Tetris

En cherchant un peu, j'ai trouvé le programme ci-dessous sur internet :
https://codereview.stackexchange.com/questions/193495/tkinter-1-player-tetris-game
Cela a été traduit en français dans votre programme, mais il ne s'agissait pas d'un projet de traduction… Je comprends que l'on s'inspire d'un jeu existant, que l'on cherche à en comprendre le fonctionnement. Ainsi, on arrive mieux et plus vite, à réaliser ce que l'on souhaite. Mais ici il s'agit d'une sorte de plagia, de tricherie. Quelle est la part réalisée par vos soins (en dehors de la traduction), on peut se le demander. J'ai donc comparé la version initiale et la version francisée de votre projet.

J'ai trouvé une fonction :
```
def love_cookies(self):
        return rd.choice(("saddle brown","sandy brown"))
```
qui colorie les cases de façon aléatoire entre deux couleurs : c'est original !

Quelques menues modifications :
self.vitesse = 100 au lieu de 500
self.root.geometry("420x520") au lieu de 500x500.
Les commandes sont attribuées à d'autres touches :
"q","s","z","+" au lieu de "a","d","s","w", ce qui n'est pas forcément bien vu mais il s'agit encore une fois d'un choix mineur.
Les scores sont modifiés :
100,250,550,1300 au lieu de 400,1000,3000,12000
Encore un choix mineur.
J'ai repéré aussi un autre changement mineur, un réglage de la vitesse :
self.vitesse = 500 - (level - 1) * 20 (au lieu de 25)

Comme vous l'avez dit dans votre compte-rendu vous avez changé
def quitter(self): self.root.destroy() à la place de quit()
Plus grave, vous dites « nous avons dû définir de nombreuses fonctions. »
Et vous citez :
```
def sens_pivoter(self):
   tournage = self.__rotation()
   directions = [(tournage[i][0] - self.__coords[i][0],
      tournage[i][1] - self.__coords[i][1]) for i in range(len(self.__coords))]
```
Mais qu'avez vous fait à part traduire :
```
def rotate_directions(self):
  rotated = self.__rotate()
  directions = [(rotated[i][0] - self.__coords[i][0],
     rotated[i][1] - self.__coords[i][1]) for i in range(len(self.__coords))]
  return directions
```

Vous dites aussi, « Nous avons aussi dû créer des fonctions... » Et vous citez :
```
def fin_partie(self):
  if not self.piece_en_jeu.move((0,1)):
    self.jouer_jeu_ds = Button(self.root, text="Rejouer", command=self.rejouer_pt)
    self.quitter_ds = Button(self.root, text="Quitter", command=self.quitter)
    self.jouer_jeu_ds.place(x = Cookiemino.LARGEUR_JEU + 10, y = 200, width=100,
    height=25)
    self.quitter_ds.place(x = Cookiemino.LARGEUR_JEU + 10, y = 300, width=100,
    height=25)
    return True
  return False
```
Mais qu'avez vous fait à part traduire :
```
def is_game_over(self):
  if not self.current_piece.move((0,1)):
    self.play_again_btn = Button(self.root, text="Play Again", command=self.play_again)
    self.quit_btn = Button(self.root, text="Quit", command=self.quit)
    self.play_again_btn.place(x = Tetris.GAME_WIDTH + 10, y = 200, width=100, height=25)
    self.quit_btn.place(x = Tetris.GAME_WIDTH + 10, y = 300, width=100, height=25)
    return True
return False
```

Bref ! Ce projet ne tient pas sa promesse. Vous avez sans doute appris quelque chose de tkinter mais l'essentiel, la dynamique du jeu, sa conception graphique, etc. n'est pas de vous. Il aurait été plus profitable de partir de rien, quite à faire quelque chose de plus modeste. La satisfaction d'avoir construit un jeu qui fonctionne, même mal, est plus grande que celle d'avoir juste réussi à faire fonctionner le projet compliqué d'un autre.

