

pyReversi

par Guillaume DuriAUD ([Page personnelle](#))

Date de publication : 01/01/2008

Dernière mise à jour : 28/01/2008

- Développement du jeu **Reversi** (ou **Othello**) en **Python**
- Utilisation de la bibliothèque **wxPython**
- Algorithme **Alpha-Beta** pour la réflexion de l'ordinateur

- I - Prérequis
- II - Introduction
- III - Le jeu
- IV - Le code source
- V - Global.py
 - V.A - Les constantes
 - V.B - Les variables globales
- VI - Plate.py
 - VI.A - TestLeClic(Plate, r, joueur) et TestLeClic2(Plate, r, joueur)
 - VI.B - DoitPasser(Plate, joueur) et MovePossible(Plate, joueur)
 - VI.C - Reverse(Plate, r, joueur)
 - VI.D - Score(Plate)
 - VI.E - Calcul du score d'une position
 - VI.E.1 - Valeur d'une case du plateau de jeu.
 - VI.E.2 - TabJouable(Plate)
 - VI.E.3 - TabImprenable(Plate) et ExisteImprenable(Plate, r)
- VII - Player.py
 - VII.A - La classe Player
 - VII.B - La classe Computer
 - VII.B.1 - __init__(self, reversi, couleur)
 - VII.B.2 - Initialize(self)
 - VII.B.3 - MoveAleatory(self)
 - VII.B.4 - Play(self)
 - VII.B.5 - Waiting(self)
 - VII.B.6 - Reflexion(self, tab, profondeur, joueur, A, B, prof, passe, tempssomme, tempsdifference)
- VIII - Reversi.py - La classe Reversi
 - VIII.A - Initialize(self, black, white)
 - VIII.B - ChangePlayerMain(self, n = 0)
- IX - wxReversi.py
 - IX.A - La classe wxPlate
 - IX.B - La classe wxReversi
 - IX.B.1 - pnlPlateLDown(self, event)
 - IX.B.2 - pnlPlateMove(self, r, score)
- X - Conclusion
- XI - Téléchargement

I - Prérequis

Voici les langages et bibliothèques que j'ai utilisés pour développer pyReversi:

Système d'exploitation: Windows XP

Langage:  **Python 2.5.1**

Bibliothèque:  **wxPython 2.8.7.1**

Le jeu a été testé sous Linux (merci [hiko-seijuro](#)) et devrait également tourner sur Mac ainsi qu'avec d'autres versions de Python ou de wxPython mais je n'ai pas toutes les ressources matérielles pour faire les tests.

II - Introduction

Le jeu *Reversi* avait été le premier projet sur lequel j'avais travaillé pour apprendre le langage **Delphi** en 2003. En remettant un oeil dans les sources, je me suis aperçu qu'il était très difficile de se replonger dans le code. C'est pourquoi, j'ai voulu refaire une version en **Python** en profitant de mon expérience acquise depuis ces 4 années mais sans chercher à tout redévelopper. Ainsi, je me suis contenté de redévelopper le jeu avec le même algorithme (alpha-beta) pour la réflexion de l'ordinateur. Par contre, je n'ai pas réécrit l'apprentissage ni la création d'une liste de coups pour l'ouverture d'une partie. J'ai aussi omis quelques animations présentes dans la version Delphi. Si je trouve un peu de temps pour me replonger entièrement dans mon code Delphi et remettre un peu d'ordre, peut-être que je ferai une page pour présenter ce que j'avais codé.

L'interface graphique du jeu a été développée avec la bibliothèque **wxPython**. Vous pourrez ainsi vous servir de ce code comme exemple d'application manipulant cette bibliothèque.

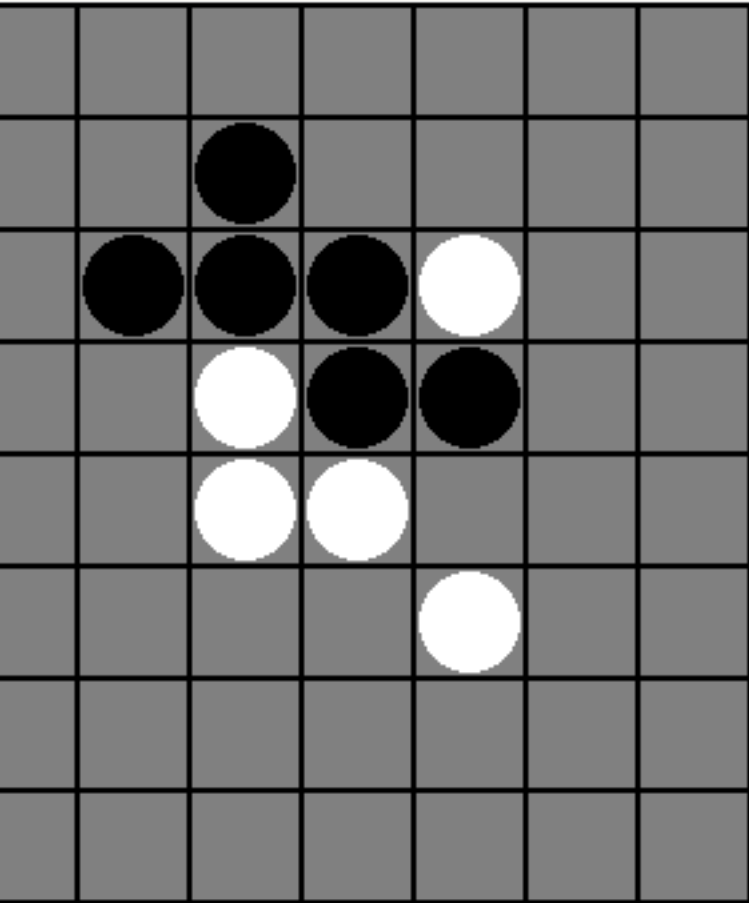
La réflexion de l'ordinateur est quant à elle beaucoup plus lente que celle de la version de Delphi (un facteur 100 quand même), alors qu'il peut jouer très rapidement avec une profondeur de réflexion de 6 coups voire 8 sur des machines récentes sur ma version Delphi, il faudra se contenter ici d'une profondeur de 4 coups (peut-être 6 avec une machine récente).

Dans tous les cas, ma version reste très très loin derrière  **WZebra** qui est capable d'aller jusqu'à une profondeur de 26 coups en un temps raisonnable, la version la plus complète à ma connaissance.

A partir de la version 1.0.0, j'ai commencé à optimiser légèrement la réflexion de l'ordinateur (un facteur 3 de récupérer avec la version 1.0.2).

si

u Options Aide



Infos


Noirs:
Temps: 00:00:00
Move : 133439

Blancs:
Temps: 00:00:34
Move : 0

11. B - C 6 2
10. W - F 3 0
9. B - D 7 2
8. W - D 6 0
7. B - F 5 1
6. W - F 6 0
5. B - E 6 -2

Nombre de coups : 8
Profondeur en cours : 5
Coup 3 : G5 contre : F4 évaluation
Coup 4 : F7 contre : E7 évaluation
Coup 5 : D8 contre : C7 évaluation
Coup 6 : G6 contre : C5 évaluation
Coup 7 : B6 contre : C7 évaluation
Coup 8 : B7 contre : C5 évaluation
Profondeur en cours : 6
Coup 1 : F4 contre : E3 évaluation
Coup 2 : E7 contre : F4 évaluation

III - Le jeu

Ce jeu est aussi connu sous le nom d'Othello. je vous renvoi sur le site de  **yahoo** pour une explication détaillée des règles.

