



**Cracow University of Technology**

**Department of Computer Sciences**



**Python – Erasmus**

**Ac. Year 2023/2024**

# **Project**

# **Snakezuku**

## ***Team's Member***

Kenza Bennani

Anissa Bustarret—Labaali

Déborah Renard



# Contents

Abstract .....	3
Introduction.....	4
1. Aim .....	4
2. Scope .....	4
3. Methodology.....	4
Theoretical part.....	6
1. Brief history of Takuzu and Snake games .....	6
2. Rules of the games .....	6
Practical part .....	8
1. Interface .....	8
2. Snake .....	11
3. Takuzu .....	13
i. Condition 1 .....	13
ii. Condition 2 .....	14
iii. Condition 3 .....	15
iv. Resolution.....	16
Results .....	18
Summary .....	21
References.....	22



## Abstract

This project presents the development of two interactive games, Snake and Takuzu, using Python. We designed an easy-to-use interface that lets players choose which game they want to play.

Our goal was to develop something entertaining while integrating all the knowledge we have acquired during the semester and learning more about game development.

This report covers a general introduction which begins with an introduction outlining the aim, scope, and methodology of our project. Then, the theoretical part provides a brief history and the rules of Takuzu and Snake. Last but not least, the practical part details the development of the user interface and the implementation of both games.

# Introduction

## 1. Aim

The goal of this project is to develop an application that allows users to choose between playing Snake and Takuzu through an interactive interface. The interface not only lets users select their preferred game but also provides access to the rules for both games. For Snake, an automatic point counter is integrated to keep track of the player's score. For Takuzu, if the player is unable to solve the puzzle, they have the option to view the solution. This project aims to create an enjoyable experience by combining game development with practical programming skills.

## 2. Scope

This project covers several keys that we have used to implement the game:

- Knowledge: Crafting the rules, objectives, and overall gameplay experience for each game.
- Implementation: Writing the code using Python. Splitting the code into 3 files and focusing on creating functional and efficient functions to play the game in a graphical user interface.
- Testing and Debugging: Fix bugs, ensuring the games run smoothly and without errors.
- Documentation: Providing comprehensive documentation, including user instructions, code comments, and this project report, to facilitate understanding.

## 3. Methodology

We coded in Python directly on text files which we then executed directly via the terminal. We have split our code into 3 files : interface3.py in which the general interface and the takuzu interface are coded ; takuzu3.py in which the takuzu is coded without its interface and snake3.py in which the snake and its interface is coded.

We want to raise an important point. In the beginning, we wanted to implement only a snake game for our project, but we realized that we had enough time to implement a sort of mini games project so we decided to add the Takuzu game. This is why we have a file with the snake game and its own interface and two different files : one with the Takuzu game and another with its interface including also the main interface of the game.

For the interface we used only one library:



- tkinter : comprehensive library used for creating graphical user interfaces with buttons, menus, and other widgets. We used it to code the general interface and the interface of takuzu. We also imports the messagebox module from the tkinter library, which provides various types of message boxes for displaying information, warnings, and errors.
- The pygame library, which is a set of Python modules designed for writing video games, including graphics and sound libraries

For the snake we used several librairies :

- turtle : library focused on creating graphics and drawings by controlling a cursor (the "turtle") on a canvas. We used it to create the interface of the Snake game.
- time : provides functions for handling time-related tasks. We used it to give the player time to change the direction of the snake and to restart the game.
- random : generates random numers. We used it to generate a random food for the snake.
- Image: This imports the Image module from the PIL (Python Imaging Library) package, which allows for image processing tasks such as opening, manipulating, and saving various image file formats.

For the takuzu we used only one library :

- numpy : used for numerical computing, providing support for large, multi-dimensional arrays and matrices, along with a vast collection of mathematical functions to operate on these arrays efficiently. We used it to generate arrays.

## Theoretical part

### 1. Brief history of Takuzu and Snake games

#### **Snake Game**

The Snake game originated as an arcade game called Blockade, released in 1976 by Gremlin. It gained widespread popularity with its inclusion on Nokia mobile phones in 1997, becoming a classic. The first known coding of a digital version was in 1976 on arcade machines, and it has since been implemented on various platforms, including PCs and mobile devices. [2]

#### **Takuzu Game**

Takuzu, also known as Binary Puzzle, is a logic puzzle game involving a grid where players fill in cells with 0s and 1s according to specific rules. It does not have a clear, singular origin like Snake, but its concept is rooted in binary and logic puzzles, gaining popularity in puzzle magazines and online platforms in the early 21st century. The first coded versions appeared on puzzle websites in early 2000s and apps as its popularity grew. [3]

### 2. Rules of the games

#### **Snake**

In the Snake game, the player controls a growing line, or "snake," that moves around the screen. The objective is to eat food items that appear at random positions, which causes the snake to grow longer. The game ends if the snake collides with itself. In our case, when the user plays left and right successively only if the snake has eaten at least 2 items.