Projet de note : 10/20
Détail de la note :

- préprojet : 1/2 (en retard, pdf initialement illisible envoyé par mail)
- compte-rendu : 2/4 (trompeur : tente de masquer/minimiser le plagia)
- Programme : 7/14 (fonctionnel, pas original (grossier plagia), compliqué, utilisation de nombreuses classes et de tkinter)

```
from tkinter import Canvas, Label, Tk, StringVar, Button, LEFT
from random import choice, randint

class GameCanvas(Canvas):
    def clean_line(self, boxes_to_delete):
        for box in boxes_to_delete:
            self.delete(box)
        self.update()

    def drop_boxes(self, boxes_to_drop):
        for box in boxes_to_drop:
            self.move(box, 0, Tetris.BOX_SIZE)
        self.update()

    def completed_lines(self, y_coords):
        cleaned_lines = 0
        y_coords = sorted(y_coords)
        for y in y_coords:
            if sum(1 for box in self.find_withtag('game') if self.coords(box)[3] == y) == \
               ((Tetris.GAME_WIDTH - 20) // Tetris.BOX_SIZE):
                self.clean_line([box
                                 for box in self.find_withtag('game')
                                 if self.coords(box)[3] == y])

                self.drop_boxes([box
                                 for box in self.find_withtag('game')
                                 if self.coords(box)[3] < y])
                cleaned_lines += 1
        return cleaned_lines

    def game_board(self):
        board = [[0] * ((Tetris.GAME_WIDTH - 20) // Tetris.BOX_SIZE)\
                 for _ in range(Tetris.GAME_HEIGHT // Tetris.BOX_SIZE)]
        for box in self.find_withtag('game'):
            x, y, _, _ = self.coords(box)
            board[int(y // Tetris.BOX_SIZE)][int(x // Tetris.BOX_SIZE)] = 1
        return board
    def boxes(self):
        return self.find_withtag('game') == self.find_withtag(fill="blue")

class Shape():
    def __init__(self, coords = None):
        if not coords:
            self.__coords = choice(Tetris.SHAPES)
        else:
            self.__coords = coords

    @property
    def coords(self):
        return self.__coords

    def rotate(self):
        self.__coords = self.__rotate()

    def rotate_directions(self):
        rotated = self.__rotate()
        directions = [(rotated[i][0] - self.__coords[i][0],
                       rotated[i][1] - self.__coords[i][1]) for i in range(len(self.__coords))]

        return directions

    @property
```

```
    def matrix(self):
        return [[1 if (j, i) in self.__coords else 0 \
                 for j in range(max(self.__coords, key=lambda x: x[0])[0] + 1)] \
                 for i in range(max(self.__coords, key=lambda x: x[1])[1] + 1)]

    def drop(self, board, offset):
        # print("\n\n\n")
        # print('\n'.join(''.join(map(str, b)) for b in board))
        # print("\n\n\n")
        off_x, off_y = offset
        # print(off_x,off_y)
        last_level = len(board) - len(self.matrix) + 1
        for level in range(off_y, last_level):
            for i in range(len(self.matrix)):
                for j in range(len(self.matrix[0])):
                    if board[level+i][off_x+j] == 1 and self.matrix[i][j] == 1:
                        return level - 1
        return last_level - 1

    def __rotate(self):
        max_x = max(self.__coords, key=lambda x:x[0])[0]
        new_original = (max_x, 0)

        rotated = [(new_original[0] - coord[1],
                    new_original[1] + coord[0]) for coord in self.__coords]

        min_x = min(rotated, key=lambda x:x[0])[0]
        min_y = min(rotated, key=lambda x:x[1])[1]
        return [(coord[0] - min_x, coord[1] - min_y) for coord in rotated]

class Piece():
    def __init__(self, canvas, start_point, shape = None):
        self.__shape = shape
        if not shape:
            self.__shape = Shape()
        self.canvas = canvas
        self.boxes = self.__create_boxes(start_point)

    @property
    def shape(self):
        return self.__shape

    def move(self, direction):
        if all(self.__can_move(self.canvas.coords(box), direction) for box in self.boxes):
            x, y = direction
            for box in self.boxes:
                self.canvas.move(box,
                                 x * Tetris.BOX_SIZE,
                                 y * Tetris.BOX_SIZE)
            return True
        return False

    def rotate(self):
        directions = self.__shape.rotate_directions()
        if all(self.__can_move(self.canvas.coords(self.boxes[i]), directions[i]) for i in
range(len(self.boxes))):
            self.__shape.rotate()
            for i in range(len(self.boxes)):
                x, y = directions[i]
                self.canvas.move(self.boxes[i],
                                 x * Tetris.BOX_SIZE,
