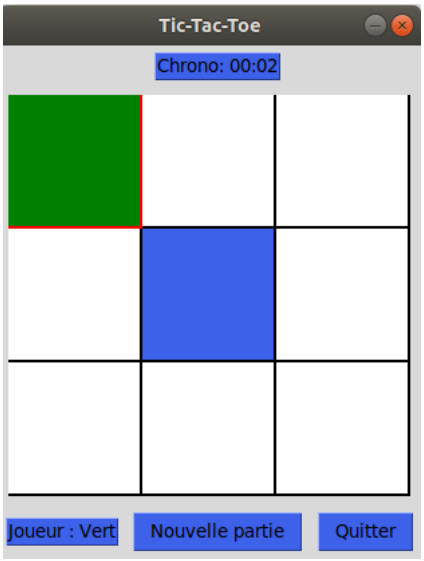
Introduction

L'objectif de ce mini projet est de programmer un jeu de Tic-Tac-Toe ou Morpion avec une opposition de deux joueurs humains. L'application devra se présenter sous la forme d'une interface graphique similaire à celle présentée ci-dessous :



Vous compléterez le squelette de code fourni dans le fichier `MorpionEleve2021.py`, en insérant des annotations sous forme de commentaires lorsque les questions ne portent pas sur l'écriture de code Python.

1 Fixons le cadre

Précisons les règles du jeu :

- ☞ *Le joueur qui commence est choisi aléatoirement.*
- ☞ *Chaque joueur identifié par une couleur, joue à tour de rôle, l'un en cliquant sur la case sélectionnée avec le bouton gauche de la souris, l'autre en déplaçant un curseur (cadre rouge autour de la case sélectionnée) avec les flèches du pavé directionnel puis en validant la sélection par un appui sur la touche `Espace`.*
- ☞ *Lorsqu'une case libre a été sélectionnée par un joueur elle prend sa couleur caractéristique, si elle n'est pas libre une fenêtre pop up le signale au joueur.*
- ☞ *Chaque tour doit s'effectuer en temps limité de 5 seconde, le décompte du temps étant affiché par un chronomètre. Une seconde de pause s'intercale entre deux tours consécutifs.*
- ☞ *Le jeu s'arrête soit parce qu'un joueur a gagné en réussissant un alignement de trois cases (horizontal, diagonal ou vertical), soir parce qu'un joueur n'a pas sélectionné de case dans le temps imparti (son adversaire est désigné vainqueur), soit parce que les neuf tours ont été épuisés sans vainqueur (match nul).*

Pour réaliser l'interface graphique, nous utiliserons le module `tkinter` livré avec la distribution standard de `Python`. Une documentation assez complète sur `tkinter` est disponible sur le web :

- ☞ une documentation en français se trouve sur `http://tkinter.fdex.eu/`.
- ☞ de nombreux exemples sont développés sur `http://fsincere.free.fr/isn/python/cours_python_tkinter.php?version=3`

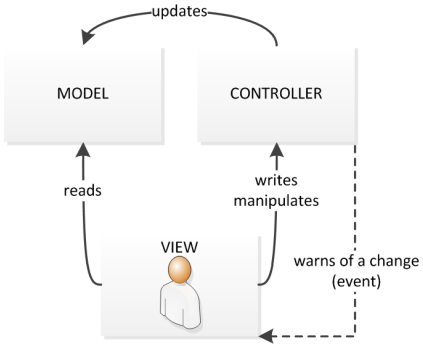
Par ailleurs, si nous souhaitons afficher le message de fin de partie dans une fenêtre pop-up, nous aurons besoin du sous-module de `tkinter` nommé `tkinter.messagebox`.
Enfin, pour choisir aléatoirement le joueur qui commence, nous utiliserons le module `random` et pour gérer le temps, nous pourrons employer le module `time`.
Commençons donc par éditer avec `Idle` ou `Spyder` le fichier `MorpionEleve2021.py`. En préambule, on trouve les lignes suivantes pour importer les modules nécessaires, que nous compléterons si besoin en cours de développement :

```
## Modules

from tkinter import *
import tkinter.messagebox
import random
import time
```

`tkinter` sera le plus utilisé, nous pouvons importer les noms de tous ses éléments dans l'espace de nom de notre programme. Les autres modules ne seront sollicités que ponctuellement et nous importons juste le module.

2 Architecture logicielle

Lorsque le programme sera terminé, il comptera environ 200 lignes, c'est modeste mais suffisant pour l'organiser en trois grandes parties selon l'architecture Modèle Vue Contrôleur.



