres d'informations (vides) à droite et à gauche de la console.

Pour rendre le terrain plus vivant, nous avons rajouté des cerisiers.

Pour cela, il nous a fallu utiliser une image de cerisiers, créer une classe pour gérer les pétales qui tombent de l'arbre ainsi qu'un tableau qui initialise et contient tous les pétales du jeu.

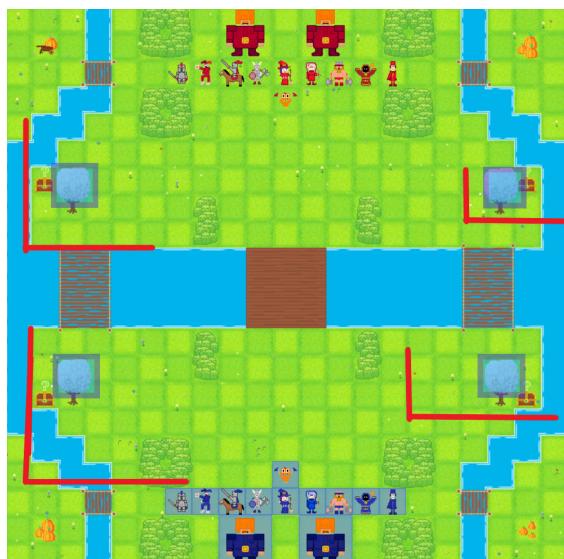
Voici plus de détail :

```
class Petale():
    ...
    Une classe Petale qui gère les petites pétales des cerisiers
    ...
    def __init__(self, position):
        ...
        Initialise la classe
        : param position (str), 'haut_gauche' ou 'haut_droit' ou 'bas_gauche' ou 'bas_droit'
        ...
        #Assertion :
        assert position in ['haut_gauche', 'haut_droit', 'bas_gauche', 'bas_droit'],
        ...
        #Attributs :
        self.taille = randint(1, 3)
        self.position = position
        ...
        if self.position == 'haut_gauche':
            self.x = randint(317, 378)
            self.y = randint(225, 278)
            self.distance_max = randint(295, 360)
```

On initialise un pétale de coordonnées x, y avec un attribut `distance_max`, étant le x et le y que le pétale ne doit pas dépasser.

La distance max est déterminée par l'emplacement du pétal mis en paramètre “position” (haut\_gauche, haut\_droit, bas\_gauche ou bas\_droit)

Il y a également un attribut `taille` qui varie et détermine la taille du pétale sur l'écran.



En bleu, vous pouvez voir la zone de génération du x et du y du pétale, et en rouge, vous pouvez visualiser des `distance_max`, une sorte de ligne invisible à une distance aléatoire de l'arbre duquel le pétale ne doit pas sortir, sinon il sera réinitialisé.

Des accesseurs et des mutateurs sont aussi codés pour accéder plus facilement aux données des pétales.

```

self.tab_petales = [
    Petale('haut_gauche'),
    Petale('haut_gauche'),
    Petale('haut_droit'),
    Petale('haut_droit'),
    Petale('bas_gauche'),
    Petale('bas_gauche'),
    Petale('bas_droit'),
    Petale('bas_droit')
]

```

Voici le tableau qui initialise les pétales au début du jeu.

```

for petale in self.tab_petales :
    petale.mut_y(1)
    petale.mut_x(-1)
    pygame.draw.rect(self.ecran, (252, 235, 237), (int(petale.acc_x()), petale.acc_y(), petale.taille, petale.taille))

#Si la pétales dépasse ce cadre, on la supprime du tableau des pétales :
if petale.acc_y() >= petale.acc_distance_max() or petale.acc_x() < 250:
    self.tab_petales.append(Petale(petale.acc_position()))
    self.tab_petales.remove(petale)

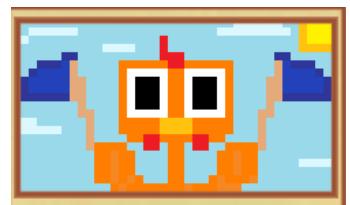
```

Chaque pétales dans le tableau descend à chaque appel de la méthode d'un pixel vers le bas et d'un pixel vers la gauche. On dessine ensuite le rectangle du pétales dans l'écran puis on vérifie qu'il n'a pas dépassé la limite, si c'est le cas, on supprime ce pétales et on génère un autre pétales avec une autre limite.

### Méthode afficher\_personnage\_selection :

La méthode *afficher\_personnage\_selection* est une méthode qui permet d'afficher dans un cadre en haut à droite de l'écran une image du personnage :

Il existe quatre cadres pour chaque type de personnage, deux cadres selon l'équipe, bleue ou rouge, un cadre pour la nuit et un cadre pour le jour.



On charge ces images dans un dictionnaire d'images : l'attribut *self.cadres\_personnages* :

Le chargement d'images dans ce dictionnaire se passe ainsi :

```

self.cadres_personnages = {
    'poulet' : [[pygame.image.load("medias/perso_selection/Jour/por.png"),
    pygame.image.load("medias/perso_selection/Jour/pob.png")]
    ....
}

```

Si on veut l'image d'un personnage rouge la nuit, on fera alors :

```
image = self.cadres_personnages[personnage][1][0]
```

Les images de la nuit sont en deuxième position dans le tableau d'où le [1] et les images de l'équipe rouge sont avant celles de l'équipe bleue d'où le [0].

Le chargement des images de tous les personnages, à l'exception du monstre qui n'a pas d'équipe, fonctionne ainsi. Pour le monstre, un tableau simple dans le dictionnaire suffit, et non pas un tableau de tableaux.

Ensuite, dans cette méthode, on affiche les informations des autres personnages.

```
perso_selection = self.attributs_jeu.acc_selection()
```

On récupérera les informations sur le personnage à partir du module attribut\_jeu.

On affichera ensuite certaines informations sur le personnage sélectionné :

```
perso_selection.acc_personnage() pour le type du personnage
```

```
perso_selection.acc_pv() pour ses PV
```

Pour la sorcière, les informations en dessous du personnage sont un peu plus poussées :

Si la sorcière est sélectionnée, on affiche sa réserve de potions. Les 4 images des potions vont donc être affichées au bon endroit avec, juste en dessous, le nombre de potions que la sorcière possède. Si il n'y a plus de potions d'un certain type dans la réserve de la sorcière, l'image de la potion est grisâtre pour le signifier au joueur. De plus, si la sorcière sélectionnée appartient à l'équipe adverse, le joueur ne peut pas savoir combien de potions de chaque type elle possède exactement.

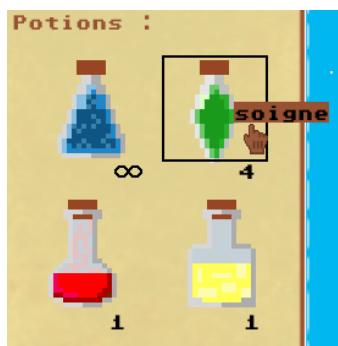
De plus, une petite infobulle et un cadre ont été implémenté : dès que le joueur sélectionne une potion, un rectangle noir se met à clignoter pour le montrer et dès que la souris se situe dans ce rectangle, une infobulle apparaît expliquant le pouvoir de la potion.

```

#####INFOBULLE
dic_phrase = {1 : 'inflige des dégâts',
              2 : 'soigne',
              3 : 'tue',
              4 : "fais changer d'équipe"
            }
pos_souris = self.souris.acc_position curseur()

if rectangle.collidepoint(pos_souris): #si la souris est sur le rectangle
    # Afficher une info-bulle
    infobulle_texte = police.render(dic_phrase[potion_s], True, (0, 0, 0))
    infobulle_rect = infobulle_texte.get_rect()
    infobulle_rect.topleft = (pos_souris[0] - 4, pos_souris[1] - 18) #un peu
    pygame.draw.rect(self.ecran, (152, 82, 51), infobulle_rect)
    self.ecran.blit(infobulle_texte, infobulle_rect)

```



### Méthode afficher équipe en cours :

`self.attributs_jeu.acc_equipe_en_cours()`

Avec cet accesseur, on peut récupérer une chaîne de caractères nous informant de l'équipe en cours, on pourra donc l'afficher dans un texte.

`self.attributs_jeu.acc_nombre_action()`

On pourra aussi ajouter dans cette méthode le nombre d'actions des joueurs, dans Medieval Heroes, les joueurs disposent de trois actions avant que ce ne soit le tour de l'équipe adverse.

### Méthode afficher console :

La méthode `afficher_console` est une méthode qui permet d'afficher les anciennes actions qui ont été effectuées dans le jeu. Que ce soit un personnage qui se déplace, un personnage qui attaque, un personnage qui meurt ou même un coffre qui s'est ouvert.

Cette méthode utilise un système de pile, ce système permet un affichage plus facile, de parcourir et d'actualiser les différents événements.

```

police = pygame.font.Font("medias/polices/police_console.ttf", 13)
pile = self.attributs_jeu.acc_console()
stock = module_lineaire.Pile()
hauteur = 570 #Hauteur où s'affiche la phrase sur la fenêtre pygame

```

Tout d'abord, on initialise les différentes valeurs qui nous seront utiles :

On initialise la police et on récupère la pile de la console dans laquelle sont stockées toutes les phrases des évènements à afficher.