#### **Takuzu**

The goal is to fill the grid with 0s and 1s according to these rules:

- Each row and column must contain an equal number of 0s and 1s.
- No more than two of the same number can be adjacent vertically or horizontally.
- Each row and column must be unique, meaning no two rows or columns can be identical.

## Practical part

### 1. Interface

The code interface3 consists of a class MainApplication and a class TakuzuInterface.

In the first class, MainApplication, we first have the function for setting up the entry window. As soon as the first window opens, welcome music starts playing.

The player can then enter his or her name before playing.

```
class MainApplication:
    def __init__(self, root):
        self.root = root
        self.root.title("Welcome")
        self.root.geometry("400x200")

        # Initialize pygame mixer
        pygame.mixer.init()
        pygame.mixer.music.load("interfaceGame.mp3") # Replace with your interface music file
        pygame.mixer.music.play(-1) # Play the music in a loop

        # Create welcome frame
        self.welcome_frame = tk.Frame(self.root)
        self.welcome_frame.pack(pady=20)

        self.welcome_label = tk.Label(self.welcome_frame, text="Hello! Please enter your name:")
        self.welcome_label.pack(pady=5)

        self.name_entry = tk.Entry(self.welcome_frame)
        self.name_entry.pack(pady=5)

        self.continue_button = tk.Button(self.welcome_frame, text="Continue",
                                         command=self.show_main_menu)
        self.continue_button.pack(pady=10)

        # Main menu frame (initially hidden)
        self.main_menu_frame = tk.Frame(self.root)
```

Then click on "Continue" to display the main menu, which allows, with the help of buttons, to choose the game we want to play or to see the rules of the said games thanks to a messagebox.

#### → Choice number 1: "Play Snake"

Initializes and starts, with stopping music, the Snake game by creating an instance of SnakeGame and calling its run method.

```
def play_snake(self):
    pygame.mixer.music.stop()
    self.root.destroy()
    snake_game = SnakeGame()
    snake_game.run()
```

#### → Choice number 2: "Play Takuzu"

Initializes a new window, with stopping the music, the Takuzu game by creating an instance of TakuzuInterface.

```
def play_takuzu(self):
    pygame.mixer.music.stop()
    self.root.destroy()
    root = tk.Tk()
    app = TakuzuInterface(root)
    root.mainloop()
```

Regarding the second class, TakuzuInterface, similarly, we first have an `__init__` function that creates a Takuzu game window using the `create_takuzu_grid` function. This window also contains, in addition to the Takuzu grid, a button that solves the Takuzu. In this window we also have a new music starting.

```
class TakuzuInterface:
    def __init__(self, root):
        self.root = root
        self.root.title("Takuzu Grids")
        self.root.geometry("800x600")

        pygame.mixer.init()
        pygame.mixer.music.load("takuzu.mp3")
        pygame.mixer.music.play(-1)

        self.initial_grid = [
            [None, 1, None, 0],
            [None, None, 0, None],
            [None, 0, None, None],
            [1, 1, None, 0]
        ]

        self.grid = self.create_takuzu_grid(0, 0, self.initial_grid)

        self.solve_button1 = tk.Button(self.root, text="Solve Grid", command=lambda:
self.solve_grid(self.initial_grid))
        self.solve_button1.grid(row=1, column=0, pady=10)

        self.back_button = tk.Button(self.root, text="Back to Main Menu", command=self.go_back)
        self.back_button.grid(row=2, column=0, pady=10)
```

### create takuzu grid:

This function takes the initial grid and transforms each cell with a None value into an editable cell. Thus, the player can attempt to solve the Takuzu on this main window.

```
def create_takuzu_grid(self, row, column, initial_values=None):
    frame = tk.Frame(self.root, width=200, height=200)
    frame.grid(row=row, column=column, padx=10, pady=10)
    entries = []
    for i in range(4):
        row_entries = []
        for j in range(4):
            value = initial_values[i][j] if initial_values else None
            cell = tk.Entry(frame, width=3, font=("Helvetica", 16), justify="center")
            cell.grid(row=i, column=j, padx=5, pady=5)
            cell.insert(tk.END, value if value is not None else '')
            row_entries.append(cell)
        entries.append(row_entries)
    return entries
```