- Le code commence par l'import des modules/bibliothèques nécessaires.
- Ensuite vient la partie *Modèle* avec la définition des données et des fonctions permettant de les manipuler. On distingue les constantes (couleurs, identifiants des joueurs) et la structure de données principale retournée par la fonction `initialiser_grille` qui est une liste de listes représentant la grille de jeu 3×3 qui va enregistrer les marques déposées par les joueurs. Ces fonctions seront complétées à la fin du projet.
- La partie *Vue* va rassembler les fonctions permettant de définir l'interface graphique et des fonctions utilitaires qui leur sont liées. La fonction `interface\_jeu()` crée puis renvoie les objets graphiques appelés *widgets*.
- La partie *Contrôleur* réunit les gestionnaires d'événements qui vont s'occuper des interactions entre l'utilisateur et l'interface graphique.
- Enfin le programme principal définit les *widgets* renvoyés par `interface_jeu()` comme variables globales, fait la liaison avec les gestionnaires d'événements (voir exercice 7) puis la boucle réceptionnaire d'événements. Le nombre d'attributs de l'interface est important, pour organiser ce foisonnement il faudrait utiliser la programmation orientée objet et définir une classe Interface.

```
# Programme principal
if __name__ == "__main__":
    fenetre, textJoueur, etiquetteJoueur, bouton_quitter, bouton_jouer,
        textHorloge, horloge, plateau, case_vers_identifiant,
        identifiant_vers_case = interface_jeu()
    #Liaisons événements/gestionnaires d'événements
    plateau.bind('<ButtonPress-1>', clic_gauche)
    plateau.bind_all('<KeyPress>', appui_touche)
    #boucle réceptionnaire d'événement
    fenetre.mainloop()
```

Exercice 1

Repérer les parties *Modèle, Vue, Contrôleur* et le programme principal dans `MorpionEleve2021.py`.

3 Partie Vue : mise en place de l'interface graphique

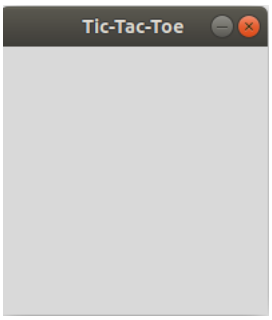
Dans cette partie, nous allons mettre en place les différents éléments de l'interface graphique.
Pour mieux comprendre le fonctionnement du module `tkinter`, vous travaillerez d'abord avec un autre squelette de code fourni dans le fichier `sandbox_eleve.py`.

3.1 Fenêtre racine et réceptionnaire d'événements

Exercice 2

L'exécution du programme `sandbox_eleve.py` produit le résultat ci-contre.
Le code minimal pour afficher la fenêtre est donné ci-dessous.

```
1  from tkinter import *
2
3  #----fenetre principale----
4  fenetre = Tk()
5  fenetre.title("Tic-Tac-Toe")
6  #construire la fenetre à 100 pixels du cote gauche de l'
       ecran et 150 pixels du cote haut
7  fenetre.geometry("+100+150")
8  #interdire la modification de la taille de la fenetre
9  fenetre.resizable(width=False, height=False)
10
11
12 #----boucle de réception des événements----
13 #à  placer à la fin du programme
14 fenetre.mainloop()
```