On crée une pile de stockage afin de pouvoir afficher tous les éléments de la pile tout en la parcourant.

On initialise aussi une hauteur qui servira pour désigner le "y" de l'affichage du texte d'un événement.

```

while not pile.est_vide() :
    tab = pile.depiler() #Dé
    stock.empiler(tab) #Empi

```

Ensuite, on dépile la phrase et ses paramètres de la pile principale.

On sauvegarde cette phrase dans une pile de stockage annexe pour la remettre ensuite dans la pile principale plus tard et ainsi éviter de la modifier.

```
self.attributs_jeu.ajouter_console(['·La partie commence !', 'noir'])
```

Voici comment on ajoute une phrase dans la pile de la console, la phrase est donc sous forme de tableau :

```

if tab[1] == 'bleu':
    texte = police.render(tab[0], 1, (42, 51, 176))

#Si la couleur demandée est rouge, change la couleur
elif tab[1] == 'rouge':
    texte = police.render(tab[0], 1, (237, 28, 36))

#Sinon, la couleur demandée est noir, change la coule
else :
    texte = police.render(tab[0], 1, (0, 0, 0))

```

On remplira la pile des phrases de la console en dépliant la pile de stockage ensuite.

**Méthode afficher\_annonce\_coffre :**

Une fois un coffre ouvert, une annonce s'affiche à l'écran pour indiquer aux joueurs quel est le contenu du coffre.

```
self.attributs_jeu.acc_annonce_coffre()
```

On utilisera cet accesseur dans cette fonction pour savoir si l'on doit afficher un

effet et un texte.

Le filtre, ici, augmentera d'abord l'opacité du rectangle qui s'affiche sur le terrain puis, une fois son maximum atteint, baissera l'opacité pour revenir à l'état normal du jeu. L'attribut `self.direction` lui indiquera s'il doit augmenter ou diminuer l'opacité.

### Méthode afficher\_personnages :

Cette méthode affiche chaque personnage présent sur le terrain et un contour de couleur (aussi appelé sol de couleur) pour chaque personnage de l'équipe en cours si le contour de couleur est activé dans les options.

```
#Personnages/Monstres :  
for personnage in self.attributs_jeu.acc_tab_personnages() + self.attributs_jeu.acc_tab_monstres() :
```

Si le personnage est en train d'attaquer (nous n'avons dessiné pour l'instant que l'animation d'attaque de l'archère), on affichera ses différentes positions. Pour l'archère, elle tendra son arc puis le détendra.

Si le personnage est endommagé/attaqué ou soigné, on utilisera des dictionnaires spécifiques avec les personnages soit en rouge soit en vert.



L'affichage du géant est problématique étant donné qu'il fait quatre cases de large, c'est pourquoi nous avons ajouté un attribut au géant étant son numéro (de 0 à 3) qui permet d'identifier chaque partie du géant. Chaque partie correspond à une case sur le terrain. Le géant est donc coupé en quatre parties.

Si le personnage n'est pas en déplacement, qu'il n'attaque pas, n'est pas attaqué et n'est pas soigné, on affichera alors le personnage normalement :

Quand un personnage est affiché sur la grille, il est animé (il fait un léger mouvement de haut en bas, tel un PNJ), la suite met donc en place ce système d'animation.

```
images_personnage = self.personnages[nom] #Appelle les différentes images du personnage
```

Tout d'abord on récupère les images du personnage :

```
nombre_images = len(images_personnage[0])
```

On récupère aussi le nombre d'images nécessaire pour faire toutes les animations du personnage.

```
index_image = int(self.attributs_jeu.acc_compteur() / 70 * nombre_images)
```

Ensuite, on utilisera un attribut compteur, celui-ci est une boule qui va de 0 à 70 puis recommence à 0 pour déterminer quelle image de l'animation doit s'afficher.

Avec ce système, chaque image du personnage à le même temps d'affichage réparti sur 70 actualisations.

Par exemple, pour un personnage possédant une animation de 7 images, les images se succéderont en 10 “actualisations” chacune.

Pour un monstre, son affichage dépend de son état , si il est encore dans la terre (personnage.acc\_etat() vaut 1) ou s'il est sorti de terre (personnage.acc\_etat() vaut 2)

Cette méthode permet également d'afficher les coffres du jeu :

```
x = coffre.acc_x() * 38 + 250  
y = coffre.acc_y() * 38
```

Elle récupère d'abord le x et le y du coffre qu'elle convertit en coordonnées sur le terrain pour le placement de l'image du coffre.

`etat_ouvert = coffre.acc_est_ouvert()` On récupère ensuite un booléen renvoyant l'état du coffre, True s'il est ouvert et False sinon.

Si le coffre est ouvert et que son animation d'ouverture n'est pas finie, alors on la continue en passant à l'image du tableau du coffre suivante qui est une image du coffre un peu plus ouverte, sinon on le laisse ouvert jusqu'à ce que le booléen change. Si le coffre est fermé, on affiche alors l'image du coffre fermé.

### Méthode afficher\_deplacements :

Cette méthode va récupérer toutes les coordonnées dans un tableau de coordonnées comportant toutes les cases de déplacement du personnage sélectionné :

```
for coordonnees in self.attributs_jeu.acc_deplacements() :
```

On placera ensuite l'image de l'attribut `self.deplacements` pour voir le déplacement possible sur la grille.

### Méthode afficher\_personnage\_en\_deplacement :

Cette méthode affiche un personnage en déplacement sur la grille, avec des graphes obtenus grâce au module graphe.

Pour que cette méthode se lance, il faut qu'il y ait un déplacement en cours et que la selection soit vide :

```
self.attributs_jeu.acc_deplacement_en_cours() and self.attributs_jeu.acc_selection() != None :
```

Le chemin que le personnage doit parcourir sur la grille est constitué d'un tableau de coordonnées par laquelle il doit passer :

```
self.attributs_jeu.acc_chemin()
```

Son parcours est divisé en plusieurs étapes : le numéro de l'étape peut être récupéré par l'accesseur :

```
self.attributs_jeu.acc_indice_courant()
```

```
self.attributs_jeu.acc_indice_courant() < len(self.attributs_jeu.acc_chemin()) - 1
```

Le déplacement du personnage s'arrête quand il a atteint la dernière coordonnée du chemin.

```
destination = self.attributs_jeu.acc_chemin()[self.attributs_jeu.acc_indice_courant() + 1]
```

La destination du personnage est temporaire, c'est quand il doit se déplacer d'une coordonnée à une autre dans le tableau des coordonnées.

```
x, y = self.attributs_jeu.acc_coordonnees_personnage()
```

```
#Calcule le vecteur de déplacement :
```

```
dx = min(max(destination[0] - x, -3), 3)
```

```
dy = min(max(destination[1] - y, -3), 3)
```

```
#Met à jour les coordonnées du personnage:
```

```
self.attributs_jeu.mut_coordonnees_personnage((x + dx, y + dy))
```

On calcule ensuite le vecteur de déplacement du personnage dont la taille change en fonction de sa distance avec la coordonnée de la destination. (la vitesse de son déplacement est donc ici 3).

Puis une fois calculé, on l'ajoute à la position de l'image du personnage en déplacement.

On affiche ensuite cette image.

### Méthode afficher\_monstres\_en\_deplacement :

Cette méthode permet d'afficher les monstres qui sont en train de se déplacer, tout comme les personnages, cela permet d'avoir une animation fluide et plus agréable. Cette méthode utilise notamment les nombreux attributs des monstres tels que `coordonnees_x`, `coordonnees_y`, `futur_coordonnees_x`, `futur_coordonnees_y`, `futur_x` et `futur_y`. C'est le même cas de figure que la méthode précédente avec les personnages, mais la seule différence est que les monstres construisent leur chemin avec leur graphe directement dans cette méthode avec la méthode “`prochaines_coordonnees`” de la classe Monstre.

### Méthode afficher\_attaques :

Cette méthode affiche les attaques possibles du personnage sélectionné sur le plateau, attaque possible symbolisée par un petit rond rouge en haut à droite de la case.

Pour cette méthode, on aura besoin de récupérer le tableau des attaques possibles du joueur sélectionné : `self.attributs_jeu.acc_attaques()`

Une fois celles-ci récupérées, on transforme les coordonnées de cases en coordonnées pour le plateau de l'écran :

```
case_x = coordonnees[0] * 38 + 250  
case_y = coordonnees[1] * 38
```

Si le personnage sélectionné est une sorcière, cette méthode peut également afficher un rond vert avec un cœur pour montrer au joueur que la sorcière peut guérir ce personnage.

### Méthode afficher\_tombes :

La méthode afficher\_tombes, comme son nom l'indique, permet d'afficher des tombes sur les cases sur lesquels des personnages sont morts.