### solve grid:

If the "Solve Grid" button is pressed, then the `solve_grid` function comes into play. It uses three functions from the Takuzu3 file:

- `non_to_deux`: transforms the None values in the initial grid into twos.
- `Résolution_`: which solves the Takuzu.
- `get_final`: retrieves the new solved grid.



Once these three functions are finished, the new solved grid is displayed using the `show_additional_grid` function.

```
def solve_grid(self, grid_entries):
    grid = none_to_deux(grid_entries)
    Résolution(grid)
    solved_grid = get_final(grid)
    self.show_additional_grid(solved_grid)
```

### show\_additional\_grid:

This function creates a new window showing the solution grid (non-editable). The player can compare their grid with the result grid.

```
def show_additional_grid(self, solved_grid):
    additional_grid = tk.Toplevel()
    additional_grid.title("Solved Takuzu Grid")
    additional_grid.geometry("400x400")

    frame = tk.Frame(additional_grid, width=200, height=200)
    frame.pack(padx=50, pady=50)

    for i in range(4):
        for j in range(4):
            value = solved_grid[i][j]
            label = tk.Label(frame, text=value, width=3, font=("Helvetica", 16),
                             justify="center", borderwidth=2, relief="solid")
            label.grid(row=i, column=j, padx=5, pady=5)
```

### go back:

This function simply takes you back to the main menu, with the old music of course.

```
def go_back(self):
    pygame.mixer.music.stop()
    self.root.destroy()
    root = tk.Tk()
    app = MainApplication(root)
    root.mainloop()
```

Finally,

```
if __name__ == "__main__":
    root = tk.Tk()
    app = MainApplication(root)
    root.mainloop()
)
```

Initializes the main tkinter root window and starts the `MainApplication`. Runs the tkinter main loop to keep the application running.

## 2. Snake

For the implementation of the game, we have been inspired by a gitHub code [1]. But we have modified almost everything, and we have added some amusing functionalities such as images and music. Indeed, we have used Pygame library for background music.

The SnakeGame class is responsible for initializing the game, handling user inputs, updating the game state, and managing game elements such as the snake and the food. The main components and methods of this class are described below.

This part of the code is used to replace the head of the snake by a real image of a snake's head and a real image of the food's snake (here an apple).

```
# Open an existing image
original_image = Image.open("apple.gif")

# Resize the image
resized_image = original_image.resize((20, 20), Image.ANTIALIAS)

# Save the resized image
resized_image.save("apple2.gif")

# Open an existing image of the snake head
original_head_image = Image.open("serpent.gif")
# Resize the snake head image
resized_head_image = original_head_image.resize((20, 20), Image.ANTIALIAS)
# Save the resized snake head image
resized_head_image.save("serpent2.gif")
```

The `__init__` method sets up the game environment, initializes variables, loads background music, and creates the initial game objects.

The methods `go_up`, `go_down`, `go_left`, and `go_right` change the direction of the snake based on keyboard inputs. These methods are bound to specific keys using the `onkeypress` method.

```
def go_up(self):
    self.head.direction = "up"

def go_down(self):
    self.head.direction = "down"

def go_left(self):
    self.head.direction = "left"

def go_right(self):
    self.head.direction = "right"
```

The 'move' method updates the position of the snake's head based on its current direction. It also handles the screen wrap-around feature, ensuring the snake reappears on the opposite side if it crosses the border.

```
# Wrap the snake around if it crosses the border
if self.head.xcor() > 290:
    self.head.setx(-290)
if self.head.xcor() < -290:
    self.head.setx(290)
if self.head.ycor() > 290:
    self.head.sety(-290)
if self.head.ycor() < -290:
    self.head.sety(290)
```

The `play_game` method is the main game loop. It continuously updates the game state, checks for collisions, moves the snake, updates the score, and redraws the game screen.