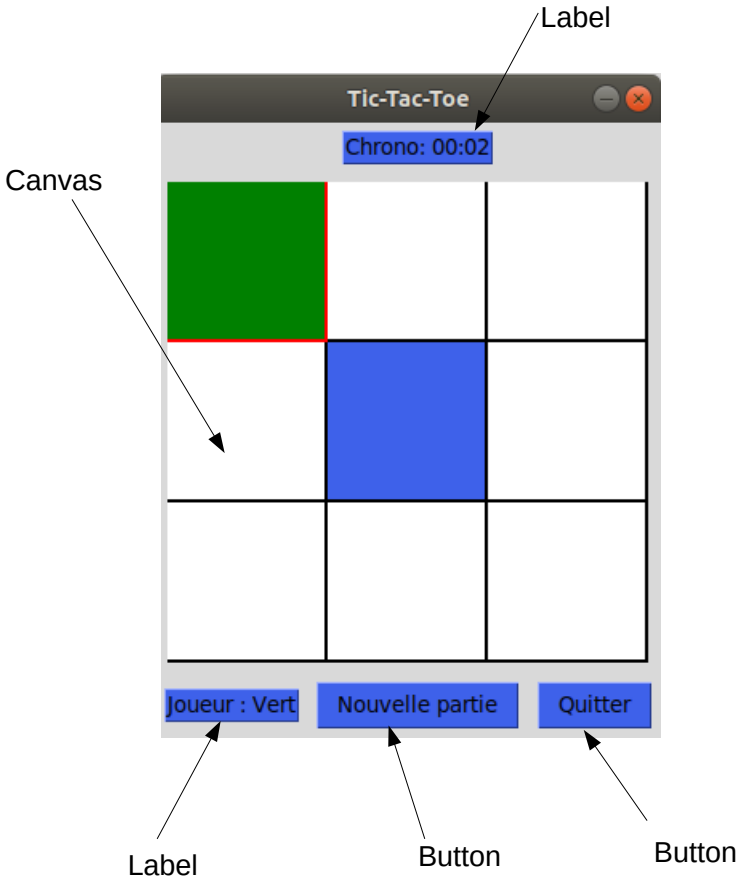


1. Quelles sont les interactions possibles avec la fenêtre ?
2. Modifier le positionnement de la fenêtre.

3. Rechercher dans la documentation http://tkinter.fdex.eu/doc/intro.html le rôle de l'instruction `fenetre.mainloop()`.

3.2 Widgets

Nous allons disposer dans la fenêtre racine `fenetre` des objets graphiques appelés **widgets**.
On donne ci-dessous un aperçu de la fenêtre racine contenant les six widgets.
Ils seront rattachés à `fenetre` par un lien de dépendance et les instructions de construction vont être placées après la création de `fenetre` et juste avant **la boucle de réception des événements** `fenetre.mainloop()`.



3.2.1 Widget Canvas

Un **canevas** définit une zone rectangulaire qui peut contenir des figures géométriques (rectangles, arcs, ellipses, textes ...) ou des images bitmap.
La documentation se trouve sur la page http://tkinter.fdex.eu/doc/caw.html.
Nous utilisons un canevas pour représenter la grille de jeu. Voici les instructions pour le faire apparaître dans la fenêtre. Le canevas définit dans la partie *Vue* l'apparence de la grille (les couleurs des joueurs) alors que la variable `grille` définie par la fonction `initialiser_grille` de la partie *Modèle* contiendra les données de la grille (les marques numériques des joueurs) sur lesquelles s'effectuera la validation du vainqueur.

```
1 #----Canevas----
2 #il faut diriger les entrées claviers vers le canevas qui n'a pas le focus par
      défaut
3 #voir http://tkinter.fdex.eu/doc/focus.html#focus
4 plateau = Canvas(fenetre, width = COTE_CASE * 3, height = COTE_CASE * 3, bg =
      'white', takefocus = 1)
```

```
5 plateau.grid(row = 1, column = 0, columnspan = 3, padx = 5, pady = 5)
```

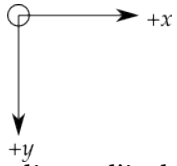
☞ En ligne 4 on crée le canevas avec le constructeur `Canvas` auquel on passe plusieurs options dont la première, obligatoire, précise qu'il est rattaché à la fenêtre racine `fenetre`. On assigne le canevas à une variable `plateau`, ce qui permettra de le manipuler dans la suite du programme.

☞ La ligne 4 ne suffit pas pour afficher le canevas dans la fenêtre. Il faut le placer dans la fenêtre avec la méthode `grid` qui est un **gestionnaire de placement**. Il existe une méthode similaire `pack`, il ne faut pas mélanger les deux. `grid` place le widget selon un découpage de la fenêtre en lignes (`row`) et colonnes (`column`) indexées à partir de 0. L'option `columnspan` permet d'étendre le widget sur plusieurs colonnes. Dans notre cas, nous découperons la fenêtre en 3 colonnes (de 0 à 2) et 3 lignes (de 0 à 2).
Les options `padx` et `pady` permettent de définir un espace en pixels autour du widget.

☞ Une propriété de widget est accessible en lecture par la méthode `cget` et en écriture par la méthode `configure`. Il existe une syntaxe raccourcie avec l'opérateur `[]` comme pour les dictionnaires.

```
In [14]: plateau['background']
Out[14]: 'white'

In [15]: plateau['background'] = 'red'

In [16]: plateau.cget('background')
Out[16]: 'red'

In [17]: plateau.configure(background = 'white')

In [18]: plateau['bg'] #'bg' est un raccourci pour 'background'
Out[18]: 'white'
```

Chaque case de la grille va être dessinée sur le canevas, muni d'un repère d'origine son coin supérieur gauche avec pour axe des abscisses d'orientation Ouest→Est le bord supérieur et pour axe des ordonnées d'orientation Nord→Sud, le bord gauche.