Pour cela, elle parcourt un tableau qui est actualisé à chaque fois qu'un personnage meurt, elle accède y grâce à cet accesseur :

```
self.attributs_jeu.acc_positions_tombes()
```

Ce tableau possède les coordonnées des endroits où il doit afficher une tombe, et elle le parcourt ainsi :

```
for tombe in self.attributs_jeu.acc_positions_tombes():  
    self.ecran.blit(self.image_tombe, (tombe[0], tombe[1]))
```

### Méthode afficher\_cases\_potions :

Cette méthode permet d'afficher un effet de bulles sur les cases où la potion a eu un effet. Comme vu dans la documentation du module\_objets, les potions peuvent parfois avec une étendue supérieure à une case, les joueurs peuvent alors le voir grâce à cet affichage.

La couleur des bulles dépend de la potion utilisée.

Un réglage a été nécessaire pour que l'animation soit naturelle, ni trop rapide ni trop lente :

```

##régagements pour l'animation
attaque1 = self.attributs_jeu.acc_attaque_temps()
attaque = attaque1 // 3
if attaque1 > 23:
    attaque -= 8

```

### Méthode afficher\_fin\_jeu :

Dans cette méthode, un menu de fin du jeu s'affichera une fois la partie terminée. Le menu de fin apparaît d'abord tout en haut de l'écran puis descend progressivement pour arriver au centre de l'écran.

```
self.attributs_jeu.acc_position_y_menu_fin()
```

L'attribut position\_y\_menu\_fin sera utilisé comme référent pour la hauteur du menu de fin.

Il y a trois menus de fin différents : si les monstres gagnent, si les rouges gagnent et si les bleus gagnent.

```
if self.attributs_jeu.acc_equipe_gagnante() == 'bleu':
```

On utilisera l'accesseur ci-dessus pour savoir quelle équipe a gagné.

```
self.attributs_jeu.acc_position_y_menu_fin() + 250
```

On placera ensuite des boutons, toujours de manière relative au y du cadre du menu.

Les boutons fonctionnent également pendant la descente du cadre.

### Méthode afficher\_position\_souris :

Cette méthode permet d'afficher la position de la souris suivante

Les abscisses seront représentées par des lettres allant de A à U et les ordonnées seront représentées par des nombres de 0 à 20.

Exemple : **Case : (F, 10)**

Il faudra d'abord vérifier pour cette méthode que la souris est bien sur le plateau de jeu et non pas sur les bandes des côtés :

```
position_case = self.clavier_souris.acc_position_case()
```

On peut récupérer la position en cases de la souris avec cet accesseur venant du module\_souris.

```
if position_case[0] >= 0 and position_case[0] <= 20 and position_case[1] >= 0 and position_case[1] <= 20:
    self.attributs_jeu.acc_dic_alphabet()[position_case[0]]
```

On associera la case avec une lettre avec le dictionnaire dic\_alphabet du module

attributs\_jeu puis on initialise le texte avec toutes les informations en chaîne de caractères puis on le projette à l'écran.

### **3 - Affichage plus global :**

À la fin du *module\_afficher*, on a les lanceurs de méthodes du module, celles-ci sont très importantes car elles permettent de lancer toutes les méthodes nécessaires au jeu, ce sont elles qui sont appelées à l'extérieur, dans *module\_jeu* principalement.

L'ordre des méthodes appelées dans celle-ci est important, en effet, la méthode appelée en dernière et celle qui s'affichera devant toutes les autres.

#### **Méthode afficher\_menu :**

Cette méthode gère l'affichage du menu, avant qu'une partie soit lancée.

Elle appellera donc dans cet ordre :

- le fond d'écran
- les options si le joueur a demandé les options
- les menus de mode de jeu si le joueur a cliqué sur "jouer"
- sinon elle affiche le menu du départ avec les boutons au centre
- elle affiche finalement le curseur du joueur

#### **Méthode afficher\_jeu :**

Cette méthode gère l'affichage du jeu, une fois qu'une partie est lancée.

Elle appellera donc les méthodes dans cet ordre :

- l'affichage des bandes sur les côtés
- l'affichage du terrain au milieu
- l'affichage des tombes
- l'affichage des personnages
- l'affichage des effets de potions
- Si l'option déplacement/attaques est activée, alors elle affiche l'aide de déplacements/attaques
- elle affiche ensuite le personnage en déplacement
- elle affiche ensuite les monstres en déplacement
- ensuite, elle affiche les cerisiers
- si la partie est terminée, elle affiche ensuite le menu de fin
- elle affiche ensuite les boutons du côté droit
- si l'option de la console est activée alors, elle affiche la console
- elle affiche ensuite les informations sur le personnage sélectionné
- l'équipe en cours

- puis les informations sur la position de la souris
- ensuite, elle affiche le filtre de la nuit si besoin
- ensuite, elle affiche une annonce du coffre si il y en a une
- si les options sont demandées, on les affiche
- puis finalement, on affiche le curseur de la souris du joueur

## > DOSSIER MEDIAS

Le dossier médias est assez conséquent puisqu'il regroupe absolument toutes les images et tous les sons du jeu, que ce soit les personnages, les différentes images utilisées pour les animations, les menus, le terrain, les coffres, la musique etc ... Voici le détail de chaque catégorie d'images :

• Le dossier *personnages* contient les dossiers portant le nom d'un personnage. Ceux-ci regroupent les images des différents personnages du jeu (de 4 images à 33 images par dossier suivant l'animation souhaitée, par exemple, le mage possède beaucoup d'images car l'animation de la boule tournant autour de lui est assez longue d'où les deux sous-dossiers *bleu* et *rouge* pour alléger visuellement le dossier).



Un sous-dossier *attaque* peut être trouvé chez l'archère : dans ceux-ci se trouvent les images utilisées pour l'animation de son attaque.



Le dossier *perso\_endommages* regroupe les images de tous les personnages colorés en rouge, utilisés lorsqu'un personnage est endommagé, tandis que le dossier *perso\_soins* regroupe, à l'inverse de *degats*, les images de tous les personnages colorés en vert, utilisés lorsqu'un personnage est soigné par une sorcière.



De plus, le dossier *perso\_en\_deplacement* regroupe les images des personnages affichés lorsque ceux-ci sont en mouvement.

Enfin, dans le dossier *perso\_selection* se trouvent toutes les images des personnages quand ils sont sélectionnés : ces images sont affichées en haut à gauche de l'écran et possèdent chacune un fond, qui diffère selon si il fait jour ou nuit et selon l'équipe du personnage,

• Dans le dossier *attaque\_deplacement*, on retrouve les ronds gris, placés sur les cases sur lesquelles un personnage peut aller lorsque celui-ci est



cliqué ; les ronds rouges d'une attaque possible indiquant que le personnage sélectionné peut attaquer un personnage ennemi et



finalement, les ronds verts, indiquant aux sorcières qu'elles peuvent guérir un personnage de leur équipe. C'est d'ailleurs dans le dossier *bulles* que se trouvent toutes les images des bulles servant à l'animation des bulles survenant lorsqu'une sorcière lance une potion (les bulles ont différentes couleurs suivant la potion utilisée),



- Le dossier *cadres* regroupent tous les cadres utilisés pour symboliser la présence d'un bouton (bouton On/Off, bouton Options, bouton Jouer, bouton Menu ...), 

$\infty$

- Les différentes images permettant l'animation de l'ouverture d'un coffre se trouvent dans le dossier *coffre*,



- Le curseur de la souris peut revêtir deux formes : une forme non appuyée et une forme appuyée, images situées dans le dossier *curseurs*,



- On retrouve dans le dossier *menu* les différentes images du menu tels que les fonds d'écran d'accueil et les menus latéraux lorsque le jeu est lancé,



- Toutes les musiques entendues lors du jeu sont rangées dans le dossier *musique* et tous les sons entendus sont dans le dossier *sons*,
- Dans le dossier *potions* sont stockées les images des différentes potions. Il y en a 4 types différents (voir module objets) différenciables par leurs différentes formes et différentes couleurs (vert, rouge, jaune et bleu). Il existe aussi des images grisâtres : cela permet de montrer visuellement au joueur qu'il n'y a plus de potions de ce type dans la réserve d'une sorcière. L'image *point\_interrogation.png* est utilisée lorsque le joueur clique sur une sorcière de l'équipe adverse : il ne peut pas savoir combien de potions il lui reste, juste savoir qu'elle a une infinité de potions bleues puisque cela ne peut pas changer d'où l'image *infini.png*,



- Les deux polices d'écriture utilisées pour le jeu sont rangées dans le dossier *polices*,
- On peut trouver dans le dossier *terrain* différentes images régulièrement utilisées : le terrain du niveau 1, les cerisiers placés sur le terrain, les tombes placées sur une case lorsqu'un personnage meurt (enlevées lorsque celui-ci est ressuscité) et les sols (carrés bleus ou rouges placés sur une case et permettant d'indiquer quels sont les personnages appartenant à l'équipe qui joue),
- Finalement, l'image *medieval\_heroes.png*, rangée dans aucun dossier, s'affiche dès le démarrage du jeu et représente sa miniature.