The `change_food_position` method changes the position of the food to a new random location on the screen, ensuring it does not overlap with the snake's body.

```
def change_food_position(self):
    while True:
        x = random.randint(-290, 290)
        y = random.randint(-290, 290)
        if (x, y) not in [(segment.xcor(), segment.ycor()) for segment in self.segments]:
            break
        self.food.goto(x, y)
```

The `add_segment` method adds a new segment to the snake's body, which occurs when the snake eats the food.

```
def add_segment(self):
    new_segment = turtle.Turtle()
    new_segment.speed(0)
    new_segment.shape("square")
    new_segment.color("grey")
    new_segment.penup()
    self.segments.append(new_segment)
```

The `reset_game` method resets the game state when the snake collides with itself. It clears the snake's body, resets the score, and repositions the head.

```
def reset_game(self):
    time.sleep(1)
    self.head.goto(0, 0)
    self.head.direction = "stop"
    for segment in self.segments:
        segment.goto(1000, 1000)
    self.segments.clear()
    self.score = 0
    self.delay = 0.20
    self.pen.clear()
    self.pen.write("Score: {} High Score: {}".format(self.score, self.high_score),
gn="center",
                  font=("Courier", 24, "normal"))
```

The `run` method starts the game by calling the `play_game` method.

```
def run(self):
    self.play_game()
```

The main function initializes an instance of the `SnakeGame` class and calls its `run` method to start the game.

```
if __name__ == "__main__":
    game = SnakeGame()
    game.run()
```

### 3. Takuzu

#### i. Condition 1

As a reminder, condition 1 concerns the number of 1s and 0s allowed next to each other in a row.

```
import numpy as np

# Complete line verification:

# Condition1: there cannot be more than two consecutive 1s or 0s
def condition1_colonne(colonne, a, v): # we check if in the entire grid condition 1 is not validated in the columns
    k = 0
    colonne[a] = v # we will test if by changing this cell, the grid is valid
    while k < (len(colonne) - 3): # k scans the column
        if colonne[k] != 2 and colonne[k] == colonne[k+1] == colonne[k+2]: # if the value is 2 then it doesn't matter
            #but if it is 1 or 0 and we have the same value three times in a row, we return false
            return False
        else: # otherwise, we continue
            k += 1
    return True # if neither 1 nor 0 are consecutive, we return True
```

This function is a boolean function which takes as arguments a column in the form of a list, a value for the cell to be changed and the index of the cell in this list. First we change the cell in question by the value and check whether the condition will be validated with this change. The function allows us to scan the column to see if it contains more than two 1s or two 0s next to each other in a row. We vary a variable k up to the size of line-3 so that we are not out of the table when we look at the k-th, k+1st and k+2nd cell.

```
def condition1_ligne(ligne, b, valeur): # we check if in the entire grid condition 1 is not validated in the rows
    k = 0
    ligne[b] = valeur # we will test if by changing this cell, the grid is valid
    while k < (len(ligne) - 3): # k scans the row
        if ligne[k] != 2 and ligne[k] == ligne[k+1] == ligne[k+2]: # if the value is 2 then it doesn't matter
            #but if it is 1 or 0 and we have the same value three times in a row, we return false
            return False
        else: # otherwise, we continue
            k += 1
    return True # if neither 1 nor 0 are consecutive, we return True
```

This is the same functions but for lines.

## ii. Condition 2

Condition 2 ensures that there are as many 1s and 0s in the same row or column

```
# Condition 2: Rows and columns must contain the same number of 0s and 1s
def somme_colonne(colonne, x, valeur): # the goal is to count the number of 1s and 0s to see
# if we will end up with an equal number in each column
    nbr1, nbr0 = 0, 0 # number of 1s and number of 0s
    colonne[x] = valeur # we will test if by changing this cell, the grid is valid
    for k in range(len(colonne)): # we vary the rows so we scan the column x
        if colonne[k] == 1: # we count the number of 1s
            nbr1 += 1
        if colonne[k] == 0: # we count the number of 0s
            nbr0 += 1
    if nbr1 > (len(colonne)) / 2 or nbr0 > (len(colonne)) / 2: # if the number of one implies that in the end the number
# will not be equal, we return false
        return False
    return True # otherwise, we return true
```