Dans le code ci-dessous, la méthode `create_rectangle` permet de représenter la case en ligne d'index 2 (donc la 3e ligne) et colonne d'index 1 (donc la 2e colonne) de la grille comme un rectangle carré de côté `COTE_CASE` pixels, dont le bord est de couleur `COULEUR_BORD_CASE` avec une épaisseur de 2 pixels et le fonds de couleur `COULEUR_CASE[0]`. Les coordonnées du coin supérieur gauche sont `(x, y)` et celles du coin inférieur droit sont `(x + COTE_CASE, y + COTE_CASE)`.

```
x = 1 * COTE_CASE
y = 2 * COTE_CASE
iden = plateau.create_rectangle(x, y , x + COTE_CASE, y + COTE_CASE, outline =
      COULEUR_BORD_CASE,
        fill= COULEUR_CASE[0], width=2)
```

☞ Attention, les colonnes correspondent aux abscisses et les lignes aux ordonnées.

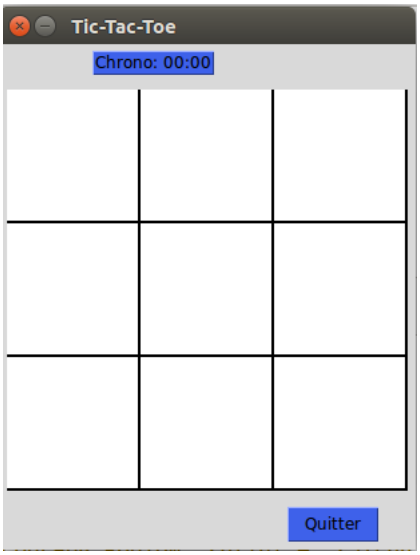
☞ La méthode `create_rectangle` renvoie l'identifiant unique de l'**item graphique** ( le plus petit est 1). Nous le récupérons dans une variable `iden`. Pour retrouver la case, repérée par (ligne, colonne), à partir de l'identifiant et réciproquement, nous allons créer deux listes de correspondance initialisée avec des valeurs par défaut et nous y stockerons `iden`. Nous pourrons ainsi manipuler l'item graphique dans la suite du programme, pour changer son apparence.

```
case_vers_identifiant = [ [0 for j in range(3)] for i in range(3) ]
identifiant_vers_case = [ (0,0) for k in range(9) ]
#case en ligne 2 et colonne 1 voir code précédent
case_vers_identifiant[2][1] = iden
identifiant_vers_case[iden - 1] = (2, 1)
```

Exercice 3

Dans le fichier MorpionEleve2021.py, compléter le code de la fonction interface_jeu (à partir de la ligne 195), pour tracer la grille complète et remplir les listes de correspondance case_vers_identifiant et identifiant_vers_case avec les identifiants de tous les items graphiques créés.

On doit obtenir la figure ci-contre sur le canevas.



Exercice 4

Compléter la fonction initialiser_plateau du fichier MorpionEleve2021.py pour qu'elle initialise les neuf rectangles du plateau avec la couleur COULEUR_CASE[0].
On utilisera la syntaxe :
plateau.itemconfig("ici l'identifiant de la case", fill = "ici la couleur").
La documentation se trouve sur http://tkinter.fdex.eu/doc/caw.html#Canvas.itemconfigure.

```
def initialiser_plateau(plateau, case_vers_identifiant):
    """Initialise le plateau"""
    for li in range(3):
        for co in range(3):
            "compléter"
```

3.2.2 Widgets Label

Le code ci-dessous permet de créer un widget horloge contenant un texte figé pour l'instant. Ce widget est défini dans la fonction initialiser_plateau de MorpionEleve2021.py à partir de la ligne 205.

```
1 textHorloge = StringVar()
2 textHorloge.set("Chrono: {:02d}:{:02d}".format(0, 0))
3 horloge = Label(fenetre, textvariable = textHorloge, bg = COULEUR_BOUTON,
      relief = 'raised')
4 horloge.grid(row = 0, column = 1, padx = 5, pady = 5)
```