IV - Le code source

Le code source est composé de 5 fichiers. Cette structure peut s'adapter à beaucoup de jeux possédant un plateau et se jouant à 2 joueurs (échec, puissance4, morpion, ...).

- **Global.py** : module contenant les constantes et variables globales du jeu
- **Plate.py** : module contenant la gestion interne du plateau de jeu (fonctions permettant le calcul du score d'une position)
- **Player.py** : module gérant les joueurs et la réflexion de l'ordinateur (présence de l'agorithme alpha-beta)
- **Reversi.py** : module principale gérant la création du jeu et le déroulement d'une partie
- **wxReversi.py** : module gérant l'interface graphique du jeu en wxPython et notamment le plateau du jeu

Nous allons étudier dans la suite chaque module un à un. Le code source est relativement court. Il doit contenir un millier de lignes dont presque la moitié sert à la gestion de l'interface graphique. En relisant le tout, il me semble relativement facile à comprendre. Je ne suis pas encore allé très loin niveau optimisation (contrairement à ce que j'avais pu faire en Delphi). J'ai toutefois essayé de détailler le plus possible chaque fonction de chaque classe pour que tout soit très clair et que vous puissiez si vous le souhaitez apporter des améliorations ou vous servir de ce code source pour aborder le développement d'autres jeux.

V - Global.py

Vous pouvez visualiser le contenu du fichier en cliquant sur le lien [Source Global.py](#)

V.A - Les constantes

Toutes mes constantes sont en majuscules. **INFINITY** représente la valeur maximum pour l'évaluation du score d'une position de jeu lors du déroulement de l'algorithme alpha-beta.

Les constantes commençant par **IA** définissent les fonctions pour le calcul du score. Ainsi, si vous souhaitez d'autres fonctions de calcul du score, il suffit de définir ici un identifiant puis d'écrire la fonction de calcul du score dans le module **Plate**.

Actuellement, j'en ai défini 6 dont 5 basiques et une faisant intervenir le calcul de quelques paramètres intéressants.

```
INFINITY = 1000
IAALEATORY = 0 ## "Coup aléatoires"
IABADSCORE = 1 ## "Pions other - Pions self"
IABESTSCORE = 2 ## "Pions self - Pions other"
IABESTSCOREMIN = 3 ## "min(1, Pions self - Pions other)"
IABESTSCOREMOVE = 4 ## "ScoreMove self - ScoreMove other"
IAWINLOSE = 5 ## "max(-1, min(1, Pions self - Pions other))"

IACHOICES = [u"Coup aléatoires",
              u"Score normal négatif",
              u"Score normal",
              u"Gagne ou meilleur perte",
              u"ScoreMove self - ScoreMove other",
              u"Gagne ou perd"]
```

Nous avons aussi 4 constantes pour le plateau de jeu ou les coups:

- **PLATEINIT** : plateau de jeu initial
- **PLATEMOVE** : cases du plateau de jeu où il peut y avoir un pion
- **FREEMOVEINIT** : case vide au départ d'une partie
- **PLATEMOVEINIT** : case prise par un pion au départ d'une partie

V.B - Les variables globales

Je définis souvent un dictionnaire global **glover** comment variable globale contenant toutes mes variables globales pour éviter d'avoir des conflits dans l'espace des noms quand on redéfinit une variable en ayant oublié de la déclarer **global**.

Ici, toutes les valeurs concernent les paramètres initiaux de la réflexion des ordinateurs noirs et blancs à savoir

- La durée des phases du début et fin de partie.
- La profondeur de réflexion des coups en début, milieu et fin de partie.
- L'IA sélectionnée pour chacune des phases.

Les deux dernières valeurs **freemove** représentent pour une partie, les cases libres en cours et **platemove**, les cases utilisées.

```
glovar = {}
glovar['levelbeginblack'] = 6
glovar['levelbeginwhite'] = 6
glovar['levelmiddleblack'] = 4
glovar['levelmiddlewhite'] = 4
glovar['levelendblack'] = 8
glovar['levelendwhite'] = 8
glovar['thinkingblack'] = True
glovar['thinkingwhite'] = True
glovar['iabeginblack'] = IABESTSCOREMOVE
glovar['iabeginwhite'] = IABESTSCOREMOVE
glovar['nbmovebeginblack'] = 6
glovar['nbmovebeginwhite'] = 6
glovar['iaendblack'] = IABESTSCOREMIN
glovar['iaendwhite'] = IABESTSCOREMIN
glovar['nbmoveendblack'] = 8
glovar['nbmoveendwhite'] = 8
glovar['iamiddleblack'] = IABESTSCOREMOVE
glovar['iamiddlewhite'] = IABESTSCOREMOVE

glovar['freemove'] = range(11, 89)
glovar['platemove'] = []
```

VI - Plate.py

Vous pouvez visualiser le contenu du fichier en cliquant sur le lien [Source Plate.py](#)

Le fichier **Plate.py** gère le plateau de jeu à travers des fonctions prenant toujours en premier paramètre une liste représentant un plateau de jeu. Initialement, j'avais construit ce module à travers une classe **Plate** mais je me suis aperçu finalement que l'IA, qui devait effectuer de nombreuses copies en particulier d'objets de cette classe, était grandement accélérée en utilisant plutôt directement des fonctions et juste une liste pour le plateau de jeu. J'ai finalement opté pour la suppression de la classe pour ne conserver que des fonctions qui s'appliquent sur un objet **list** que l'on doit fournir en paramètre à chaque appel. Au final, on y gagne tout de même un facteur 2 au niveau de la réflexion de l'ordinateur. Le plateau de jeu de 8x8 cases est géré à travers une liste de 100 éléments en ayant ajouté au préalable une bordure pour éviter que certains tests se fassent en dehors du plateau de jeu.

Ainsi, la case du plateau en haut à gauche est la 12ème case de la liste (numéro 11), quand on se déplace sur le plateau d'une case à droite, on avance d'une case dans la liste et quand on se déplace sur le plateau d'une case vers le bas, on avance de 10 cases dans la liste.

Méthodes principales applicables sur une liste représentant le plateau de jeu:

- **TestLeClic(Plate, r, joueur)** : teste si le coup **r** par le joueur **joueur** est acceptable (en vérifiant que la case **r** est vide)
- **TestLeClic2(Plate, r, joueur)** : teste si le coup **r** par le joueur **joueur** est acceptable (sans vérifier que la case **r** est vide)
- **DoitPasser(Plate, joueur)** : indique si le joueur ne peut pas jouer à son tour de jeu
- **MovePossible(Plate, joueur)** : donne la liste des coups jouables
- **Reverse(Plate, r, joueur)** : effectue le coup **r** par le joueur **joueur**
- **Score(Plate)** : retourne le score (au niveau nombre de pions) de la partie en cours
- **ScoreMove(Plate)** : retourne le score d'une position de jeu à partir du calcul de différents paramètres

Pour construire le plateau de jeu, on l'initialise avec la constante **PLATEINIT** ou bien on fait une copie du plateau précédent **plate[:]**. La liste est représentée avec les chiffres 0, 1, 2 et 3:

- 0 si la case est vide
- 1 si la case est prise par un pion noir
- 2 si la case est prise par un pion blanc
- 3 si la case est en dehors du plateau de jeu

Initialement, seuls les 4 cases du centre ont un pion dessus.

VI.A - TestLeClic(Plate, r, joueur) et TestLeClic2(Plate, r, joueur)

Cette fonction indique si le coup **r** (**r** étant compris entre 11 et 88) joué par le joueur **joueur** est acceptable. Pour cela, il suffit de vérifier que le coup permet au joueur **joueur** de retourner au moins un pion adverse en testant donc toutes les directions.