As in the previous condition, this is a boolean function and we will change the selected cell to a given value to test whether the condition is validated with this change. In this condition we initialise 2 counters which will count respectively, nbr1, the number of 1s and nbr0 the number of 0s. However, if more than half of the cells in a column contain 1s (or 0s) then the number of 1s (or 0s) are different, the function returns false which means that the grid does not meet the takuzu conditions.

```
def somme_ligne(ligne, y, valeur): # the goal is to count the number of 1s and 0s to see if we will end up with
# an equal number in each row
    ligne[y] = valeur # we will test if by changing this cell, the grid is valid
    nbr1, nbr0 = 0, 0 # number of 1s and number of 0s
    for k in range(len(ligne)): # we vary the columns so we scan the row y
        if ligne[k] == 1: # we count the number of 1s
            nbr1 += 1
        if ligne[k] == 0: # we count the number of 0s
            nbr0 += 1
    if nbr1 > (len(ligne)) / 2 or nbr0 > (len(ligne)) / 2: # if the number of one implies that in the end the number
# will not be equal, we return false
        return False
    return True
```

We do the same for the lines.

### iii. Condition 3

Condition 3 allows us to check that the rows and columns are different from each other.

```
# Condition 3: Rows and columns must be different from each other
def Condition3(grille):
    liste_ligne, liste_colonne, L = [], [], []
    c, b, k, l = 0, 0, 0, 0
    y, x = 0, 0 # we create the coordinates

    # we will first create a list of type list of rows with only complete rows
    while y < len(grille): # we fix the row
        for x in range(len(grille)): # we vary the x to scan the row
            coord = (x, y)
            L.append(grille[y][x])
            liste_ligne.append(L)
            L = [] # we reset our temporary list
            y += 1 # we then change the row

    y, x = 0, 0 # we reset our coordinates to (0,0)
    while x < len(grille): # similarly we fix the column
        for y in range(len(grille)): # we vary the y to scan the column
            coord = (x, y)
            L.append(grille[y][x])
            liste_colonne.append(L)
            L = [] # we reset our temporary list
            x += 1 # we then change the column
```

To do this, we first create a list (ligne\_liste) which will contain all the lines in the grid. For each y, we vary the x so we can add all the cells of the line in the list. When a line is finished, we move on to the next one by adding 1 to y and so on. Finally, we reset our coordinates to (0,0) and then we do the same for the columns.

We do the same for the columns with liste\_colonne. But this time we vary the y for each x.

```
# Then we compare each row with each other with the previously created list of rows
while c < len(liste_ligne) and liste_ligne[c] not in (liste_ligne[:c] + liste_ligne[c+1:]): # If the list c in the list
# of rows is not in the list of rows without the list c then we move to c+1
    c += 1

# finally we compare each column with each other
while b < len(liste_colonne) and liste_colonne[b] not in (liste_colonne[:b] + liste_colonne[b+1:]): # If the list b
# in the list of columns is not in the list of columns without the list b then we move to b+1
    b += 1

# if b = len(liste_colonne) then no column is similar, likewise for c and the list_ligne
return b == len(liste_colonne) and c == len(liste_ligne)
```

This part of the program allows us to compare the elements of liste\_ligne (or liste\_colonne), which is equivalent to comparing lines with each other (or columns). We have introduced a variable c in order to identify which element of the list is being compared with the others. We compare liste\_ligne[c] with the elements before itself in liste\_ligne (liste\_ligne[:c]) and then with the elements after itself (liste\_ligne[c+1:]). In this way, we don't compare element c with itself, so we don't have to display the equality each time.

Then we do the same procedure with liste\_colonne with variable b.

#### iv. Resolution

The resolution function is the final function which, using the recursive backtracking method, will test the values in the grid and see if they meet the various conditions. If they don't, the program returns to the last completed cell to modify it and retest the conditions, and so on to find the correct values.

```
def Résolution(grille, i=0):
    cote = len(grille) # we get the side length of the grid

    if i == cote * cote: # if i equals the dimension then we have filled the entire grid and we then test the last condition
        get_final(grille)

        return True if Condition3(grille) else False

    # we now create the coordinates x (the column) and y (the row)
    x = i % cote # x takes the value of the remainder of the Euclidean division of i by the side, for example: if i = 4
    # then x will be equal to 1, so we are on the second column
    y = i // cote # y takes the value of the Euclidean division of i by the side, for example: if i = 4
    # then y = 0, so we are on the first row

    if grille[y][x] != 2: # if the value is not 2 then we move to the next cell
        return Résolution(grille, i + 1)

    for j in range(2): # j takes the value 0 then will take 1
        ligne = grille[y] # we get the row at index y
        colonne = [L[x] for L in grille] # we transform column x into a list
        if condition1_colonne(colonne, y, j) and condition1_ligne(ligne, x, j) and somme_colonne(colonne, y, j) and somme_ligne(ligne, x, j):
            # if condition 1 and condition 2 are met by changing the cell to j then the cell takes the value j
            grille[y][x] = j
            if Résolution(grille, i + 1): # if it is validated for the next cell we return true
                return True
        grille[y][x] = 2
    return False # otherwise the grid cannot be filled
```

First, we assign the variable 'cote' the length of one side of the grid. Then, we perform several tests.