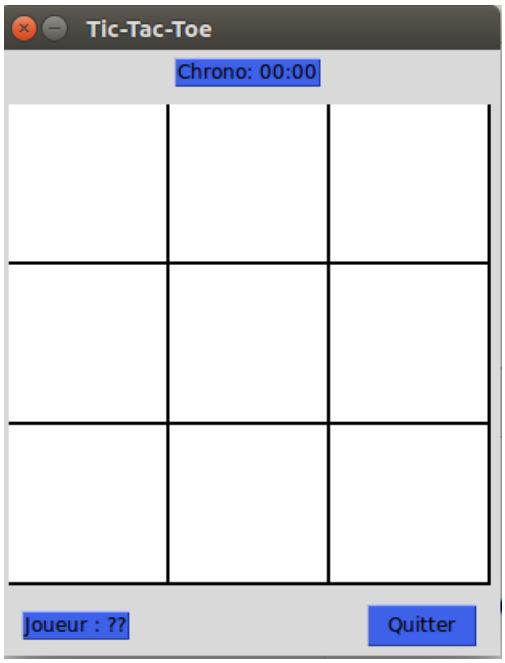
- En ligne 3, on crée le widget avec le constructeur `Label` qui sert à afficher une ou plusieurs lignes de texte. On précise obligatoirement que le widget est rattaché à `fenetre`, les options suivantes, facultatives, sont détaillées dans http://tkinter.fdex.eu/doc/labw.html. Le widget est positionné en ligne 4 du code, avec `grid` : en première ligne (index 0) et deuxième colonne (index 1) de la grille de positionnement.
- L'option `textvariable` lie le texte affiché à une variable de contrôle `textHorloge` qui est définie précédemment en ligne 1 avec `StringVar`.
- En ligne 2, le contenu de la variable de contrôle `textHorloge`, de type texte, est modifié avec la méthode `set`. On utilise la méthode `format` pour le formatage de la chaîne de caractères. La documentation sur `format` se trouve sur https://docs.python.org/fr/3.5/library/string.html#formatstrings.

Pour animer l'horloge, nous allons utiliser la fonction `chronometre` qui se trouve déjà dans `MorpionEleve2021.py`.

```
1 def chronometre():
2     """Fonction récursive qui met à jour l'horloge lors d'un tour"""
3     duree = time.time()- debut_tour
4     minute, seconde = divmod(int(duree), 60)
5     textHorloge.set("Chrono: {:02d}:{:02d}".format(minute, seconde))
6     if duree < TEMPS_MAX:
7         fenetre.after(1000, chronometre)
```

- En ligne 3, on calcule le temps écoulé, en secondes, depuis l'origine des temps stockée dans la variable globale `debut_tour`, cette dernière sera définie dans la fonction `nouvelle_partie`.
- En ligne 4, on convertit `duree` en minutes et secondes avec la fonction `divmod` qui retourne le quotient et le reste d'une division euclidienne.
- En ligne 5, on modifie la variable de contrôle `textHorloge` avec sa méthode `set`.
- En ligne 7, on teste si le temps maximum pour choisir une case (variable globale `TEMPS_MAX`) est dépassé et si ce n'est pas le cas, on appelle de nouveau la fonction `chronometre` après un temps d'attente de 1000 millisecondes avec la méthode `after` de la fenêtre racine.
  La fonction `chronometre` pouvant s'appeler elle-même, c'est une **fonction récursive**.

Exercice 5



Dans la fonction `initialiser_plateau` de `MorpionEleve2021.py`, compléter à partir de la ligne 213, le code permettant de créer et positionner dans la fenêtre racine un nouveau widget de type `Label` :

- Il doit être stocké dans la variable `etiquetteJoueur` et placé en `row = 2` et `column = 0`.
- Le texte doit être lié à la variable de contrôle `textJoueur = StringVar()`.
- Ce texte doit indiquer le joueur courant ou "Joueur :??" si aucune partie n'est en cours.
- L'affichage obtenu doit être similaire à celui donné ci-dessus.

3.2.3 Widgets Button

Nous allons placer dans la fenêtre des widgets de type `Button` qui sont des boutons d'action.
La documentation sur les widgets `Button` se trouve sur http://tkinter.fdex.eu/doc/bw.html.

```
1 #----Bouton pour quitter----
2 bouton_quitter = Button(fenetre, text="Quitter", bg = COULEUR_BOUTON, relief =
      'raised', command = quitter)
3 bouton_quitter.grid(row = 2, column = 2, padx = 5, pady = 5)
```

Détaillons le code ci-dessus qui met en place un bouton qui ferme proprement la fenêtre si on clique dessus. Ce widget est défini dans la fonction `initialiser_plateau` de `MorpionEleve2021.py` à partir de la ligne 223.

☞ En ligne 2, on assigne à la variable `bouton_quitter` un widget `Button` dont l'option `command = quitter` signifie que la fonction `quitter` sera appelée lorsque le bouton recevra un clic gauche.