Dans le cas de **TestLeClic**, on vérifie avant tout que la case est vide, ce qu'on ne fait pas dans **TestLeClic2** (ce qui peut nous permettre de gagner un peu de temps lors de la réflexion de l'ordinateur).

```
def TestLeClic(Plate, r, joueur):
    if Plate[r] == 0:
        jou = 3 - joueur
        k = r-1
        if Plate[k] == jou:
            k -= 1
            while Plate[k] == jou: k -= 1
            if Plate[k] == joueur: return True

        k = r+1
        if Plate[k] == jou:
            k += 1
            while Plate[k] == jou: k += 1
            if Plate[k] == joueur: return True

        k = r-10
        if Plate[k] == jou:
            k -= 10
            while Plate[k] == jou: k -= 10
            if Plate[k] == joueur: return True

        k = r+10
        if Plate[k] == jou:
            k += 10
            while Plate[k] == jou: k += 10
            if Plate[k] == joueur: return True

        k = r-11
        if Plate[k] == jou:
            k -= 11
            while Plate[k] == jou: k -= 11
            if Plate[k] == joueur: return True

        k = r+11
        if Plate[k] == jou:
            k += 11
            while Plate[k] == jou: k += 11
            if Plate[k] == joueur: return True

        k = r - 9
        if Plate[k] == jou:
            k -= 9
            while Plate[k] == jou: k -= 9
            if Plate[k] == joueur: return True

        k = r+9
        if Plate[k] == jou:
            k += 9
            while Plate[k] == jou: k += 9
            if Plate[k] == joueur: return True

    return False
```

VI.B - DoitPasser(Plate, joueur) et MovePossible(Plate, joueur)

La fonction **DoitPasser(Plate, joueur)** vérifie si le joueur **joueur** peut jouer, donc s'il existe au moins une case qui permette au joueur de retourner des pions adverses.

la fonction **MovePossible(Plate, joueur)** retourne la liste de tous les coups possibles pour le joueur **joueur**

```
def DoitPasser(Plate, joueur):
    for r in glovar['freemove']:
        if TestLeClic(Plate, r, joueur): return False
    return True

def MovePossible(Plate, joueur):
    return [r for r in glovar['freemove'] if TestLeClic(Plate, r, joueur)]
```

VI.C - Reverse(Plate, r, joueur)

Cette fonction pose un pion de la couleur du joueur **joueur** sur la case **r** et retourne les autres pions en conséquences. On teste ainsi, dans les 8 directions, s'il est possible de retourner ou non des pions adverses.

```
def Reverse(Plate, r, joueur):
    jou = 3 - joueur
    Plate[r] = joueur
    k = r-1
    while Plate[k] == jou: k -= 1
    if Plate[k] == joueur:
        k += 1
        while Plate[k] == jou:
            Plate[k] = joueur
            k += 1

    k = r+1
    while Plate[k] == jou: k += 1
    if Plate[k] == joueur:
        k -= 1
        while Plate[k] == jou:
            Plate[k] = joueur
            k -= 1

    k = r-10
    while Plate[k] == jou: k -= 10
    if Plate[k] == joueur:
        k += 10
        while Plate[k] == jou:
            Plate[k] = joueur
            k += 10

    k = r+10
    while Plate[k] == jou: k += 10
    if Plate[k] == joueur:
        k -= 10
        while Plate[k] == jou:
            Plate[k] = joueur
            k -= 10

    k = r-11
    while Plate[k] == jou: k -= 11
    if Plate[k] == joueur:
        k += 11
        while Plate[k] == jou:
            Plate[k] = joueur
            k += 11

    k = r+11
    while Plate[k] == jou: k += 11
    if Plate[k] == joueur:
        k -= 11
```

```
        while Plate[k] == jou:
            Plate[k] = joueur
            k -= 11

    k = r-9
    while Plate[k] == jou: k -= 9
    if Plate[k] == joueur:
        k += 9
        while Plate[k] == jou:
            Plate[k] = joueur
            k += 9

    k = r+9
    while Plate[k] == jou: k += 9
    if Plate[k] == joueur:
        k -= 9
        while Plate[k] == jou:
            Plate[k] = joueur
            k -= 9
```

VI.D - Score(Plate)

Cette fonction retourne le nombre de pions de chaque joueur et permet donc d'avoir le score actuel de la partie.

```
def Score(Plate):
    return Plate.count(1), Plate.count(2)
```

VI.E - Calcul du score d'une position

La réflexion de l'ordinateur est développée par l'algorithme alpha-beta. Celui-ci repose sur le calcul de la valeur (ou score) de la position étudiée du plateau. Le calcul du score d'une position du plateau se fait soit par la fonction **Score(Plate)** qui retourne basiquement le nombre de pions de chaque joueur soit par la méthode **ScoreMove(Plate)** qui calcule différents paramètres du jeu et retourne un tuple représentant le score de chaque joueur.

Cette dernière fonction va se servir de plusieurs paramètres:

- Le nombre de pions de chaque joueur
- La valeur des cases sur lesquelles reposent les pions du plateau
- Le nombre de coups jouables par chaque joueur
- Le nombre de cases imprenables de chaque joueur

Cette fonction étant très souvent appelée dans la réflexion de l'ordinateur (à chaque feuille de l'arbre d'une série de coups), il est important que celle-ci soit la plus optimisée possible. Il est important de calculer un maximum de paramètres pour obtenir le score le plus réaliste pour la position étudiée du plateau sans pour autant trop en faire pour ne pas perdre trop de temps. Il peut ainsi être plus intéressant de pouvoir augmenter la profondeur de la réflexion en contrepartie d'une fonction du calcul du score d'une position moins ambitieuse.

Je me suis contenté des 4 éléments précédent auxquels j'affecte un coefficient multiplicatif (arbitraire choisi après quelques tests expérimentaux).

Le score de chaque joueur est enregistré dans les variables locales **BScore** et **WScore**.

L'algorithme est le suivant:

- On commence par compter le nombre de pions de chaque joueur (**BScore**, **WScore** = **Score(Plate)**)
- Si l'un des 2 joueurs n'a plus de pion sur le plateau, le joueur adverse gagne et on retourne directement **+INFINITY** et **0** pour le perdant.
- Puis on calcule les 3 derniers paramètres que l'on ajoute au score de chaque joueur.

```
def ScoreMove(Plate):
    BScore, WScore = Score(Plate)
    if BScore == 0 or WScore == 0: return (WScore == 0) * INFINITY, (BScore == 0) * INFINITY
    BScore *= -1
    WScore *= -1
    jou1, jou2 = TabJouable(Plate)
    BScore += 2 * jou1
    WScore += 2 * jou2
    for r in PLATEMOVE:
        if Plate[r] == 1: BScore += TABVALUE[r]
        elif Plate[r] == 2: WScore += TABVALUE[r]
    imp1, imp2 = TabImprenable(Plate)
    BScore += 3 * imp1
    WScore += 3 * imp2
    return BScore, WScore
```

Vous pourrez ainsi créer d'autres fonctions de calcul du score d'une position en changeant soit les constantes multiplicatives soit en calculant d'autres paramètres et confronter les différentes fonctions en faisant jouer 2 ordinateurs.

VI.E.1 - Valeur d'une case du plateau de jeu.