First, if 'i' equals the dimension of the grid, it means we have reached the end of the grid. We then return True only if Condition 3 is met.

Otherwise, we assign to 'x' (which represents the columns here) the value of the remainder of the Euclidean division of 'i' by the side, and to 'y' (which represents the rows) the value of the quotient of the Euclidean division of 'i' by the side. These assignments ensure that we never go outside the grid.

If the cell is different from 2, then we do not modify it because it contains the base grid that we need to complete, and we thus call the resolution function for the next cell.

If the cell is a 2, we will vary a 'j' between 0 and 2 (exclusive), which represents the value that we will test in the cell. We then initialize our row and column and test the first three conditions written above with this value 'j'. If the three conditions are met, we assign the value 'j' to the cell and move on to the resolution of the next cell.

If the value 'j' does not meet the conditions, we put a 2 back in the cell while the for loop changes the 'j' to see if the problem lies with the current cell or the previous one.

Finally, if none of these cases match, the grid is not fillable and False is returned.

```
def none_to_deux(grille): #this function allows to change the nones in 2s
    for i in range(len(grille)):
        for j in range(len(grille[i])):
            if grille[i][j] is None:
                grille[i][j] = 2
    return grille
```



It is easier to use Nones in the code of the interface but the code of the Takuzu uses twos when the cell is empty. We coded the Takuzu before the interface (the interface was not planned at the beginning) so to avoid changing all the code of the Takuzu we created this function.

```
def get_final(grille): #function which returns the grid  
    return grille
```

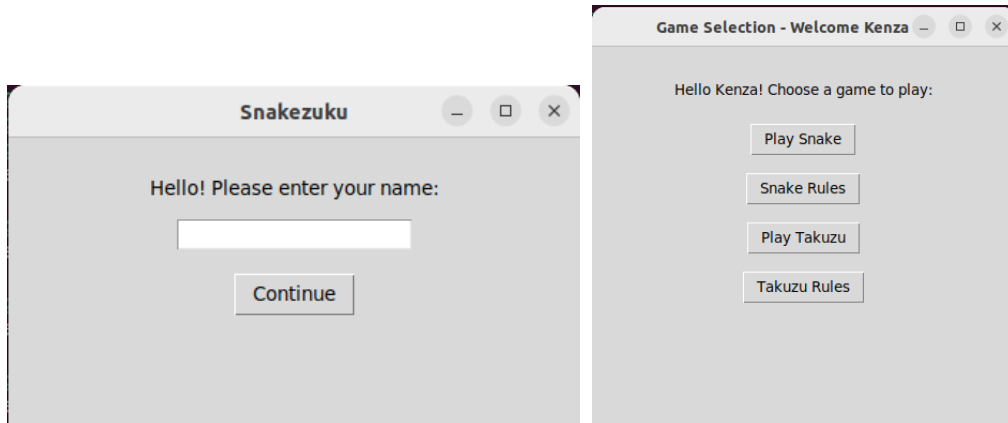
This a function which returns the grid to use for the code of the interface.