⚠ **Attention, le paramètre de `command` doit être juste le nom de la fonction, sans les parenthèses qui déclenchent son appel.**

Cette fonction est définie en amont et elle permet de fermer la fenêtre proprement.

```
def quitter():
    """Quitter proprement la fenetre"""
    fenetre.quit()
    fenetre.destroy()
```

Exercice 6

1. Dans la fonction `initialiser_plateau` de `MorpionEleve2021.py`, compléter à partir de la ligne 213, lc code permettant de créer et positionner dans la fenêtre racine un nouveau widget de type `Button` :

   - Il doit être stocké dans la variable `bouton_jouer` et commander l'appel de la fonction `nouvelle_partie` qui elle même lance la fonction `nouveau_tour`.
   - Ce widget doit être placé en ligne d'index `row = 2` et colonne d'index `column = 1`.

2. Le fichier `MorpionEleve2021.py` contient déjà les fonctions `nouvelle_partie` et `nouveau_tour` dont on donne le code ci-dessous.

   a. Quelle est la portée des variables déclarées avec le mot clef `global` ?

   b. Décrire la séquence d'instruction réalisée par l'appel `nouvelle_partie()`.

   c. Quel est le type des variables `choix_joueur1` et `choix_joueur2` ?
   Ces variables servent de *drapeaux* pour indiquer si lors du tour du joueur concerné, une action de sélection de case a déjà été réalisée. Un joueur ne peut en effet sélectionner qu'une seule case lors de son tour.

   d. Quel est le point commun entre la fonction `nouveau_tour` et la fonction `chronometre` ?

   e. Une partie oppose un joueur 1 et un joueur 2. L'un des deux peut sélectionner une case avec la souris et l'autre avec le clavier en déplaçant un curseur rectangulaire rouge qui entoure la case en haut à gauche de coordonnées $(0,0)$.
   D'après le code de la fonction `nouveau_tour`, quel joueur sélectionne avec le clavier ?

   f. Rechercher dans le fichier `MorpionEleve2021.py`, le message affiché par l'appel de fonction `message_fin(vainqueur)`.

   g. Décrire la séquence d'instruction réalisée par l'appel `nouveau_tour()`.

```
def nouvelle_partie():
    """Initialise et lance une nouvelle partie"""
    global vainqueur, joueur, tour, grille, choix_joueur1, choix_joueur2
    grille = initialiser_grille()
    tour = 1
    joueur = random.randint(1, 2)
    textJoueur.set("Joueur : {}".format(COULEUR_JOUEUR[joueur]))
    initialiser_plateau(plateau, case_vers_identifiant)
    vainqueur = 0
    choix_joueur1 = False
```

```
    choix_joueur2 = False
    nouveau_tour()

def nouveau_tour():
    """Fonction qui lance un nouveau tour"""
    global vainqueur,joueur, tour, debut_tour, curseur, xcurseur,
        ycurseur, choix_joueur1, choix_joueur2
    #on positionne les joueurs
    joueur_precedent = joueur
    joueur = adversaire(joueur)
    #mise à jour de l'affichage du joueur
    textJoueur.set("Joueur : {}".format(COULEUR_JOUEUR[joueur]))
    #si le joueur précédent n'a pas cliqué le joueur courant est
        vainqueur
    #attention à bien mettre des parenthèses autour du or pour changer la
        priorité par défaut des opérateurs booléens
    if tour >= 2 and (joueur_precedent == JOUEUR_1 and not choix_joueur1
        \
    or joueur_precedent == JOUEUR_2 and not choix_joueur2):
        vainqueur = joueur
    #sinon on vérifie si le joueur précédent a gagné
    elif verifier(grille, joueur_precedent):
        vainqueur = joueur_precedent
    #On affiche le vainqueur sil y en un ou si on a atteint le 10ème tour
    if tour == 10 or vainqueur != 0:
        if joueur == JOUEUR_1:
            plateau.delete(curseur)
        message_fin(vainqueur)
    else: #sinon on commence un nouveau tour
        #pause de 1 seconde avant le changement de joueur
        time.sleep(2)
        #on démarre le chronomètre
        debut_tour = time.time()
        chronometre()
        #on positionne à False les booléens indiquant si le joueur courant
            a fait son choix
        choix_joueur1 = False
        choix_joueur2 = False
        #pour le joueur 2
        if joueur == JOUEUR_2:
            xcurseur, ycurseur = 0, 0
            curseur = plateau.create_rectangle(xcurseur, ycurseur ,
                xcurseur + COTE_CASE, ycurseur + COTE_CASE,
        outline = 'red', fill='', width=2)
        elif tour >= 2: #pour le joueur1 à partir du tour 2
            plateau.delete(curseur)
        #on incrémente le compteur de tour pour le tour suivant
        tour = tour + 1
        #on attend TEMPS_MAX secondes (argument en millisecondes) avant de
            commencer un nouveau tour
```

```
54        #pendant la durée d'un tour les gestionnaires d'événements gèrent
              les actions
55        fenetre.after(TEMPS_MAX * 1000, nouveau_tour)
```