Chaque case possède une valeur qui détermine si il est intéressant ou non d'avoir un pion dessus. Ces valeurs sont définies dans la liste constante **TABVALUE**. Ainsi, les quatre coins sont bien cotés (+20):

```
TABVALUE = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 20, -3, -1, -1, -1, -1, -3, 20, 0,
            0, -3, -6, -2, -2, -2, -2, -6, -3, 0,
            0, -1, -2, 0, 0, 0, 0, -2, -1, 0,
            0, -1, -2, 0, 0, 0, 0, -2, -1, 0,
            0, -1, -2, 0, 0, 0, 0, -2, -1, 0,
            0, -3, -6, -2, -2, -2, -2, -6, -3, 0,
            0, 20, -3, -1, -1, -1, -1, -3, 20, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

J'ai choisi une constante, mais on aurait très bien pu prendre des valeurs variant en fonction de l'avancée de la partie.

VI.E.2 - TabJouable(Plate)

TabJouable(Plate) retourne pour chaque joueur, le nombre de coups jouables (s'il jouait au prochain tour, mais cette valeur n'est en fait effective que pour le joueur qui aurait la main).

```
def TabJouable(Plate):
    j1 = 0
    j2 = 0
    for r in glovar['freemove']:
```

```
    if Plate[r] == 0:
        j1 += TestLeClic2(Plate, r, 1)
        j2 += TestLeClic2(Plate, r, 2)
    return j1, j2
```

VI.E.3 - TabImprenable(Plate) et ExisteImprenable(Plate, r)

Pour chaque case du plateau, si un pion est dessus, on teste si celui-ci est entièrement protégé, c'est à dire qu'il est entouré dans les 8 directions par un pion adverse ou un bord.

Ce calcul peut ne pas être pertinent et peut-être qu'on peut sans passer pour gagner du temps dans le calcul du score (notamment en début de partie).

```
def TabImprenable(Plate):
    j1 = 0
    j2 = 0
    for r in PLATEMOVE:
        if Plate[r] == 1: j1 += IsImprenable(Plate, r)
        elif Plate[r] == 2: j2 += IsImprenable(Plate, r)
    return j1, j2

def IsImprenable(Plate, r):
    jou = Plate[r]
    other = 3-jou

    k = r - 10
    while Plate[k] == jou: k -= 10
    if Plate[k] == 0 or Plate[k] == other:
        sidel = Plate[k]
        k = r + 10
        while Plate[k] == jou: k += 10
        if Plate[k] == 0 or (Plate[k] == other and sidel == 0): return False

    k = r - 1
    while Plate[k] == jou: k -= 1
    if Plate[k] == 0 or Plate[k] == other:
        sidel = Plate[k]
        k = r + 1
        while Plate[k] == jou: k += 1
        if Plate[k] == 0 or (Plate[k] == other and sidel == 0): return False

    k = r - 11
    while Plate[k] == jou: k -= 11
    if Plate[k] == 0 or Plate[k] == other:
        sidel = Plate[k]
        k = r + 11
        while Plate[k] == jou: k += 11
        if Plate[k] == 0 or (Plate[k] == other and sidel == 0): return False

    k = r - 9
    while Plate[k] == jou: k -= 9
    if Plate[k] == 0 or Plate[k] == other:
        sidel = Plate[k]
        k = r + 9
        while Plate[k] == jou: k += 9
        if Plate[k] == 0 or (Plate[k] == other and sidel == 0): return False

    return True
```

VII - Player.py

Vous pouvez visualiser le contenu du fichier en cliquant sur le lien [Source Player.py](#)

VII.A - La classe Player

La classe **Player** identifie un joueur humain. Les paramètres donnés sont en général uniquement présents pour assurer une compatibilité avec la classe **Computer**. La classe **Player** n'a de toute façon qu'un constructeur qui initialise quelques paramètres. Mais nous nous attarderons plutôt sur la classe **Computer**.

```
class Player:
    def __init__(self, reversi, couleur):
        self.reversi = reversi
        self.couleur = couleur
        if couleur == 'white': self.joueur = 2
        elif couleur == 'black': self.joueur = 1
        self.isWaiting = False
        self.isEndGame = False
        self.timer = 0
        self.compteur = 0
```

VII.B - La classe Computer

L'intérêt de programmer ce jeu réside surtout dans le fait de pouvoir jouer contre un ordinateur. La classe **Computer** permet de gérer les choix de jeu de l'ordinateur. La difficulté est de ne pas se perdre dans les nombreux attributs. Les attributs de la classe **Computer**:

- **reversi**: pointeur sur une instance de **Reversi.Reversi** pour avoir un accès complet au jeu
- **couleur**: **black** or **white**
- **joueur**: **1** (noirs) ou **2** (blancs)
- **isWaiting**: indique si l'ordinateur est dans la fonction **Waiting(self)**
- **levelbegin**: profondeur des coups de la réflexion de l'ordinateur en début de partie
- **levelmiddle**: profondeur des coups de la réflexion de l'ordinateur en milieu de partie
- **levelend**: profondeur des coups de la réflexion de l'ordinateur en fin de partie
- **level**: profondeur actuelle des coups de la réflexion de l'ordinateur
- **BestMove**: meilleurs coups à jouer donnés par la fonction **Réflexion**
- **timer**: temps total de réflexion d'un joueur
- **IAbegin**: type d'IA en début de partie
- **NbMoveIAbegin**: durée (en coups) du début de partie
- **IAmiddle**: type d'IA en milieu de partie
- **IAend**: type d'IA en fin de partie
- **NbMoveIAend**: durée (en coups de la fin de partie)
- **IA**: IA actuelle
- **ThinkingDuringOther**: booléen indiquant si l'ordinateur réfléchit pendant la réflexion de l'adversaire.
- **endreflexion**: Force l'arrêt de la réflexion pour que l'ordinateur puisse jouer immédiatement.
- **endgame**: Force l'arrêt du jeu pour l'ordinateur

- **movenext** : Liste des prochains coups possibles du joueur ayant la main
- **moveagainst** : Liste des coups de contre aux coups possibles (-1 indiquant que l'ordinateur n'a pas encore choisi le meilleur coup possible)
- **scoreagainst** : Liste des scores correspondants à chaque coup de contre
- **compteur** : Compteur du nombre d'appel de la fonction **Reflexion**

Les méthodes de la classe **Computer**:

- **__init__(self, reversi, couleur)** : constructeur
- **Initialize(self)** : initialise les paramètres nécessaires à chaque tour de jeu
- **MoveAleatory(self)** : retourne un des coups possibles aléatoirement
- **Play(self)** : lance la réflexion de l'ordinateur pour effectuer son tour de jeu
- **Waiting(self)** : lance la réflexion de l'ordinateur pendant le tour de jeu de l'adversaire
- **Reflexion(self, tab, profondeur, joueur, A, B, passe, tempssomme, tempsdifference)** : Etude des coups par l'algorithme alpha-beta

VII.B.1 - __init__(self, reversi, couleur)

Le constructeur **__init__(self, reversi, couleur)** crée une instance de **Computer** en initialisant ces différents attributs par les variables globales du fichier **Global.py**.

```
def __init__(self, reversi, couleur):
    self.reversi = reversi
    self.couleur = couleur
    if couleur == 'white': self.joueur = 2
    elif couleur == 'black': self.joueur = 1
    self.isWaiting = False
    self.levelbegin = glovar['levelbegin'+couleur]
    self.levelmiddle = glovar['levelmiddle'+couleur]
    self.level = self.levelbegin
    self.levelend = glovar['levelend'+couleur]
    self.BestMove = -1
    self.timer = 0
    self.IAbegin = glovar['iabegin' + couleur]
    self.NbMoveIAbegin = glovar['nbmovebegin' + couleur]
    self.IAmiddle = glovar['iamiddle' + couleur]
    self.IAend = glovar['iaend' + couleur]
    self.NbMoveIAend = glovar['nbmoveend' + couleur]
    self.ThinkingDuringOther = glovar['thinking'+couleur]
    self.compteur = 0
    self.Initialize()
```

VII.B.2 - Initialize(self)

La fonction **Initialize(self)** est appelée à chaque tour pour réinitialiser les différents paramètres présents dans cette fonction.