                                 y * Tetris.BOX_SIZE)

    @property
    def offset(self):
        return (min(int(self.canvas.coords(box)[0]) // Tetris.BOX_SIZE for box in self.boxes),
                min(int(self.canvas.coords(box)[1]) // Tetris.BOX_SIZE for box in self.boxes))

    def predict_movement(self, board):
        level = self.__shape.drop(board, self.offset)
        min_y = min([self.canvas.coords(box)[1] for box in self.boxes])
        return (0, level - (min_y // Tetris.BOX_SIZE))

    def predict_drop(self, board):
        level = self.__shape.drop(board, self.offset)
        self.remove_predicts()

        min_y = min([self.canvas.coords(box)[1] for box in self.boxes])
        for box in self.boxes:
            x1, y1, x2, y2 = self.canvas.coords(box)
```

```
            box = self.canvas.create_rectangle(x1,
                                               level * Tetris.BOX_SIZE + (y1 - min_y),
                                               x2,
                                               (level + 1) * Tetris.BOX_SIZE + (y1 - min_y),
                                               fill="yellow",
                                               tags = "predict")

    def remove_predicts(self):
        for i in self.canvas.find_withtag('predict'):
            self.canvas.delete(i)
        self.canvas.update()

    def __create_boxes(self, start_point):
        boxes = []
        off_x, off_y = start_point
        for coord in self.__shape.coords:
            x, y = coord
            box = self.canvas.create_rectangle(x * Tetris.BOX_SIZE + off_x,
                                               y * Tetris.BOX_SIZE + off_y,
                                               x * Tetris.BOX_SIZE + Tetris.BOX_SIZE + off_x,
                                               y * Tetris.BOX_SIZE + Tetris.BOX_SIZE + off_y,
                                               fill="blue",
                                               tags="game")
            boxes += [box]

        return boxes

    def __can_move(self, box_coords, new_pos):
        x, y = new_pos
        x = x * Tetris.BOX_SIZE
        y = y * Tetris.BOX_SIZE
        x_left, y_up, x_right, y_down = box_coords

        overlap = set(self.canvas.find_overlapping((x_left + x_right) / 2 + x,
                                                   (y_up + y_down) / 2 + y,
                                                   (x_left + x_right) / 2 + x,
                                                   (y_up + y_down) / 2 + y))
        other_items = set(self.canvas.find_withtag('game')) - set(self.boxes)

        if y_down + y > Tetris.GAME_HEIGHT or \
           x_left + x < 0 or \
           x_right + x > Tetris.GAME_WIDTH or \
           overlap & other_items:
            # print("y_down + y > Tetris.GAME_HEIGHT : {}".format(y_down + y > Tetris.GAME_HEIGHT))
            # print("x_left + x < 0                  : {}".format(x_left + x < 0))
            # print("x_right + x > Tetris.GAME_WIDTH : {}".format(x_right + x > Tetris.GAME_WIDTH))
            # print("overlap & other_items           : {}".format(overlap & other_items))
            return False
        return True

class Tetris():
    SHAPES = ([(0, 0), (1, 0), (0, 1), (1, 1)],     # Square
              [(0, 0), (1, 0), (2, 0), (3, 0)],     # Line
              [(2, 0), (0, 1), (1, 1), (2, 1)],     # Right L
              [(0, 0), (0, 1), (1, 1), (2, 1)],     # Left L
              [(0, 1), (1, 1), (1, 0), (2, 0)],     # Right Z
              [(0, 0), (1, 0), (1, 1), (2, 1)],     # Left Z
              [(1, 0), (0, 1), (1, 1), (2, 1)])     # T

    BOX_SIZE = 20

    GAME_WIDTH = 300
    GAME_HEIGHT = 500
    GAME_START_POINT = GAME_WIDTH / 2 / BOX_SIZE * BOX_SIZE - BOX_SIZE

    def __init__(self, predictable = False):
        self._level = 1
        self._score = 0
        self._blockcount = 0
        self.speed = 500
        self.predictable = predictable

        self.root = Tk()
        self.root.geometry("500x550")
        self.root.title('Tetris')
        self.root.bind("<Key>", self.game_control)
        self.__game_canvas()
        self.__level_score_label()
        self.__next_piece_canvas()
```

```
def game_control(self, event):
    if event.char in ["a", "A", "\uf702"]:
        self.current_piece.move((-1, 0))
        self.update_predict()
    elif event.char in ["d", "D", "\uf703"]:
        self.current_piece.move((1, 0))
        self.update_predict()
    elif event.char in ["s", "S", "\uf701"]:
        self.hard_drop()
    elif event.char in ["w", "W", "\uf700"]:
        self.current_piece.rotate()
        self.update_predict()

def new_game(self):
    self.level = 1
    self.score = 0
    self.blockcount = 0
    self.speed = 500