## > LE MODULE MUSIQUE ET SONS

```
class GestionnaireSon:  
    def __init__(self, volume=0.5):  
        ...  
        initialise le gestionnaire de sons  
        ...  
        pygame.mixer.init()  
        self.volume = volume  
        self.musique_fond = [pygame.mixer.music.load("medias/musiques/1.MP3"), pygame.mixer.music.load("medias/musiques/2.MP3")]  
        self.effets_sonores = {"feu" : pygame.mixer.Sound("medias/sons/feu.WAV"),  
                              "lame" : pygame.mixer.Sound("medias/sons/lame.WAV"),  
                              "potion" : pygame.mixer.Sound("medias/sons/potion.WAV"),  
                              "marche" : pygame.mixer.Sound("medias/sons/marche.WAV")}  
        ...  
    self.regler_volume()
```

Tout d'abord, on initialise un “mixer”, puis on initialise un gestionnaire de sons avec un paramètre/ attribut volume, un attribut musique\_fond (tableau contenant les différentes musiques du jeu) et enfin un attribut effets\_sonores qui associe dans un dictionnaire une chaîne de caractères représentant le son voulu avec un fichier son.

On appelle ensuite la méthode regler\_volume pour mettre le son à 0.2 (paramètre par défaut)

```
def regler_volume(self):  
    ...  
    ajuste le volume du jeu  
    ...  
    pygame.mixer.music.set_volume(self.volume)  
    for effet_sonore in self.effets_sonores.values():  
        effet_sonore.set_volume(self.volume)
```

Dans le mix, on change le volume directement, pour les effets sonores, on doit régler leur volume 1 à 1.

```
def mut_volume(self, volume):  
    ...  
    modifie le volume du jeu  
    ...  
    self.volume = volume  
    self.regler_volume()
```

Avec la barre de sons, il sera possible de régler le son manuellement, il nous faudra donc un mutateur pour régler le volume du son

```

def boucle_musique(self, music):
    ...
    lance la boucle des musiques de fond
    : pas de return
    ...
    if pygame.mixer.music.get_pos() == -1 :
        self.lancer_musique_fond()

```

La méthode *boucle\_musique* permet de faire tourner en boucle la musique de fond, dès que la musique est terminée (= -1), on relance la musique. Elle est régulièrement appelée dans le module\_jeu.

```

def lancer_musique_fond(self) :
    ...
    Lance la musique de fond
    :return, music
    ...
    music = self.musique_fond
    pygame.mixer.music.play()
    return music

```

La méthode *lancer\_musique\_fond*, appelée par la méthode *boucle\_musique* permet de démarrer la musique de fond.

Si l'on veut jouer un effet sonore, il faut appeler la méthode *jouer\_effet\_sonore* qui mettra en route la musique correspondant au nom de l'action, par exemple *jouer\_effet\_sonore('lame')* joue le bruit d'une lame

```

def jouer_effet_sonore(self, nom):
    ...
    joue un effet sonore
    ...
    if nom in self.effets_sonores:
        self.effets_sonores[nom].play()

```

## > LE MODULE SAUVEGARDE

Le *module\_sauvegarde* permet de sauvegarder et de charger une partie correctement par un fichier texte (.txt) :

```

class Sauvegarde() :
    ...
    Une classe Sauvegarde qui gère le chargement et la sauvegarde d'une partie dans un fichier texte
    ...
    def __init__(self, jeu, attributs_jeu) :
        ...
        Initialise la classe
        : params
            jeu (module_jeu.Jeu)
            attributs_jeu (module.attributs_jeu.Attributs_Jeu)
        ...
        #Assertions :
        assert isinstance(jeu, module_jeu.Jeu), 'jeu doit être de la classe Jeu (module_jeu) !'
        assert isinstance(attributs_jeu, module_attributs_jeu.Attributs_Jeu), 'attributs_jeu doit être de la classe Attributs_Jeu (module_attributs_jeu) !'

        #Attributs des Paramètres :
        self.jeu = jeu
        self.attributs_jeu = attributs_jeu

```

Cette classe aura donc besoin seulement du Jeu et des Attributs\_Jeu pour communiquer avec la partie.

```

def generer_chaines(self) :
    ...
    Renvoie un tableau de chaînes de caractères qui sera sauvegarder dans le fichier texte
    : return (list of str)
    ...
    tab_chaines = []

```

Cette méthode va se diviser en plusieurs étapes pour extraire les personnages et les paramètres correctement en générant plusieurs chaînes de caractères dans un tableau ordonné.

```

# Personnages :
chaine_personnages = '['
for personnage in self.attributs_jeu.acc_tab_personnages() :

    if isinstance(personnage, module_personnage.Geant) :
        chaine_personnages = chaine_personnages + '[' + personnage.acc_equipe() + ',' + str(personnage.acc_x()) + ',' + str(
            personnage.acc_y()) + ',' + str(personnage.acc_pv()) + ',' + str(personnage.acc_gauche()) + ',' + str(
            personnage.acc_droite())
    else :
        chaine_personnages = chaine_personnages + '[' + personnage.acc_personnage() + ',' + personnage.acc_equipe() + ',' +
        personnage.acc_gauche() + ',' + personnage.acc_droite()
chaine_personnages += ']'
tab_chaines.append(chaine_personnages)

```

- Les personnages :

Pour les personnages, le système de sauvegarde prend le tableau de personnages et convertit chaque personnage en un tableau converti en chaîne de caractères (str). Cela permettra de charger plus facilement la partie ou encore de modifier la sauvegarde si c'est nécessaire.

Remarque : Pour les géants, la création de ce tableau en chaîne de caractères est différente des autres personnages. Le géant est une classe différente. Donc, on ne mettra pas son nom mais simplement son équipe, ses coordonnées x et y, son numéro de géant et ses points de vie (PV).

Pour finir, on ajoute cette chaîne de caractères au tableau des chaînes de caractères.

```
# Monstres :  
chaine_monstres = '['  
for monstre in self.attributs_jeu.acc_tab_monstres() :  
    chaine_monstres = chaine_monstres + '[' + str(monstre.acc_x()) + ',' + str(monstre.acc_y()) + ',' +  
    chaine_monstres += ']  
tab_chaines.append(chaine_monstres)
```

- Les monstres :

Pour les monstres, c'est le même fonctionnement. Pour chaque monstre du tableau des monstres, on crée un tableau converti en chaîne de caractères comportant les coordonnées x et y, les PV et l'état du monstre. Pour finir, on ajoute cette grande chaîne de caractères contenant tous les monstres dans le tableau des chaînes de caractères.

```
# Coffres :  
chaine_coffres = '['  
for coffre in self.attributs_jeu.acc_tab_coffres() :  
    chaine_coffres = chaine_coffres + '[' + str(coffre.acc_x()) + ',' + str(coffre.acc_y()) + ']  
    chaine_coffres += ']  
tab_chaines.append(chaine_coffres)
```

- Les coffres :

Pour les coffres, c'est encore le même fonctionnement. Pour chaque coffre du tableau des coffres, on crée un tableau converti en chaîne de caractères comportant les coordonnées x et y du coffre. Pour finir, on ajoute cette grande chaîne de caractère contenant tous les coffres dans le tableau des chaînes de caractères.

```
# Equipe / Action / Tour / Nombre_Monstres / PV_Monstres / Robot :  
tab_chaines.append('[' + self.attributs_jeu.acc_equipe_en_cours() +
```

- Les paramètres de la partie :

Pour les paramètres, on ajoute également dans le tableau de chaînes de caractères, la chaîne de caractères suivante contenant l'équipe en cours, le nombre d'actions faite de l'équipe en cours, le nombre de tours, le nombre de monstres qui peuvent apparaître dans la partie quand c'est demandé, les pv des monstres quand on demande d'en rajouter puis True si la partie est lancé avec le robot (False sinon).

```

# Tombes :
chaine_tombes = '['
for tombe in self.attributs_jeu.acc_positions_tombes() :
    chaine_tombes = chaine_tombes + '[' + str(tombe[0]) + ',' + str(tombe[1]) + ']'
chaine_tombes += ']'
tab_chaines.append(chaine_tombes)

```

- Les tombes :

Pour chaque tombe dans le tableau des positions des tombes, on ajoute tout simplement leurs coordonnées x et y puis on ajoute cette chaîne de caractères dans le tableau de chaînes de caractères.

```

def sauvegarder(self, attributs_jeu) :
    ...
    Sauvegarde la partie
    : return (str), une phrase qui sera ajouté dans la console du jeu.
    ...
#Demande le chemin où sera sauvegardé le fichier :
self.attributs_jeu = attributs_jeu
fichier = filedialog.asksaveasfilename(
defaultextension = ".txt",
filetypes = [("Fichiers texte", "*.txt"), ("Tous les fichiers", "*.*")]
)

if not fichier :
    self.attributs_jeu.ajouter_console(["·Aucun chemin sélectionné", "noir"])

else :
    tab_chaines = self.generer_chaines()
    try :
        ecriture = open(fichier,'w',encoding='utf_8')
        for elt in tab_chaines :
            ecriture.write(elt + '\n')
        ecriture.close()
        self.attributs_jeu.ajouter_console(["·Partie Sauvegardé !", "noir"])

    except :
        self.attributs_jeu.ajouter_console(["·Erreur de Sauvegarde !", "noir"])