```
def Initialize(self):
    self.endreflexion = False
    self.endgame = False
    self.isWaiting = False
    self.isEndGame = False
    self.movenext = MovePossible(self.reversi.plate, 3 - self.joueur)
```

```

self.moveagainst = len(self.movenext) * [-1]
self.scoreagainst = len(self.movenext) * [0]

if self.NbMoveIAbegin + 4 < sum(Score(self.reversi.plate)):
    if 64 - sum(Score(self.reversi.plate)) <= self.NbMoveIAend:
        self.level = self.levelend
        self.IA = self.IAend
    else:
        self.level = self.levelmiddle
        self.IA = self.IAmiddle
else:
    self.level = self.levelbegin
    self.IA = self.IAbegin

```

VII.B.3 - MoveAleatory(self)

Cette fonction retourne un des coups possibles de l'ordinateur aléatoirement. On réutilise ici la fonction **MovePossible(Plate, joueur)** du fichier **Plate.py**

```

def MoveAleatory(self):
    return random.choice(MovePossible(self.reversi.plate, self.joueur))

```

VII.B.4 - Play(self)

Cette fonction calcule le coup que va jouer l'ordinateur. Si ce dernier a eu le temps de réfléchir pendant le temps de réflexion de son adversaire (un coup de contre doit être différent de -1 pour le dernier coup joué (**self.moveagainst[self.movenext.index(self.reversi.lastMove)] != -1**), l'ordinateur joue immédiatement ce coup de contre. Sinon, en fonction de son IA, il calcule son coup de contre par l'algorithme alpha-beta à travers la fonction **Reflexion**. Le coup est finalement joué en appelant la méthode **pnlPlateMove** de l'instance **wReversi** (qui se chargera alors de changer le tour de jeu).

```

def Play(self):
    print self.movenext
    print self.moveagainst
    if max(self.moveagainst) == -1 or
self.moveagainst[self.movenext.index(self.reversi.lastMove)] == -1:
        if self.IA == IAALATORY:
            move = self.MoveAleatory()
            score = 0
        else:
            self.BestMove = -1
            if self.IA == IABESTSCOREMIN: score = self.Reflexion(self.reversi.plate, 0,
self.joueur, -INFINITY, 1, False, 0, 100)
            elif self.IA == IAWINLOSE: score = self.Reflexion(self.reversi.plate, 0,
self.joueur, -2, 1, False, 0, 100)
            else: score = self.Reflexion(self.reversi.plate, 0, self.joueur, -INFINITY,
INFINITY, False, 0, 100)
            move = self.BestMove
        else:
            move = self.moveagainst[self.movenext.index(self.reversi.lastMove)]
            score = self.scoreagainst[self.movenext.index(self.reversi.lastMove)]

    if self.endgame: return
    self.movenext = [-1]
    self.moveagainst = [-1]
    wx.CallAfter(self.reversi.wReversi.pnlPlateMove, move, score)

```

VII.B.5 - Waiting(self)

La fonction **Waiting(self)** permet à l'ordinateur de réfléchir pendant le tour de jeu de son adversaire (ce qui peut être utile pour gagner du temps si l'adversaire est un humain). Pour chaque coup possible (**movenext**), l'ordinateur va calculer le coup de contre en remplissant au fur et à mesure la liste **moveagainst** et la liste des scores correspondant **scoreagainst**. Chaque coup est calculé avec l'IA qui lui a été définie. Une fois que tous les coups ont été contrés, il recommence l'opération mais en augmentant la profondeur de sa réflexion d'une unité. Cependant, on trie les coups dans l'ordre de la plus petite évaluation pour que l'ordinateur se concentre ensuite d'abord sur le meilleur coup pour son adversaire.

Ainsi plus l'adversaire met de temps à jouer, plus l'ordinateur a le temps de préparer un bon coup. Dès que l'adversaire a joué (**endreflexion = True**), l'ordinateur stoppe sa réflexion.

Comme les fonctions **Waiting(self)** et **Play(self)** sont exécutées à partir d'un thread, il est important que toutes les commandes **wxPython** ne soient pas envoyés directement sous peine de risque de plantage de l'application. Ainsi, on utilise la fonction sécurisée **wx.CallAfter(mafonction, *mesparamètres)**.

```
def Waiting(self):
    try:
        self.isWaiting = True
        if len(self.movenext) == 0 or (not self.ThinkingDuringOther): return
        wx.CallAfter(self.reversi.wReversi.lThinking.Clear)
        wx.CallAfter(self.reversi.wReversi.lThinking.Append, 'Nombre de coups : ' +
str(len(self.movenext)))
        bolend = False
        random.shuffle(self.movenext)
        if self.IA == IAALATORY: return
        else:
            plate = self.reversi.plate[:]
            levelbackup = self.level
            while True:
                wx.CallAfter(self.reversi.wReversi.lThinking.Append, 'Profondeur en cours : ' +
str(self.level))
                for i in range(len(self.movenext)):
                    letab = plate[:]
                    Reverse(letab, self.movenext[i], 3 - self.joueur)
                    self.BestMove = -1
                    if self.IA == IABESTSCOREMIN: mn = self.Reflexion(letab, 0, self.joueur,
-INFINITY, 1, False, 0, 100)
                    elif self.IA == IAWINLOSE: mn = self.Reflexion(letab, 0, self.joueur, -2,
1, False, 0, 100)
                    else: mn = self.Reflexion(letab, 0, self.joueur, -INFINITY, INFINITY,
False, 0, 100)
                    if self.endreflexion or (not self.ThinkingDuringOther): return
                    self.moveagainst[i] = self.BestMove
                    self.scoreagainst[i] = mn
                    bm = self.BestMove
                    mni = self.movenext[i]
                    if levelbackup != self.level:
                        try: wx.CallAfter(self.reversi.wReversi.lThinking.Delete, 2)
                        except: pass
                    wx.CallAfter(self.reversi.wReversi.lThinking.Append, u"Coup %d : %s contre
: %s évaluation : %d" %
                                (i+1, chr(mni % 10 + 64) + str(9 - mni // 10), chr(bm % 10 +
64) + str(9 - bm // 10), mn))
                    wx.CallAfter(self.reversi.wReversi.lThinking.Clear)
                    wx.CallAfter(self.reversi.wReversi.lThinking.Append, 'Nombre de coups : ' +
str(len(self.movenext)))
```

```

        wx.CallAfter(self.reversi.wReversi.lThinking.Append, 'Profondeur en cours : ' +
str(self.level))
        a = sorted(zip(self.scoreagainst, self.moveagainst, self.movenext))
        self.moveagainst = [i[1] for i in a]
        self.movenext = [i[2] for i in a]
        self.scoreagainst = [i[0] for i in a]
        for i in range(len(self.movenext)):
            wx.CallAfter(self.reversi.wReversi.lThinking.Append, u"Coup %d : %s contre
: %s évaluation : %d" %
                (i+1, chr(self.movenext[i] % 10 + 64) + str(9 -
self.movenext[i] // 10),
                chr(self.moveagainst[i] % 10 + 64) + str(9 -
self.moveagainst[i] // 10), self.scoreagainst[i]))
            if 64 - (sum(Score(plate)) + self.level) <= 0:
                if bolend: return
                else:
                    self.IA = self.IAend
                    bolend = True
            else: self.level += 1

        finally:
            self.level = levelbackup
            self.isWaiting = False
    
```

VII.B.6 - Reflexion(self, tab, profondeur, joueur, A, B, prof, passe, tempssomme, tempsdifference)

Cette fonction a été l'une des plus intéressantes à développer. Tout repose sur l'algorithme **alpha-beta**. Cet algorithme est une optimisation de l'algorithme **MinMax** qui évite d'étudier des branches inutiles.

Le fonction de l'algorithme **alpha-beta** pour le jeu Reversi est le suivant:

```

Nous avons besoin de 2 constantes:
    joueurprincipal : joueur ayant initié l'algorithme
    profondeurmax : hauteur de l'arbre des coups

fonction AlphaBeta(plateau, profondeur, joueur, A, B):
    si profondeur == profondeurmax: // on est sur une feuille de l'arbre
        on retourne le score (par rapport au joueur principal)
    si joueur != joueurprincipal:
        si joueur ne peut pas jouer:
            on retourne AlphaBeta(plateau, profondeur, joueurprincipal, A, B)
        beta = B
        pour chaque coup possible de joueur:
            nouveauplateau = plateau + coup
            score = AlphaBeta(nouveauplateau, profondeur+1, joueurprincipal, A, beta)
            beta = min(score, beta)
            if A >= beta: on retourne beta
        on retourne beta
    sinon:
        si joueurprincipal ne peut pas jouer:
            on retourne AlphaBeta(plateau, profondeur, joueur, A, B)
        alpha = A
        pour chaque coup possible de joueurprincipal:
            nouveauplateau = plateau + coup
            score = AlphaBeta(nouveauplateau, profondeur+1, joueur, alpha, B)
            alpha = max(score, alpha)
            if alpha >= B: on retourne alpha
        on retourne alpha
    
```

Il est aussi important de vérifier que le jeu ne s'arrête pas (c'est à dire qu'il n'y a plus de case libre sur le plateau ou que plus aucun joueur ne peut jouer. C'est l'utilité de la variables **passee**.

Les variables **tempssomme** et **tempdifference** permettent de savoir où l'ordinateur en est dans sa réflexion.

tempdifference est égal au temps précédent divisé par le nombre de coups possibles. Et pour chaque coup d'un même niveau, on ajoute **tempdifference** à **tempssomme**.

Par exemple, au départ, on lance la réflexion avec **tempssomme = 0** et **tempdifference = 100**.

Supposons qu'au premier niveau, il y ait 4 coups possibles. Pour le premier coup à traiter, on lancera la réflexion avec **tempssomme = 0** et **tempdifference = 100 / 4 = 25**. Pour le deuxième coup, on lancera la réflexion avec **tempssomme = 0 + 25 = 25** et **tempdifference = 100 / 4 = 25** et ainsi de suite récursivement.

```
def Reflexion(self, tab, profondeur, joueur, A, B, passe, tempssomme, tempdifference):
    self.compteur += 1

    if self.endreflexion and (self.BestMove >= 0 or profondeur != 0):
        return -INFINITY + 1

    joueurprinc = self.joueur

    if profondeur == self.level:
        if self.IA == IABADSCORE:
            s = Score(tab)
            if s[joueurprinc-1] == 0: return -INFINITY + 1
            elif s[2-joueurprinc] == 0: return INFINITY - 1
            else: return s[2 - joueurprinc] - s[joueurprinc-1]
        elif self.IA == IABESTSCOREMOVE:
            s = ScoreMove(tab)
            return s[joueurprinc-1] - s[2-joueurprinc]
        elif self.IA == IAWINLOSE:
            s = Score(tab)
            return max(-1, s[joueurprinc-1] - s[2 - joueurprinc])
        else:
            s = Score(tab)
            if s[joueurprinc-1] == 0: return -INFINITY + 1
            elif s[2-joueurprinc] == 0: return INFINITY - 1
            return s[joueurprinc-1] - s[2 - joueurprinc]
    else:

        lMove = MovePossible(tab, joueur)
        lenlMove = len(lMove)

        if joueurprinc != joueur:
            if lenlMove == 0:
                if passe:
                    if self.IA == IABADSCORE:
                        s = Score(tab)
                        if s[joueurprinc-1] == 0: return -INFINITY + 1
                        elif s[2-joueurprinc] == 0: return INFINITY - 1
                        else: return s[2 - joueurprinc] - s[joueurprinc-1]

                    elif self.IA == IABESTSCOREMOVE:
                        s = ScoreMove(tab)
                        return s[joueurprinc-1] - s[2-joueurprinc]
                    elif self.IA == IAWINLOSE:
                        s = Score(tab)
                        return max(-1, s[joueurprinc-1] - s[2 - joueurprinc])
                    else:
                        s = Score(tab)
```

```

        if s[joueurprinc-1] == 0: return -INFINITY + 1
        elif s[2-joueurprinc] == 0: return INFINITY - 1

        return s[joueurprinc-1] - s[2 - joueurprinc]
    letab = tab[:]
    return self.Reflexion(letab, profondeur, joueurprinc, A, B, True, tempssomme,
tempsdifference)
    letempsdif = tempsdifference / lenlMove
    letempssom = tempssomme
    beta = B
    for coup in lMove:
        letab = tab[:]
        Reverse(letab, coup, joueur)
        res = self.Reflexion(letab, profondeur+1, joueurprinc, A, beta, False,
letempssom, letempsdif)
        letempssomancien = letempssom
        letempssom = letempssom+ letempsdif
        if (int(letempssomancien) != int(letempssom)):
            if joueurprinc == 1: wx.CallAfter(self.reversi.wReversi.gBlack.SetValue,
letempssom)

            else: wx.CallAfter(self.reversi.wReversi.gWhite.SetValue, letempssom)
        ##beta = min(beta, res)
        ##if A >= beta: return beta
        if res < beta:
            beta = res
            if A >= beta: return beta
    return beta
else:
    troisjou = 3 - joueur
    if lenlMove == 0:
        if passe:
            if self.IA == IABADSCORE:
                s = Score(tab)
                if s[joueurprinc-1] == 0: return -INFINITY + 1
                elif s[2-joueurprinc] == 0: return INFINITY - 1

                if s[joueurprinc-1] == 0: return -INFINITY + 1
                else: return s[2 - joueurprinc] - s[joueurprinc-1]

            elif self.IA == IABESTSCOREMOVE:
                s = ScoreMove(tab)
                return s[joueurprinc-1] - s[2-joueurprinc]
            elif self.IA == IAWINLOSE:
                s = Score(tab)
                return max(-1, s[joueurprinc-1] - s[2 - joueurprinc])
            else:
                s = Score(tab)
                if s[joueurprinc-1] == 0: return -INFINITY + 1
                elif s[2-joueurprinc] == 0: return INFINITY - 1

                return s[joueurprinc-1] - s[2 - joueurprinc]
        letab = tab[:]
        return self.Reflexion(letab, profondeur, troisjou, A, B, True, tempssomme,
tempsdifference)
    letempsdif = tempsdifference / lenlMove
    letempssom = tempssomme
    alpha = A
    for coup in range(lenlMove):
        letab = tab[:]
        Reverse(letab, lMove[coup], joueur)
        ## Pour le dernier coup, on peut se contenter de B = alpha+1
        if profondeur == 0 and coup == lenlMove-1:
            res = self.Reflexion(letab, profondeur+1, troisjou, alpha, alpha+1, False,
letempssom, letempsdif)
        else: res = self.Reflexion(letab, profondeur+1, troisjou, alpha, B, False,
letempssom, letempsdif)

```