4 Partie contrôleur, les gestionnaires d'événements

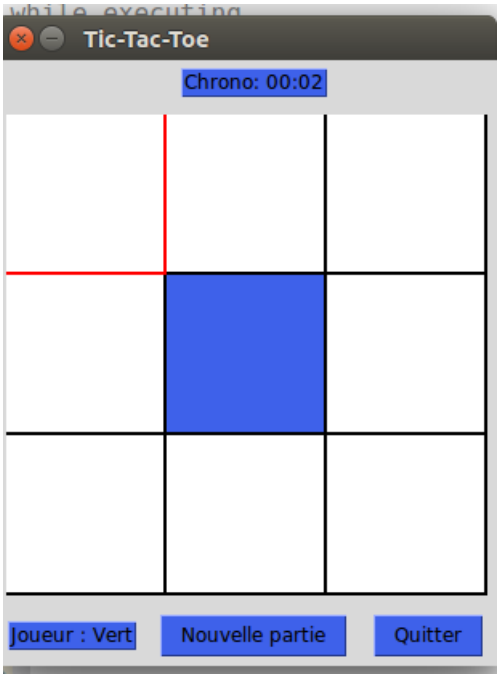
La **boucle réceptionnaire d'événements** `fenetre.mainloop()` scrute en permanence les **événements** qui peuvent survenir dans les widgets. Un **événement** peut être l'appui ou le relâchement d'une touche de clavier, un clic de souris ...
La documentation sur les événements se trouve sur http://tkinter.fdex.eu/doc/event.html.
Nous avons besoin de gérer deux types d'événements sur le widget `plateau` de type `Canvas`.

- ☞ un clic gauche sur une case permet de la sélectionner en changeant sa couleur s'il s'agit du tour du joueur autorisé à cliquer, s'il n'a pas déjà sélectionné une case pendant son tour, si la case est libre et s'il n'y a pas encore de vainqueur.
- ☞ des appuis sur les touches `Up`,`Down`, `Left`, `Right` du pavé directionnel permettent de déplacer le curseur (rectangle rouge) puis de sélectionner une case avec la touche `space` s'il s'agit du tour du joueur autorisé, s'il n'a pas déjà sélectionné une case pendant son tour, si la case est libre et s'il n'y a pas encore de vainqueur.

Un clic gauche de la souris est l'événement `'<ButtonPress-1>'`, alors qu'un appui sur une touche est l'événement `'<KeyPress>'`.
Pour chacun de ces événements, le fichier `MorpionEleve2021.py` contient dans la partie *Contrôleur*, des fonctions qui jouent le rôle de **gestionnaire d'événement**.
Chaque gestionnaire d'événement `clic_gauche` ou `appui_touche` est attaché à l'événement ciblé par un **binder** grâce à la méthode `bind` de l'objet canevas stocké dans la variable `plateau`. Ces instructions sont placées dans le programme principal, juste avant l'appel à la boucle réceptionnaire d'événements `fenetre.mainloop()`.

⚠ Attention, comme pour le paramètre `command` d'un widget `Button`, lors de la liaison, on note juste le nom du gestionnaire, sans parenthèses.
De plus il faut définir le gestionnaire avant d'établir la liaison.

```
1  #Liaisons événements/gestionnaires d'événements
2  plateau.bind('<ButtonPress-1>', clic_gauche)
3  plateau.bind_all('<KeyPress>', appui_touche)
4
5  fenetre.mainloop()
```



Exercice 7

Intéressons-nous au code fourni dans `MorpionEleve2021.py` pour le gestionnaire d'événement `'<ButtonPress-1>'`. L'événement capturé est stocké dans le paramètre `event`.

```
def clic_gauche(event):
    """Gestionnaire de clic à gauche, pour le joueur 1"""
    global choix_joueur1
    #clic gauche bloqué si une nouvelle partie n'est pas lancée
    #A MODIFIER pour bloquer le clic s'il y a un vainqueur
    if joueur == JOUEUR_1 and not choix_joueur1 and tour != 0:
        (iden,) = plateau.find_closest(event.x, event.y)
        (lig, col) = identifiant_vers_case[iden - 1]
        if grille[lig][col] != 0:
            showinfo("Erreur", "Case non libre")
        else:
            grille[lig][col] = joueur
            plateau.itemconfig(iden, fill = COULEUR_CASE[joueur])
            choix_joueur1 = True
```