```

Cette méthode est la suite de la méthode précédente et qui va permettre de sauvegarder une fois pour toute la partie dans un fichier texte. Elle prend en paramètre les attributs du jeu pour être sûr d'avoir la même partie. Ensuite, on demande à l'utilisateur de sauvegarder son fichier .txt où il le souhaite avec un nom grâce au module tkinter qui affiche les dossiers de l'utilisateur. Si l'utilisateur a fermé la fenêtre tkinter, il y a une annonce dans la console qui explique qu'aucun chemin n'a été sélectionné. Sinon, la sauvegarde se réalise : la méthode génère un tableau contenant des chaînes de caractères pour chaque objet important de la partie. On écrit dans le fichier donné par l'utilisateur la partie en UTF-8. Donc, pour chaque chaîne de caractères dans le tableau de

chaînes de caractères, on écrit ligne par ligne l'objet dans le fichier. Si il n'y pas de problème, une annonce dans la console du jeu montre que la partie est bien sauvegardée. Sinon, s'il y a un problème, une annonce dans la console du jeu explique qu'il y a eu un problème quand le joueur a essayé de sauvegarder la partie.

```
def convertir_chaine_list(self, chaine) :  
    """  
        Convertit la chaîne de caractères passé en paramètre en tableau  
        : param chaine (str)  
        : return (list)  
    """  
  
    #Assertion  
    assert isinstance(chaine, str), "la chaîne doit être de type str"  
    #Code  
    elt_important = ''  
    tab_important = []  
    tab_complet = []  
    for elt in chaine :  
        if elt == ',' :  
            tab_important.append(elt_important)  
            elt_important = ''  
        elif elt == ']' :  
            if elt_important != '' :  
                tab_important.append(elt_important)  
                tab_complet.append(tab_important)  
            elt_important = ''  
            tab_important = []  
        elif elt != '[' :  
            elt_important += elt  
    return tab_complet
```

La méthode suivante convertit une chaîne de caractères en un tableau de type list.

*elt\_important* garde en mémoire l'élément qui sera ajouté dans le *tab\_important*.  
*tab\_important* garde en mémoire le tableau qui sera ajouté dans le *tableau\_complet*, qui est le tableau qui sera renvoyé à la fin du programme.

Le fonctionnement de la méthode est simple :

On regarde chaque caractère dans la chaîne de caractères passé en paramètre :  
\* si le caractère est une virgule, c'est-à-dire que l'élément est fini, alors on l'ajoute dans le *tableau\_important* comme un élément à part entière.

- \* Sinon, si le caractère est un crochet fermé et que l'élément important n'est pas vide, c'est donc que le tableau est fini. On ajoute le dernier élément important dans le tableau important puis on ajoute le tableau important dans le tableau complet. On initialise donc leur valeur par défaut.
- \* Sinon si le caractère n'est pas un crochet ouvert, on ajoute tout simplement le caractère avec les autres caractères importants pour former l'élément important.

Pour finir, on renvoie le tableau complet.

```
def restaurer_partie(self, tab) :
    ...
    Restaure la partie grâce au tableau passé en paramètre
    : param tab (list)
    ...
    #Assertion
    assert isinstance(tab, list), 'Le paramètre doit être un tableau (list) !'
    #Code
    tab_converti = []
    for chaine in tab :
        tab_converti.append(self.convertir_chaine_list(chaine[:-1]))
```

Cette méthode, comme son nom l'indique, restaure la partie. Elle prend en paramètre le tableau contenant les tableaux en chaîne de caractères. Tout d'abord, elle convertit donc ces chaînes de caractères en tableaux. Puis, elle se divise en plusieurs étapes :

```

# Personnages :
tab_personnages = tab_converti[0]

geant_bleu = []
geant_rouge = []

for personnage in tab_personnages :
    #géant
    if personnage[0] in ['bleu', 'rouge'] :
        geant = module_personnage.Geant(personnage[0], int(personnage[1]),
                                         self.attributs_jeu.ajouter_personnage(geant))
        if personnage[0] == 'rouge' :
            geant_rouge.append(geant)
        else :
            geant_bleu.append(geant)
    #personnage "normal"
    else :
        self.attributs_jeu.ajouter_personnage(module_personnage.Personnage(
            personnage[0], int(personnage[1]), self.attributs_jeu))

if len(geant_bleu) == 4 :
    self.attributs_jeu.ajouter_famille_geant_bleu(geant_bleu)
    geant_bleu = []

if len(geant_rouge) == 4 :
    self.attributs_jeu.ajouter_famille_geant_rouge(geant_rouge)
    geant_rouge = []

```

- Les personnages :

Pour les personnages, on initialise le tableau de personnage qui va servir à créer les personnages pour la partie. Les tableaux geant\_bleu et geant\_rouge servent à créer correctement les familles de géant. Donc pour chaque tableau de personnage dans le tableau personnages, si le premier élément est une couleur, alors le personnage est un géant et alors, on l'initialise correctement, on l'ajoute dans le vrai tableau de personnages venant des attributs du jeu et on l'ajoute dans la famille de sa couleur. Sinon, le personnage n'est pas un géant et donc on initialise directement avec la classe Personnage puis on l'ajoute dans le tableau des personnages de l'attributs\_jeu. Par ailleurs, si une famille de géant d'une couleur est complète, on l'ajoute dans le tableau de famille de géant.

```

# Monstre :
tab_monstres = tab_converti[1]
self.attributs_jeu.mut_tab_monstres([])
for monstre in tab_monstres :
    self.attributs_jeu.tab_monstres.append(module_personnage.Monstre(int(monstre[0]),
                                                                     self.attributs_jeu))

```

- Les monstres :

Pour les monstres, c'est moins compliqué. Pour chaque tableau de monstre du tableau converti d'indice 1, on ajoute l'objet de la classe Monstre avec ses

paramètres (coordonnées / points de vie / état) dans le tableau des monstres de l'attributs\_jeu.

```
# Coffre :  
tab_coffres = tab_converti[2]  
self.attributs_jeu.mut_tab_coffres([])  
for coffre in tab_coffres :  
    self.attributs_jeu.tab_coffres.append(module_objets.Coffre(int(coffre[0]), int(coffre[1])))
```

- Les coffres :

Pour chaque tableau de coffre du tableau converti d'indice 2, on ajoute l'objet de la classe Coffre avec ses coordonnées dans le tableau des coffres de l'attributs\_jeu.

```
# Equipe / Action / Tour / Nombres_Monstres / PV_Monstres / Robot :  
tab_param = tab_converti[3][0]  
  
self.attributs_jeu.mut_equipe_en_cours(tab_param[0])  
self.attributs_jeu.mut_nombre_action(int(tab_param[1]))  
self.attributs_jeu.mut_nombre_tour(int(tab_param[2]))  
self.attributs_jeu.mut_nombre_monstre_a_ajoute(int(tab_param[3]))  
self.attributs_jeu.mut_pv_monstre(int(tab_param[4]))  
  
dic = {'True' : True, 'False' : False}  
self.attributs_jeu.mut_mode_robot(dic[tab_param[5]])  
  
#Si il y a eu un multiple de 4 tours passé :  
if self.attributs_jeu.acc_nombre_tour() % 8 in [4, 5, 6, 7] and self.attributs_jeu.acc_nombre_tour() != 0 :  
    self.attributs_jeu.mut_temps('Nuit')
```

- Les paramètres de la partie :

Pour les paramètres, on reprend tout simplement les mêmes positions que dans la méthode generer\_chaines. Par contre, pour savoir s'il faisait Jour ou Nuit, on peut le savoir avec le nombre de tours passés sur 8. Si le nombre de tours est 4, 5, 6 ou 7, c'est-à-dire que la partie a été sauvegardée pendant la Nuit et donc change le temps.

```
# Tombe :  
tab_tombes = tab_converti[4]  
self.attributs_jeu.mut_positions_tombes([])  
  
for tombe in tab_tombes :  
    self.attributs_jeu.ajouter_positions_tombes((int(tombe[0]), int(tombe[1])))
```

- Les tombes :

Pour chaque tombe du tableau converti d'indice 4, on ajoute dans le tableau positions\_tombes la tombe.

```

def charger(self) :
    """
    Charge la partie
    : pas de return, effet de bord
    """

    #Demande le chemin du fichier qui sera chargé :
    fichier = filedialog.askopenfilename(
        filetypes = [ ("Fichiers texte", "*.txt"), ("Tous les fichiers", "*.*") ]
    )

    if not fichier :
        self.attributs_jeu.ajouter_console(["·Aucun fichier sélectionné", "noir"])

    else :
        try :
            attributs_importation, nouveau_attributs_jeu = self.jeu.reinitialiser_attributs() #R
            self.attributs_jeu = nouveau_attributs_jeu #Donne les nouveaux attributs du jeu
            lecture = open(fichier,'r',encoding='utf_8')
            tab = lecture.readlines()
            lecture.close()
            self.restaurer_partie(tab) #Charge la partie du fichier texte
            self.jeu.placer()
            self.attributs_jeu.mut_menu(False)
            self.attributs_jeu.ajouter_console(["·Partie Chargée !", "noir"])

        except :
            self.attributs_jeu = self.jeu.restaurer_attributs_importation(attributs_importation)
            self.attributs_jeu.ajouter_console(["·Erreur de Chargement !", "noir"])