```
        letempssomancien = letempssom
        letempssom = letempssom+letempsdif
        if (int(letempssomancien) != int(letempssom)):
            if joueurprinc == 1: wx.CallAfter(self.reversi.wReversi.gBlack.SetValue,
letempssom)
                else: wx.CallAfter(self.reversi.wReversi.gWhite.SetValue, letempssom)
        if res > alpha:
            if profondeur == 0:
                ##if self.BestMove != -1:
                ##    pass
                self.BestMove = lMove[coup]
                self.BestMoveValue = res
                ##alpha = max(alpha, res)
                ##if alpha >= B: return alpha
                alpha = res
                if alpha >= B: return alpha

    return alpha
```

VIII - Reversi.py - La classe Reversi

Vous pouvez visualiser le contenu du fichier en cliquant sur le lien [Source Reversi.py](#)

Ce fichier initialise le jeu et gère le changement de joueur. Il comprend une unique classe **Reversi**.

Cette classe possède les attributs suivants:

- **plate** : liste représentant le plateau de jeu en mémoire.
- **player1** : objet **Player.Player** représentant le joueur ayant les pions noirs.
- **player2** : objet **Player.Player** représentant le joueur ayant les pions blancs.
- **playermain** : pointeur sur **player1** ou **player2**, représentant le joueur ayant la main.
- **wReversi** : objet **wxReversi.wxReversi** représentant le plateau de jeu à l'écran.
- **lastMove** : entier représentant le dernier coup joué.

Cette classe possède les méthodes suivantes:

- **__init__(self)** : constructeur de la classe **Reversi**.
- **Initialize(self, black, white)** : initialise le jeu.
- **ChangePlayerMain(self, n = 0)** : donne la main à l'adversaire.
- **EndGame(self)** : affiche le score en fin de partie.

Le jeu est lancé par les dernières lignes du fichier en créant une instance de **Reversi**

```
random.seed()
app = wx.PySimpleApp()
rev = Reversi()
app.MainLoop()
```

VIII.A - Initialize(self, black, white)

Cette fonction initialise le tout début du jeu. Cette fonction est très simple à lire:

- On arrête tout d'abord les threads gérant la réflexion des 2 précédents joueurs (**self.player1.endreflexion = True** et **self.player1.endgame = True**).
- Nous recréons ensuite les 2 joueurs en fonction de s'il s'agit de joueurs humains ou d'ordinateurs.
- On réinitialise le plateau de jeu.
- On lance dans un thread la phase d'attente du joueur 2 s'il s'agit d'un ordinateur (pour qu'il puisse commencer à réfléchir sur son prochain coup).
- On lance dans un thread la phase de jeu du joueur 1 s'il s'agit d'un ordinateur (pour qu'il puisse jouer).

```
def Initialize(self, black, white):
    self.player1.endreflexion = True
    self.player2.endreflexion = True
    self.player1.endgame = True
```

```

self.player2.endgame = True
while self.player1.isWaiting: continue
while self.player2.isWaiting: continue
##     self.player1.endreflexion = False
##     self.player2.endreflexion = False

if black == 'player': self.player1 = Player.Player(self, 'black')
else: self.player1 = Player.Computer(self, 'black')
if white == 'player': self.player2 = Player.Player(self, 'white')
else: self.player2 = Player.Computer(self, 'white')

self.plate = PLATEINIT[:]
glover['freemove'] = FREEMOVEINIT[:]
glover['platemove'] = PLATEMOVEINIT[:]

self.playermain = self.player1
if isinstance(self.player2, Player.Computer):
    self.player2.Initialize()
    threading.Thread(None, self.player2.Waiting, None).start()

if isinstance(self.player1, Player.Computer):
    self.player1.Initialize()
    threading.Thread(None, self.player1.Play, None).start()

```

VIII.B - ChangePlayerMain(self, n = 0)

Cette fonction est toujours appelée depuis une instance de **wxReversi.wxReversi** et les méthodes **pnIPlateLDown** ou **pnIPlateMove** qui effectue l'ajout d'un pion sur le plateau de jeu puis le changement de main.

- Si personne ne peut jouer, on quitte (**n == 2**)
- Pour ne pas avoir de mauvaise surprise, on stoppe la réflexion de tous les ordinateurs en jeu.
- On change le joueur ayant la main
- On vérifie que celui-ci peut jouer, sinon, on recharge la main
- On relance l'attente du joueur ne jouant pas et on relance la phase de jeu pour l'autre

```

def ChangePlayerMain(self, n = 0):
    if n == 2:
        self.EndGame()
        return
    ## On stoppe la réflexion de tout le monde
    self.player1.endreflexion = True
    self.player2.endreflexion = True
    while self.player1.isWaiting: continue
    while self.player2.isWaiting: continue
    self.player1.endreflexion = False
    self.player2.endreflexion = False

    ## On inverse les joueurs
    if self.playermain.joueur == 1: self.playermain = self.player2
    else: self.playermain = self.player1

    if DoitPasser(self.plate, self.playermain.joueur):
        return self.ChangePlayerMain(n + 1)

    ## On relance le jeu
    if self.playermain.joueur == 1:
        if isinstance(self.player2, Player.Computer):

```

```
        self.player2.Initialize()
        threading.Thread(None, self.player2.Waiting, None).start()
self.wReversi.timerBlack.Start(1000)
self.wReversi.timerWhite.Start(1000)
if isinstance(self.player1, Player.Computer):
    threading.Thread(None, self.player1.Play, None).start()
else:
    if isinstance(self.player1, Player.Computer):
        self.player1.Initialize()
        threading.Thread(None, self.player1.Waiting, None).start()
self.wReversi.timerBlack.Start(1000)
self.wReversi.timerWhite.Start(1000)


    if isinstance(self.player2, Player.Computer):
        threading.Thread(None, self.player2.Play, None).start()
```

IX - wxReversi.py

Vous pouvez visualiser le contenu du fichier en cliquant sur le lien [Source wxReversi.py](#)

Ce fichier contient 2 classes **wxPlate** et **wxReversi**. La première permet de gérer uniquement le plateau de jeu, la deuxième le reste de l'interface graphique. Vous aurez avec ce fichier un bel exemple d'utilisation de la bibliothèque **wxPython**.

IX.A - La classe wxPlate

wxPlate gère le plateau de jeu à travers un **wx.BufferedDC** et **wx.BufferedPaintDC**. Il existe plusieurs façon de dessiner à l'écran sans que celui-ci soit saccadé. Ici, ce couple est adapté car on se contente de dessiner à l'écran l'état du plateau sans qu'il y ait d'interaction avec l'utilisateur. Pour mettre cette classe en place, je me suis servi d'un des exemples du livre  **wxPython in Action** que je recommande fortement si vous souhaitez rapidement progresser avec cette bibliothèque. Le tout consiste à dessiner la totalité du plateau de jeu dans un buffer de type **wx.BufferedDC** puis de l'afficher à l'écran avec un **wx.BufferedPaintDC**.

Les attributs de la classe **wxPlate**:

- **parent** : parent de type **wxReversi**
- **btmplate** : Bitmap du plateau de jeu vide
- **btmwhite** : Bitmap d'un pion blanc
- **btmblack** : Bitmap d'un pion noir
- **reversi** : pointeur sur une instance de **Reversi.Reversi** pour avoir un accès complet au jeu.

Les méthodes de la classe **wxPlate**:

- **__init__(self, parent, reversi)** : constructeur initialisant les paramètres et affichant le plateau de jeu initial.
- **OnPaint(self, event = None)** : dessine à l'écran le buffer du plateau de jeu lorsqu'un rafraîchissement de l'écran est demandé.
- **Redraw(self)** : permet de forcer le rafraîchissement du plateau à l'écran
- **InitBuffer(self)** : crée en mémoire le buffer du plateau de jeu avant le dessiner dedans
- **DoDrawing(self, dc)** : remplit le buffer de la fenêtre du plateau de jeu

Au final, le code de cette classe est relativement court et très facile à lire. Dans la fonction **InitBuffer**, on crée tout d'abord un bitmap vide de la taille du plateau de jeu à l'écran. On va ensuite gérer ce bitmap à travers un **wx.BufferedDC** dans lequel on dessine le plateau de jeu.

On affiche tout d'abord la grille du plateau (**dc.DrawBitmap(self.btmplate, 0, 0, False)**). On le remplit ensuite avec les différents pions présents sur le plateau.