1. Expliquer les instructions qui permettent de récupérer la case concernée à partir des coordonnées du clic.
2. Quels rôles différents jouent les variables `grille` et `plateau`?
3. Pourquoi la variable `choix_joueur1` est-elle déclarée avec le mot clef `global`?

Exercice 8

Intéressons-nous au code fourni dans `MorpionEleve2021.py` pour le gestionnaire d'événement `'<KeyPress>'`. L'événement capturé est stocké dans le paramètre `event`.

```
def appui_touche(event):
    """Gestionnaire d'appui sur une touche, pour le joueur 2"""
    global choix_joueur2, xcurseur, ycurseur
    if joueur == JOUEUR_2 and not choix_joueur2 and tour != 0 and vainqueur ==
          0:
        if event.keysym == 'Left' and xcurseur >= COTE_CASE:
            plateau.coords(curseur, xcurseur - COTE_CASE, ycurseur, xcurseur,
                 ycurseur + COTE_CASE)
            xcurseur, ycurseur = xcurseur - COTE_CASE, ycurseur
        elif event.keysym == 'Up' and ycurseur >= COTE_CASE:
            plateau.coords(curseur, xcurseur, ycurseur - COTE_CASE, xcurseur +
                 COTE_CASE, ycurseur)
            xcurseur, ycurseur = xcurseur, ycurseur - COTE_CASE
        #A MODIFIER : déplacement vers le bas ou la droite
        elif event.keysym == 'space':
            (lig, col) = (ycurseur//COTE_CASE, xcurseur//COTE_CASE)
            if grille[lig][col] == 0:
                iden = case_vers_identifiant[lig][col]
                grille[lig][col] = joueur
                plateau.itemconfig(iden, fill = COULEUR_CASE[joueur])
                choix_joueur2 = True
            else:
                showinfo("Erreur", "Case non libre")
```

1. Expliquer les modifications des variables `xcurseur` et `ycurseur` lors du déplacement vers la gauche du curseur et la condition nécessaire pour qu'un tel déplacement soit possible.

   Quelle instruction permet de déplacer le rectangle représentant le curseur dans la fenêtre graphique ?

2. Compléter la fonction `appui_touche` avec des instructions permettent des déplacements vers le haut ou vers le bas.

5 Partie modèle du jeu et finalisation

Pour finaliser le jeu, il nous reste à compléter les fonctions de la partie *Modèle* fournie dans `MorpionEleve2021.py`.

Exercice 9

1. Compléter la fonction `copie(grille)` pour qu'elle retourne une copie profonde/déréférencée de la grille passée en argument.

2. Compléter la fonction `liste_alignement(grille)` pour qu'elle retourne une liste de 8 listes, représentant chacune les valeurs des cases de l'un des 8 alignements possibles sur le plateau . En effet, on a 3 lignes, 3 colonnes et 2 diagonales.

```
In [12]: liste_alignement([[0, 0, 2], [0, 1, 0], [2, 0, 1]])
Out[12]:
[[0, 0, 2],
 [0, 1, 0],
 [2, 0, 1],
 [0, 0, 2],
 [0, 1, 0],
 [2, 0, 1],
 [0, 1, 1],
 [2, 1, 2]]
```

3. Compléter la fonction `nb_occurrence(liste, joueur)` qui prend en argument la liste des valeurs de trois cases alignées sur le plateau et le code d'un joueur (1 ou 2) et qui retourne un couple dont le premier élément est le nombre de cases marquées par le joueur dans l'alignement sélectionné et le deuxième est le nombre de cases marquées par son adversaire.

```
In [17]: nb_occurrence([1,2,2], 1)
Out[17]: (1, 2)

In [18]: nb_occurrence([1,2,2], 2)
Out[18]: (2, 1)
```

4. Compléter la fonction `verifier(grille, joueur)` pour qu'elle retourne un booléen, `True` si parmi les 8 alignements possibles, l'un contient 3 symboles du joueur sélectionné et `False` sinon.

```
In [20]: verifier([[0, 1, 2], [0, 2, 0], [2, 1, 1]], 2)
Out[20]: True
```