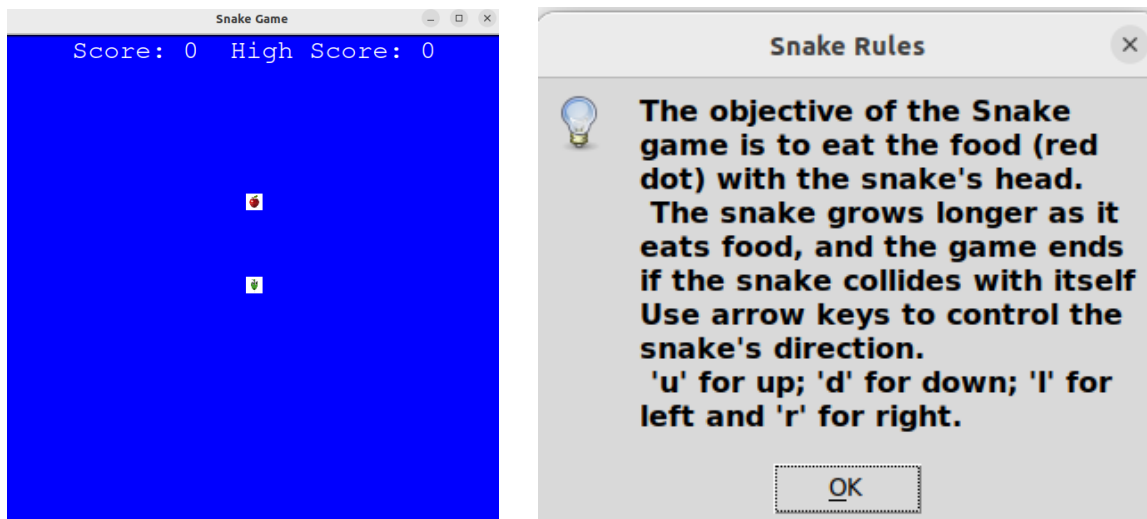
## Results

Finishing the implementation, we run our code on the terminal.

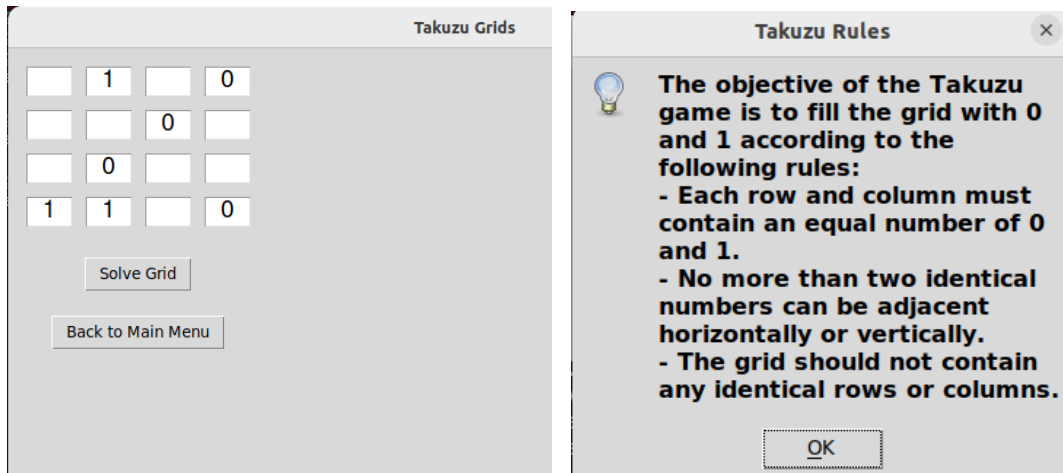
First, we will have a frame that enables the user to enter his name and then choose if he wants to play some games or read their rules.



If the users choose to play the snake game, here is the frame that they will get. With some apple images for feeding the snake and amusing music. And, to the right if the users choose to know more about the Snake Rules.



It is quite the same if the users choose to play Takuzu Game and read the rules.



But here the user can go back to the main Menu and restart the game, which is very useful.



## Summary

To conclude, this project enabled us to apply the knowledge we had acquired throughout the semester by developing an application where users can choose to play either Snake or Takuzu. The application features a user-friendly graphical interface and incorporates music within the games, enhancing the overall user experience. This project not only solidified our understanding of Python programming but also provided practical experience in game development, GUI design, and multimedia integration.

This project could also serve as the initial version of a more extensive mini-games application. The aim is to eventually expand this application to include a variety of other mini games, transforming it into a general-purpose gaming platform. By starting with Snake and Takuzu, we have established a solid foundation and framework that can be built upon to add more games in the future.



## References

[1] Snake Game created by @TokyoEdTech

<https://gist.github.com/wynand1004/ec105fd2f457b10d971c09586ec44900>

[2] Snake Game

[https://fr.wikipedia.org/wiki/Snake\\_\(genre\\_de\\_jeu\\_vid%C3%A9o\)](https://fr.wikipedia.org/wiki/Snake_(genre_de_jeu_vid%C3%A9o))

[3] Takuzu Game

<https://fr.wikipedia.org/wiki/Takuzu>

[4] The musics in the games are from this website

<https://downloads.khinsider.com/game-soundtracks/album/dr.-kobushis-labyrinthine-laboratory-linux-macos-windows-gamerip-2022>