```
class wxPlate(wx.Window):
    """ Created: 2007.10.25 - Updated: 2008.01.08 """
    def __init__(self, parent, reversi):
        """ Created: 2007.10.25 - Updated: 2008.01.08 """
        wx.Window.__init__(self, parent, -1, size = (338, 338))
        self.parent = parent
```

```

        self.btmplate = wx.Image('.') + os.sep + 'images' + os.sep + "plate.png",
wx.BITMAP_TYPE_ANY).ConvertToBitmap()
        self.btmwhite = wx.Image('.') + os.sep + 'images' + os.sep + "white.png",
wx.BITMAP_TYPE_ANY).ConvertToBitmap()
        self.btblack = wx.Image('.') + os.sep + 'images' + os.sep + "black.png",
wx.BITMAP_TYPE_ANY).ConvertToBitmap()
        self.reversi = reversi
        self.InitBuffer()
        self.Bind(wx.EVT_PAINT, self.OnPaint)

def OnPaint(self, event = None):
    """ Created: 2007.10.25 - Updated: 2008.01.08 """
    dc = wx.BufferedPaintDC(self, self.buffer)

def Redraw(self):
    """ Created: 2007.10.25 - Updated: 2008.01.08 """
    self.InitBuffer()

def InitBuffer(self):
    """ Created: 2007.10.25 - Updated: 2008.01.08 """
    self.buffer = wx.EmptyBitmap(338, 338)
    dc = wx.BufferedDC(wx.ClientDC(self), self.buffer)
    self.DoDrawing(dc)

def DoDrawing(self, dc):
    """ Created: 2007.10.25 - Updated: 2008.01.08 """
    dc.Clear()
    dc.DrawBitmap(self.btmplate, 0, 0, False)
    for i in range(1, 9):
        for j in range(1, 9):
            if self.reversi.plate[i*10+j] == 1: dc.DrawBitmap(self.btblack, 2 + 42 * (j-1), 2
+ 42 * (i-1), False)
            elif self.reversi.plate[i*10+j] == 2: dc.DrawBitmap(self.btmwhite, 2 + 42 * (j-1),
2 + 42 * (i-1), False)

```

IX.B - La classe wxReversi

Comme je ne suis pas fan d'éditeur spécialisé aux IHM, toute l'interface graphique a été créée à la main. Mais cela se fait vraiment très bien une fois qu'on a un peu l'habitude. La classe **wxReversi** possède une barre de menu (**wx.MenuBar**, **wx.Menu**, **wx.MenuItem**) et des widgets classiques (**wx.StaticText**, **wx.Gauge**, **wx.ListBox**, **wx.Timer**). L'agencement des widgets est réalisé à travers des **wx.BoxSizer**, **wx.StaticBox** et **wx.StaticBoxSizer**.

Il vous suffira de lire linéairement la fonction `__init__(self, reversi)` pour comprendre comment l'interface est construite.

Cette classe possède de nombreuses méthodes qui correspondent toutes à des évènements depuis la barre de menu sauf les 2 fonctions **pnIPlateLDown(self, event)** et **pnIPlateMove(self, r, score)**. Là encore, vous devriez tout comprendre après une simple lecture du code. Nous allons simplement nous attarder sur ces 2 dernières méthodes qui gèrent la pose d'un pion par un joueur humain (**pnIPlateLDown**) ou par un ordinateur (**pnIPlateMove**)

IX.B.1 - pnIPlateLDown(self, event)

Un joueur humain joue son tour de jeu en cliquant avec la souris sur la case sur laquelle il veut poser un pion. On vérifie ainsi tout d'abord qu'un clic correspond bien à un joueur humain qui a la main. On récupère ensuite la case cliquée ($X = \text{event.X} // 42 + 1$ et $Y = \text{event.Y} // 42 + 1$). Si la case cliquée est une case jouable (**TestLeClic(self.reversi.plate, Y*10+ X, self.reversi.playermain.joueur)**), on pose le pion et on change le tour de jeu (**self.reversi.ChangePlayerMain()**)

```

def pnlPlateLDown(self, event):
    if (self.reversi.playermain.joueur == 1 and isinstance(self.reversi.player1,
Player.Player)) \
    or (self.reversi.playermain.joueur == 2 and isinstance(self.reversi.player2,
Player.Player)):
        X = event.X // 42 + 1
        Y = event.Y // 42 + 1

        if TestLeClic(self.reversi.plate, Y*10+ X, self.reversi.playermain.joueur):
            self.timerWhite.Stop()
            self.timerBlack.Stop()
            Reverse(self.reversi.plate, Y*10+ X, self.reversi.playermain.joueur)
            b, w = Score(self.reversi.plate)
            self.lbMove.InsertItems([str(b+w) + '. ' +
self.reversi.playermain.couleur.upper()[0] + " - " + chr(64+X) + " "+str(9-Y) + " 0"], 0)
            self.pnlPlate.Redraw()
            self.reversi.lastMove = Y * 10 + X
            glovar['freemove'].remove(Y * 10 + X)
            glovar['platemove'].append(Y * 10 + X)
            self.reversi.ChangePlayerMain()
            event.Skip()

```

IX.B.2 - pnlPlateMove(self, r, score)

Un ordinateur effectue son coup en appelant la fonction **pnlPlateMove**. Le premier test (**TestLeClic(self.reversi.plate, r, self.reversi.playermain.joueur)**) n'est utile que pour éviter des effets de bords quand on relance une partie. Le reste de la fonction est identique à ce qu'on peut avoir pour un joueur humain.

```

def pnlPlateMove(self, r, score):
    if TestLeClic(self.reversi.plate, r, self.reversi.playermain.joueur):
        self.timerWhite.Stop()
        self.timerBlack.Stop()
        Reverse(self.reversi.plate, r, self.reversi.playermain.joueur)

        b, w = Score(self.reversi.plate)
        Y, X = divmod(r, 10)
        self.reversi.lastMove = r
        glovar['freemove'].remove(r)
        glovar['platemove'].append(r)
        self.lbMove.InsertItems([str(b+w) + '. ' + self.reversi.playermain.couleur.upper()[0] +
" - " + chr(64+X) + " "+str(9-Y) + " "+str(score)], 0)

        self.pnlPlate.Redraw()
        self.reversi.ChangePlayerMain()

```

X - Conclusion

Le programme peut encore être grandement amélioré au niveau de ses possibilités. Dans la version actuelle, il devrait toutefois ne pas y subsister trop de bogues.

On peut énumérer quelques améliorations souhaitables comme:

- Ralentissement lorsqu'on redimensionne la fenêtre pendant la réflexion de l'ordinateur.
- Non prise en compte des modifications des paramètres de réflexion de l'ordinateur pendant la partie en cours.
- Prévoir le fait que l'ordinateur puisse ne pas forcément choisir le coup le meilleur pour éviter la redondance des débuts de partie
- Développer les ouvertures
- Possibilité d'annuler un coup

XI - Téléchargement

Version	Date	Taille	Mode FTP	Mode HTTP de secours
1.0.2.1.0	2008.01.17	11.9 ko	pyReversi.zip	pyReversi.zip