```

Pour finir, cette méthode sert définitivement à charger la partie de l'utilisateur. Tout d'abord, on demande à l'utilisateur le chemin du fichier texte (.txt) via une fenêtre de ces dossiers grâce à tkinter. Si l'utilisateur a fermé la fenêtre tkinter (sans sélectionner de fichier), annonce dans la console du jeu qu'aucun fichier n'a été sélectionné. Sinon, on essaye de charger la partie correctement par les étapes suivantes :

1. Réinitialise les attributs du jeu pour repartir de zéro.
2. Modifie l'attribut “attributs\_jeu” de la classe Sauvegarde par le nouveau.
3. Lit le fichier de l'utilisateur.
4. Toutes les lignes du texte sont mises dans un tableau.
5. Restaure la partie grâce au tableau.
6. Place les personnages sur le terrain.
7. Dit au jeu qu'on est dans une partie et pas dans le menu
8. Annonce dans la console du jeu que la partie est bien chargée.

Sinon, il y a eu un problème avec le fichier sélectionné par l'utilisateur, on restaure donc les anciens attributs et informe dans la console du jeu qu'une erreur est survenue lors du chargement de la partie.

## > LE MODULE ROBOT

Ce module initialise un robot qui joue contre le joueur si le mode “Joueur VS Robot” est sélectionné. Le robot joue automatiquement les personnages rouges.

```

class Robot():
    ...
    Une classe pour le robot
    ...

    def __init__(self, jeu, attributs_jeu, terrain):
        ...
        Initialise le robot
        : params
            jeu (module_jeu.Jeu)
            attributs_jeu (module_attributs_jeu.Attributs_Jeu)
        ...

        #assertions :
        assert isinstance(jeu, module_jeu.Jeu), "jeu doit venir de la classe Jeu"
        assert isinstance(attributs_jeu, module_attributs_jeu.Attributs_Jeu),
        assert isinstance(terrain, module_terrain.Terrain), 'terrain doit être'

        #Attributs Paramètres:
        self.jeu = jeu
        self.attributs_jeu = attributs_jeu
        self.terrain = terrain

        #Attributs :
        self.en_attente = False
        self.temps_attente = None

```

Le robot a besoin des classes Jeu, Attributs\_Jeu et Terrain pour fonctionner. L'attribut `en_attente` permet de savoir si le robot peut jouer ou s'il est en attente/en pause. L'attribut `temps_attente` permet de calculer combien de temps il est en pause. Si le robot est en pause, l'attribut vaut `None`. Pour ses attributs, nous allons voir plus en détail leurs utilités plus tard.

```

def acc_tab_personnages_rouges(self) :
    ...
    renvoie le tableau avec tous les personnages rouges dans le désordre
    : return (list)
    ...

    tab_personnages_rouges = []
    for perso in self.attributs_jeu.acc_tab_personnages() :
        if perso.acc_equipe() == 'rouge' and perso.acc_pv() > 0 :
            if perso.acc_personnage() != 'geant' or (perso.acc_personnage() == 'geant' and perso.acc_numero_geant() == 0) :
                tab_personnages_rouges.append(perso)
    random.shuffle(tab_personnages_rouges)
    return tab_personnages_rouges

```

Cette méthode renvoie tout simplement un tableau avec comme éléments les personnages de son équipe (de l'équipe rouge). Elle n'ajoutera pas dans ce tableau les personnages censés être morts ou encore les autres parties du corps d'un géant. Elle prend donc uniquement le numéro 0 (`acc_numero_geant = 0`) du géant pour chaque géant. Pour finir, avant de la renvoyer, elle mélange une fois le tableau pour que le robot ne choisisse pas toujours le même personnage en boucle par la suite, mais le choisisse au hasard.

## 1 - Déplacements

```
def ennemi_proche(self, allie, une_case_proche = False) :  
    ...  
    Le robot regarde s'il y a un ennemi proche de son personnage allie (dans les cases d'attaques possibles)  
    : params  
        allie (module_personnage.Personnage)  
        une_case_proche (bool), par défaut vaut False  
    : return (module_personnage.Monstre)  
    ...
```

La méthode suivante à plusieurs utilités. Elle va chercher un ennemi proche du personnage rouge dit *allie* puis le renvoyer. Cependant, l'ennemi pourra servir soit de proie, soit alerter le robot de bouger un personnage en priorité.

```
#tableau cases  
if allie.acc_personnage() == 'geant' :  
    tab = [(0, -1), (-1, 0), (-1, 1), (1, -1), (0, 2), (2, 0), (2, 1), (1, 2), (-1, -1), (-1, 2), (2, -1), (2, 2)]  
    tab = module_terrain.tuples_en_coordonnées((allie.acc_x(), allie.acc_y()), tab, allie.acc_numero_geant())  
  
else :  
    if not une_case_proche :  
        tab = allie.cases_valides_attaques(self.terrain)  
  
    else :  
        tab = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]  
        tab = module_terrain.tuples_en_coordonnées((allie.acc_x(), allie.acc_y()), tab)
```

Avant de regarder s'il y a un ennemi à proximité, le robot va déterminer les cases à rechercher par un tableau. Selon si le personnage allié est un géant ou si c'est tout sauf un géant, le tableau ne sera pas le même. Par conséquent, si le personnage allié n'est pas un géant, on regarde soit les cases proches ou les cases d'attaques du personnage allié.

```
ennemi = None  
i = 0  
  
### Regarde autour du personnage sélectionné (allie) :  
while i < len(tab) and ennemi == None :  
    perso = self.terrain.acc_terrain(tab[i][0], tab[i][1])  
  
    if allie.acc_personnage() == 'sorciere' and isinstance(perso, module_personnage.Personnage) and  
        ennemi = perso #Pour le soigner (même équipe)  
  
    elif isinstance(perso, module_personnage.Personnage) and perso.acc_equipe() != 'rouge' :  
        if isinstance(perso, module_personnage.Monstre) and perso.acc_etat() != 1 :  
            ennemi = perso  
        else :  
            ennemi = perso  
  
    i += 1  
return ennemi
```

Le robot regarde désormais chaque case du tableau tant qu'il n'a pas trouvé d'ennemi. Si le personnage allié (de l'équipe rouge) est une sorcière, que la case comporte un personnage de l'équipe rouge et qu'il a entre 1 et 4 pv compris, alors le personnage de la case est un ennemi. Cela servira à le soigner. Sinon si la case comporte bel et bien un personnage de l'équipe bleu ou un monstre, alors c'est un ennemi.

Remarque : si c'est un monstre sous-terre ("etat" valant 1), alors le robot ne peut pas l'attaquer. Pour finir, on renvoie l'ennemi.

Bien sûr, si aucun ennemi a été trouvé, alors la méthode renverra *None*.

Pour la prochaine méthode, elle sera divisée en plusieurs parties :

- Présentation :

```
def choisir_personnage_deplacement(self, tab_personnages_rouges) :  
    ...  
    Regarde pour chacun de ses personnages s'il y un ennemi pas loin ou un coffre.  
    Sinon, déplace un personnage aléatoirement  
    : param tab_personnages_rouges (list)  
    : return (list)  
    ...  
  
#Assertions  
assert isinstance(tab_personnages_rouges, list), "le tableau des personnages rouges doit être un tableau"  
#Code  
tab_perso_case = []
```

Cette méthode sert au robot pour choisir intelligemment un personnage à déplacer. Elle prend donc en paramètre le tableau de personnages rouge venant de la méthode acc\_tab\_personnages\_rouges. On définit un tableau tab\_perso\_case qui va contenir comme premier élément le personnage que le robot va déplacer et comme deuxième élément la case sur laquelle le personnage va se déplacer.

Le robot peut alors faire 4 actions, la première étant considérée comme la plus importante, intelligente à faire et la dernière comme l'action de secours :

- Ennemi Proche :

```
### MONSTRE PROCHE :  
i_personnage = 0  
while i_personnage < len(tab_personnages_rouges) and tab_perso_case == [] :  
    personnage = tab_personnages_rouges[i_personnage]  
    self.jeu.changer_personnage(personnage)  
    ennemi_proche = self.attacter_ennemi_proche(personnage, True) #on regarde si il y a un ennemi à proximité  
  
    if ennemi_proche != None : #si il y a un ennemi à attaquer  
        case = random.choice(self.attributs_jeu.acc_deplacements()) #on se déplace au hasard  
        tab_perso_case = [personnage, case]  
  
    i_personnage += 1
```

Premièrement, le robot cherche s'il y a un ennemi pas loin de ses personnages. Dès qu'il y a en un suffisamment proche, il ajoute le personnage et la case en question (une case aléatoire). Si le robot a fait tous ses personnages et qu'il n'a pas trouvé d'ennemi proche, alors il passe à la seconde partie du code.