    self.canvas.delete("all")
    self.next_canvas.delete("all")

    self.__draw_canvas_frame()
    self.__draw_next_canvas_frame()

    self.current_piece = None
    self.next_piece = None

    self.game_board = [[0] * ((Tetris.GAME_WIDTH - 20) // Tetris.BOX_SIZE)\
                       for _ in range(Tetris.GAME_HEIGHT // Tetris.BOX_SIZE)]

    self.update_piece()

def update_piece(self):
    if not self.next_piece:
        self.next_piece = Piece(self.next_canvas, (20,20))

    self.current_piece = Piece(self.canvas, (Tetris.GAME_START_POINT, 0), self.next_piece.shape)
    self.next_canvas.delete("all")
    self.__draw_next_canvas_frame()
    self.next_piece = Piece(self.next_canvas, (20,20))
    self.update_predict()

def start(self):
    self.new_game()
    self.root.after(self.speed, None)
    self.drop()
    self.root.mainloop()

def drop(self):
    if not self.current_piece.move((0,1)):
        self.current_piece.remove_predicts()
        self.completed_lines()
        self.game_board = self.canvas.game_board()
        self.update_piece()

        if self.is_game_over():
            return
        else:
            self._blockcount += 1
            self.score += 1

    self.root.after(self.speed, self.drop)

def hard_drop(self):
    self.current_piece.move(self.current_piece.predict_movement(self.game_board))

def update_predict(self):
    if self.predictable:
        self.current_piece.predict_drop(self.game_board)

def update_status(self):
    self.status_var.set(f"Level: {self.level}, Score: {self.score}")
    self.status.update()

def is_game_over(self):
    if not self.current_piece.move((0,1)):

        self.play_again_btn = Button(self.root, text="Play Again", command=self.play_again)
        self.quit_btn = Button(self.root, text="Quit", command=self.quit)
```

```
            self.play_again_btn.place(x = Tetris.GAME_WIDTH + 10, y = 200, width=100, height=25)
            self.quit_btn.place(x = Tetris.GAME_WIDTH + 10, y = 300, width=100, height=25)
            return True
        return False

    def play_again(self):
        self.play_again_btn.destroy()
        self.quit_btn.destroy()
        self.start()

    def quit(self):
        self.root.quit()

    def completed_lines(self):
        y_coords = [self.canvas.coords(box)[3] for box in self.current_piece.boxes]
        completed_line = self.canvas.completed_lines(y_coords)
        if completed_line == 1:
            self.score += 400
        elif completed_line == 2:
            self.score += 1000
        elif completed_line == 3:
            self.score += 3000
        elif completed_line >= 4:
            self.score += 12000

    def __game_canvas(self):
        self.canvas = GameCanvas(self.root,
                             width = Tetris.GAME_WIDTH,
                             height = Tetris.GAME_HEIGHT)
        self.canvas.pack(padx=5 , pady=10, side=LEFT)



    def __level_score_label(self):
        self.status_var = StringVar()
        self.status = Label(self.root,
                            textvariable=self.status_var,
                            font=("Helvetica", 10, "bold"))
        #self.status.place(x = Tetris.GAME_WIDTH + 10, y = 100, width=100, height=25)
        self.status.pack()

    def __next_piece_canvas(self):
        self.next_canvas = Canvas(self.root,
                                  width = 100,
                                  height = 100)
        self.next_canvas.pack(padx=5 , pady=10)

    def __draw_canvas_frame(self):
        self.canvas.create_line(10, 0, 10, self.GAME_HEIGHT, fill = "red", tags = "line")
        self.canvas.create_line(self.GAME_WIDTH-10, 0, self.GAME_WIDTH-10, self.GAME_HEIGHT, fill =
"red", tags = "line")
        self.canvas.create_line(10, self.GAME_HEIGHT, self.GAME_WIDTH-10, self.GAME_HEIGHT, fill =
"red", tags = "line")

    def __draw_next_canvas_frame(self):
        self.next_canvas.create_rectangle(10, 10, 90, 90, tags="frame")


    #set & get
    def __get_level(self):
        return self._level

    def __set_level(self, level):
        self.speed = 500 - (level - 1) * 25
        self._level = level
        self.update_status()

    def __get_score(self):
        return self._score

    def __set_score(self, score):
        self._score = score
        self.update_status()

    def __get_blockcount(self):
        return self._blockcount

    def __set_blockcount(self, blockcount):
        self.level = blockcount // 5 + 1
        self._blockcount = blockcount
```

```
    level = property(__get_level, __set_level)
    score = property(__get_score, __set_score)
    blockcount = property(__get_blockcount, __set_blockcount)

if __name__ == '__main__':
    game = Tetris(predictable = True)
    game.start()
```