- Ennemi à Attaquer :

```

### ATTAQUER ENNEMI :
if tab_perso_case == [] : #si aucun ennemi proche

    i_personnage = 0
    while i_personnage < len(tab_personnages_rouges) and tab_perso_case == [] :
        personnage = tab_personnages_rouges[i_personnage]
        self.jeu.changer_personnage(personnage)

    i_case = 0
    while i_case < len(self.attributs_jeu.acc_deplacements()) and tab_perso_case == [] :

        if personnage.acc_personnage() == 'geant':
            perso_annexe = module_personnage.Geant(personnage.acc_equipe(), self.attributs_jeu.acc_deplacements)
        else:
            perso_annexe = module_personnage.Personnage(personnage.acc_personnage(), personnage.acc_equipe(), s

    ennemi_a_attaquer = self.atttaquer_enemis_proche(perso_annexe)

    if ennemi_a_attaquer != None :
        tab_perso_case = [personnage, self.attributs_jeu.acc_deplacements()[i_case]]

    i_case += 1
    i_personnage += 1

```

Deuxièmement, le robot cherche une nouvelle fois s'il y a un ennemi qui pourrait attaquer avec chaque personnage de son équipe. Pour cela, il crée un double personnage qu'on appelle `perso_annexe`. Le robot va donc, pour chaque déplacement possible avec le personnage sélectionné, créer ce personnage annexe avec des coordonnées x et y d'une case de déplacement possible du personnage principal. Dès qu'il a trouvé une case dans laquelle le personnage sélectionné pourra attaquer un ennemi, alors, le personnage et la case sont enregistrés. Sinon, s'il n'y a aucune case possible qui entraîne par la suite une attaque possible avec un personnage du robot, le robot passe à la troisième solution.

- Chercher un Coffre :

```

### CHERCHER COFFRE :
if tab_perso_case == [] : #si aucun ennemi à attaquer

    i_personnage = 0
    while i_personnage < len(tab_personnages_rouges) and tab_perso_case == [] :
        personnage = tab_personnages_rouges[i_personnage]
        self.jeu.changer_personnage(personnage)
        i_coffre = 0
        while i_coffre < len(self.attributs_jeu.acc_tab_coffres()) and tab_perso_case == [] :
            coffre = self.attributs_jeu.acc_tab_coffres()[i_coffre]
            if not coffre.acc_est_ouvert() :
                alentour_coffre = module_terrain.cases_around((coffre.acc_x(), coffre.acc_y()))
                i_case = 0
                while i_case < len(self.attributs_jeu.acc_deplacements()) and tab_perso_case == [] :
                    if self.attributs_jeu.acc_deplacements()[i_case] in alentour_coffre :
                        tab_perso_case = [personnage, self.attributs_jeu.acc_deplacements()[i_case]]
                    i_case += 1
                i_coffre += 1
            i_personnage += 1

```

Troisièmement, le robot va donc regarder s'il y a un personnage avec un coffre fermé à proximité. Pour ce faire, le robot regarde pour chaque personnage et pour chaque coffre fermé s'il y a une case aux alentours du coffre et une case de déplacement de ce personnage sélectionné en commun. S'il n'y en a pas, alors il va passer à la dernière solution possible.

- Déplacement Aléatoire :

```

### DEPLACEMENT ALÉATOIRE AVEC PERSO ALÉATOIRE :
if tab_perso_case == [] :

    case_correcte = False
    case = None
    while not case_correcte :

        personnage = random.choice(tab_personnages_rouges)
        self.jeu.changer_personnage(personnage)
        ##le personnage doit pouvoir se déplacer
        while self.attributs_jeu.acc_deplacements() == [] :
            personnage = random.choice(tab_personnages_rouges)
            self.jeu.changer_personnage(personnage)

        #Choisit une case aléatoire :
        case = random.choice(self.attributs_jeu.acc_deplacements())

        if personnage.acc_y() <= 4 and case[1] - personnage.acc_y() > 0 : #En haut du terrain
            case_correcte = True
        elif personnage.acc_y() >= 16 and personnage.acc_y() - case[1] < 0 : #En bas du terrain
            case_correcte = True
        elif 4 < personnage.acc_y() < 16 : #Au milieu du terrain
            case_correcte = True

    tab_perso_case = [personnage, case]

```

Cette dernière solution est possible la plupart du temps en début de partie. Cela ne veut pas dire que le robot va faire des déplacements aléatoires, mais il va les faire intelligemment si aucune autre solution existe. Donc pour ce faire, il va prendre un de ses personnages aléatoirement et le sélectionne. Il va regarder s'il

peut au moins se déplacer, sinon il reprend un personnage aléatoirement jusqu'à temps qu'un personnage est déplaçable. Puis, le robot prend une case de déplacement possible de ce personnage et regarde cette case : si elle lui fait aller vers le bas (alors que le personnage est en haut) ou si cette case lui fait aller vers le haut (alors que le personnage est en bas) ou encore si le personnage est vers le milieu du terrain alors cette case est automatiquement validé. En somme, le robot aura au moins une case trouvée intelligemment.

```
def deplacer_personnage(self, tab_personnages_rouges) :
    """
    Le robot choisit le meilleur personnage à déplacer et le déplace
    : param tab_personnages_rouges (list)
    : pas de return
    """

    #Assertions
    assert isinstance(tab_personnages_rouges, list), "le tableau des personnages
    #Code
    tab_perso_case = self.choisir_personnage_deplacement(tab_personnages_rouges)

    try :

        if self.attributs_jeu.acc_selection().acc_personnage() == 'geant' :
            self.jeu.deplacer_geant(tab_perso_case[1][0], tab_perso_case[1][1])

        else :
            self.jeu.deplacer(tab_perso_case[1][0], tab_perso_case[1][1])

        self.jeu.effacer_actions()
        self.jeu.deplacement_console(tab_perso_case[0], tab_perso_case[1]) #Ajou

    except :
        self.deplacer_personnage(tab_personnages_rouges)
```

La méthode *choisir\_personnage\_attaquer* permet de déplacer une fois pour toute un personnage. Elle prend bien évidemment le tableau de personnages de son équipe en paramètre et ne renvoie rien.

Tout d'abord, le robot choisit intelligemment un personnage à déplacer. Si le personnage est un géant, alors il appelle la méthode pour déplacer le géant à la case x et y du tableau perso\_case. Sinon, le personnage n'est pas un géant et donc déplace également à la case x et y. Après le déplacement, on efface correctement les actions et on indique dans la console le déplacement effectué par l'équipe rouge (par le robot). Remarque : Le “try / except” sert uniquement s'il y a un problème au cours du déplacement avec le chemin vers la case. Si c'est le cas, on rappelle la méthode *deplacer\_personnages*.

```

def choisir_personnage_attaquer(self, tab_personnages_rouges) :
    """
    Le robot choisit le personnage allié avec lequel il va attaquer
    : param tab_personnages_rouges (list)
    : return (list)
    """

    #Assertions
    assert isinstance(tab_personnages_rouges, list), "le tableau des personnages rouge n'est pas une liste"
    #Code
    tab_allie_ennemi = []

    i_personnage = 0

    ### Pour chaque personnage de son équipe tant qu'il n'a pas de personnage avec
    while i_personnage < len(tab_personnages_rouges) and tab_allie_ennemi == []:
        personnage = tab_personnages_rouges[i_personnage]
        self.jeu.changer_personnage(personnage)

        ennemi = self.ennemi_proche(personnage)

```

## 2 - Attaque

Le robot peut également choisir un personnage pour attaquer (si cela est possible). Il regarde chaque personnage de son équipe jusqu'à ce qu'il trouve un ennemi ou jusqu'à ce qu'il fasse le tour des personnages. Pour cela, il va appeler la méthode `ennemi_proche` qui renverra un ennemi proche du personnage en question.

```

### Pour une sorcière (différente potion) :
if ennemi != None and personnage.acc_personnage() == 'sorciere' :
    ### Joue intelligemment :
    if ennemi.acc_equipe() == 'rouge' and not self.attributs_jeu.acc_potions_rouges()[2].est_vide():
        self.attributs_jeu.mut_potion_rouge_selectionnee(2)
    elif ennemi.acc_personnage() == 'sorciere' and not self.attributs_jeu.acc_potions_rouges()[4].est_vide() :
        self.attributs_jeu.mut_potion_rouge_selectionnee(4)
    elif ennemi.acc_personnage() == 'monstre' :
        self.attributs_jeu.mut_potion_rouge_selectionnee(1)

    ### Sinon, utilise un potion aléatoire (sauf guérison) :
    else :
        tab = [1, 3, 4]
        numero_potion = random.choice(tab)
        while self.attributs_jeu.acc_potions_rouges()[numero_potion].est_vide() :
            tab.remove(numero_potion)
            numero_potion = random.choice(tab)
        self.jeu.attaque_sorciere_console(personnage.acc_equipe())
        self.attributs_jeu.mut_potion_rouge_selectionnee(numero_potion)

        self.jeu.ouverture_potion(ennemi.acc_x(), ennemi.acc_y()) #Jette sa potion
        tab_allie_ennemi = [personnage, None]

elif ennemi != None :
    tab_allie_ennemi = [personnage, ennemi]

i_personnage += 1

```

Tout d'abord, si le robot a trouvé un ennemi proche et que le personnage sélectionné est une sorcière, alors il va procéder de différentes manières selon l'ennemi. Si l'ennemi est de la même équipe (équipe rouge) et que la sorcière a encore une potion de soin, alors il sélectionne la potion de soins. Sinon si l'ennemi est une sorcière et que la sorcière (du robot) a encore une potion de

changement d'équipe, alors il sélectionne la potion de changement d'équipe. Sinon si l'ennemi est un monstre, alors il sélectionne la potion d'attaque, potion que la sorcière aura toujours dans sa réserve. Par contre, si aucune de ces conditions n'est vraie, alors le robot choisit une potion aléatoirement selon ce qui reste.

Après cela, le robot lance la procédure d'attaque de la sorcière avec la potion sélectionnée et on ajoute au tableau `tab_allie_ennemi`, qui contient donc le personnage avec lequel le robot attaque et l'ennemi, la sorcière et None (l'ennemi est déjà attaqué). Sinon, le personnage sélectionné n'est pas une sorcière et donc on ajoute tout simplement le personnage allié et l'ennemi dans le tableau.

```
def attaquer_ennemi(self, tab_personnages_rouges) :
    """
        Le robot attaque un ennemi proche d'un de ses personnages
    : param tab_personnages_rouges (list)
    : return (bool)
    """

    #Assertions
    assert isinstance(tab_personnages_rouges, list), "le tableau des personnages"
    #Code
    tab_allie_ennemi = self.choisir_personnage_attaquer(tab_personnages_rouges)

    if tab_allie_ennemi != []:
        allie = tab_allie_ennemi[0]
        ennemi = tab_allie_ennemi[1]

        if allie.acc_personnage() == 'geant':
            for case in self.attributs_jeu.acc_attaques():
                perso = self.terrain.acc_terrain(case[0], case[1])
                perso.est_attaque('geant')
                perso.mut_endommage()

        if ennemi != None and ennemi.acc_personnage() == 'geant':
            famille = self.jeu.famille_geant(ennemi)
            for geant in famille:
                geant.est_attaque(allie.acc_personnage())
                geant.mut_endommage()

        elif allie.acc_personnage() != 'sorciere':
            ennemi.est_attaque(allie.acc_personnage())
            ennemi.mut_endommage()

        self.attributs_jeu.mut_attaque_en_cours(True)
        self.attributs_jeu.mut_attaque_temps(0)
        #Si ce n'est pas une sorcière, affiche dans la console que le personnage
        if allie.acc_personnage() != "sorciere":
            self.jeu.attaque_console(allie, ennemi)
        self.attributs_jeu.mut_personnage_qui_attaque(True)
        return True

    return False
```

Voici la méthode qui permet d'attaquer une fois pour tout un ennemi (sauf avec le personnage sorcière). Elle prend donc le tableau de personnages de son

équipe en paramètre et renvoie True si le robot a attaqué ou a guéri un personnage/monstre et False sinon. La méthode commence par appeler la méthode précédente qui choisit le personnage avec lequel le robot va attaquer et l'ennemi qui va attaquer. Si le tableau contenant ces informations n'est pas vide, on vérifie les conditions suivantes. Par contre, si le robot n'a pas trouvé d'ennemi à attaquer ou un personnage à guérir, la méthode renvoie False.

Si le personnage sélectionné par le robot est un géant, tous les personnages présents autour du géant prendront des dégâts.

Si l'ennemi est un géant, toutes les parties de son corps prennent les mêmes dégâts que la partie sélectionnée comme ennemi par le robot. Sinon si le personnage sélectionné n'est pas une sorcière, alors le robot informe dans la console l'attaque du personnage sur l'ennemi. Avant de renvoyer True pour une attaque réussie, on informe au jeu qu'il y a une attaque en cours.

```
def chercher_coffre_proche(self) :
    ...
    Cherche un coffre proche d'un de ses personnages
    : return (None or module_objets.Coffre)
    ...

    coffre = None
    i_coffre = 0
    while i_coffre < len(self.attributs_jeu.acc_tab_coffres()) and coffre == None : #Tant qu'on à pas trouvé
        coffre_possible = self.attributs_jeu.acc_tab_coffres()[i_coffre]
        if not coffre_possible.acc_est_ouvert() :
            alentour = module_terrain.cases_alentour((coffre_possible.acc_x(), coffre_possible.acc_y()))

            i_case = 0
            while coffre == None and i_case < len(alentour) : #tant qu'on n'a pas tout regardé
                perso = self.terrain.acc_terrain(alentour[i_case][0], alentour[i_case][1])
                if isinstance(perso, module_personnage.Personnage) and perso.acc_equipe() == 'rouge' :
                    coffre = coffre_possible
                i_case += 1

    i_coffre += 1
    return coffre
```

La méthode cherche donc un coffre proche d'un de ses personnages. Pour ce faire, tant qu'il n'y a pas un coffre à côté d'un personnage rouge et qu'il n'a pas regardé tous les coffres de la partie, le robot prend un coffre possible, si ce coffre possible est fermé, il regarde pour chaque case autours s'il y a un personnage rouge. Si c'est le cas, il renvoie le coffre en question.

```

def ouvrir_coffre(self) :
    """
    Le robot ouvre un coffre s'il y en a un à côté d'un de ses personnages
    : return (bool)
    """

    coffre = self.chercher_coffre_proche()

    if coffre != None :
        coffre.ouverture()
        self.jeu.ouverture_coffre(coffre)
        self.jeu.coffre_console(coffre.acc_contenu())

        if not self.attributs_jeu.acc_annonce_coffre() :
            self.attributs_jeu.mut_annonce_coffre(True)
    return True
return False

```

Tout d'abord, le robot regarde s'il y a un coffre autour d'un de ses personnages. S'il y en a un, le robot ouvre ce coffre, annonce au jeu qu'il ouvre un coffre, annonce dans la console le contenu du coffre, si il n'y a pas d'annonce de coffre au préalable, le robot dit au jeu qu'il ouvre bel et bien un coffre, puis pour finir, renvoie True pour avoir fait une action et False si ce n'est pas le cas.

```

def jouer_robot(self) :
    """
    Fait jouer le robot quand c'est à son tour avec de pause de 5 secondes entre les actions
    : pas de return
    """

    #Si c'est à son tour :
    if self.attributs_jeu.acc_equipe_en_cours() == 'rouge' :
        #S'il n'y a pas eu une pause et qu'il n'y a pas de déplacement, d'attaque ou d'ouverture
        #d'un coffre en cours
        if self.temps_attente == None and not self.attributs_jeu.acc_deplacement_en_cours() and
           self.temps_attente = time.time()
        #Dès qu'une pause de plus de 3 secondes a été faite :
        elif self.temps_attente != None and time.time() - self.temps_attente > 1 :
            tab_personnages_rouges = self.acc_tab_personnages_rouges()
            #Si le robot n'a pas attaqué ou ouvert un coffre, il déplace un personnage de son équipe
            if not (self.attacter_ennemi(tab_personnages_rouges) or self.ouvrir_coffre()) :
                self.deplacer_personnage(tab_personnages_rouges)

            self.attributs_jeu.mut_nombre_action(self.attributs_jeu.acc_nombre_action() + 1)
            self.jeu.effacer_actions()

            self.temps_attente = None

```

Cette dernière méthode combine toutes les autres méthodes et permet d'avoir un robot opérationnel et fonctionnel.

Tout d'abord, il vérifie si c'est à son tour de jouer, puis il regarde s'il n'y a pas de temps d'attente, qu'il n'y a pas de déplacement, d'attaque ou d'ouverture de coffre en cours. Si ces conditions sont validées, il lance un temps d'attente qui lui permet de faire des pauses entre ses actions.

Après ce temps d'attente, la condition suivante vérifie si son temps d'attente est

de plus d'une seconde. Puis, il fonctionne par les étapes suivantes :

1. Crée un tableau qui regroupe ses personnages grâce à la méthode acc\_tab\_personnages\_rouges
2. Vérifie s'il peut attaquer un ennemi ou ouvrir un coffre. Ceci est la priorité.
3. S'il ne peut pas attaquer un ennemi ou ouvrir un coffre, déplace un de ses personnages intelligemment.
4. Augmente son nombre d'actions effectuées de 1 puis efface correctement les actions du personnage qui a été sélectionné.

Maintenant, le fonctionnement du robot n'a plus de secret pour vous !

## > MEDIEVAL HEROES !!

Et voilà ! Si vous êtes arrivé jusqu'ici, c'est que vous avez sûrement lu toute la documentation ! Bien joué jeune chevalier ! Désormais, vous avez vu les coulisses de notre jeu Medieval Heroes et vos parties ne pourront être que meilleures !

Bon jeu !