Introduction to programming: Final project

For the final project, you must develop several games for the Sense Hat. You can test your code using [https://trinket.io/sense-hat](https://trinket.io/sense-hat). Be careful not to make big changes to your project in the browser without saving them!

Criteria

You must submit each part of the project before the deadline as a .py file. You must be able to run your project on a Raspberry Pi (even if you wrote your project using your laptop).

For each part, you can get:

- The number of points indicated in the parentheses for each successfully completed task. If you need help with a particular task, you can ask us for a hint. You can also ask us to give you a solution for a particular task if you can't solve it but need it to finish the other tasks.
- 1 point for clarity of the code (PEP 8 standard, structure of the files, docstrings, naming of the variables, comments that explain what your code does …)

At the defense, we will have two evaluation elements:

- First, we will ask you to explain what different parts of your code do. For this evaluation element, you can get the maximum of 6 points (2 points for each part).
- Second, we will ask you to do a set of programming exercises to check your knowledge of the curriculum. For this evaluation element, you can get the maximum of 10 points.

In total, you can get the maximum of 45 points that we will normalize into a final note from 1 to 20.

Description of the project

Part 1: Flappy bird

In this part, you will develop a clone of the [Flappy Bird](https://trinket.io/sense-hat) game. In this task, it will be much more convenient for you to represent the LED matrix row by row, i.e. as a list screen of 8 lists of length 8 each, because then you can address a pixel with coordinates (x, y) as screen[x][y]. However, the sense_hat module does not know how to work with 2-dimensional lists. Here is your first task:

**Task 1.** (1 point) Write a function `flatten` that takes a list of 8 lists representing the rows of the matrix and returns a 1-dimensional list of length 8 * 8 representing the LED matrix.

In Flappy Bird, the bird will have to avoid “pipes” that sprout from the top and bottom of the matrix. Your second task is to generate the pipes.

**Task 2.** (1 point) Write a function `generate_pipe` that generates two pipes for the 8th row of the LED matrix. The gap between the two pipes must be equal to three (see the image below). The pipes and the background can be of any color of your choice. A hint: it is much easier to choose the center of the gap at random and fill all the cells except the center of the gap and two neighboring cells.
We pretend that the bird flies forward. There are two ways to do: either we move the bird, or we move the pipes. You will now write a function that does so.

**Task 3.** (1 point) Write a function `update`. It must take two parameters: the list screen and the time $t$ that has passed from the beginning of the game. At any time, you must shift all the rows up by one (that is, row 1 becomes row 0, row 2 becomes row 1, etc). Additionally, if the time $t \% 3 = 0$, you must generate pipes for row 7 (we want to have pipes in every third row only).

**Task 4.** (1 point) Write a function `collision`. It receives the coordinates of the bird and the LED matrix and returns True if the bird hit a pipe and False otherwise.

**Task 5.** (1 point) Add a function `game_over`. It receives two arguments: the LED matrix and the score. When called, it must display a message GAME OVER and the number of points gained (equal to the time passed from the beginning of the game). Hint: Look at Task 7 and think what you need to pass to the `game_over` function as arguments!

**Task 6.** (1 point) Write a function `game_starts`. It must display a message GAME STARTS and then numbers 3.. 2.. 1..

Now it is the time to assemble these functions into the game!

**Task 7.** (2 points) Write a function `flappy_bird`. This will be the main function of this part. Initialize the coordinates of the bird as $x = y = 0$ and color the corresponding pixel into a color of your choice. Call the function `game_starts`. Add a cycle while. In this cycle, you must do three things: check if the joystick has been moved and update the position of the bird accordingly, update the LED matrix. Only move your bird to the right or to the left, and be careful to check if it is on the side of the LED matrix, in which case don't move it. If the bird hits a pipe, stop a game and show the score using the `game_over` function. Don't forget to add `sleep(1)` at the end of the while cycle so that the game is not too fast..

The movements of the bird will be a bit delayed (like when you are clicking on a button and it is responding with a delay because a program is running slowly), but it's OK, don't worry!

**Part 2: Snake**

In the second part, you will be developing a clone of the Snake game. Your snake will be specified by a list of lists of length 2 and the direction of its movement. At each moment of time, the snake
advances one pixel in the direction of its movement. If there is a border of the LED matrix on its way, the snake continues its movement on the other side of the matrix.

The direction can be changed by moving the joystick. When the snake “eats” a fruit (yellow and orange pixels in the figure), which means that its tail grows by one pixel, the fruit disappears and the score increases by one. We will now code the game step by step.

Task 1. (1 point) Write a function `ate_fruit`. It must take two arguments: the snake and the list of fruits. If the head of the snake is at the position of one of the fruits, the function must return True and delete the fruit and otherwise False.

Task 2. (1 point) Write a function `collision`. It must check if the snake intersects with itself (contains a duplicate pixel). If it does, return True, otherwise False.

Task 3. (1 point) Write a function `generate_fruits`. It must take the snake and the list of fruits as an argument. If there are less than three fruits, generate a new one at random. If its position is not occupied by the snake, append the fruit to the list, otherwise discard it. Return the resulting list of fruits.

Task 4. (1 point) Write a function `move`. It must take four arguments: the snake, the direction, the list of fruits, and the score. First, add a new head for the snake. Second, check if the snake ate a fruit. If it did not, delete the tail (the last pixel) of the snake. Finally, check if the snake intersects with itself. If it does, call the function `game_over` (re-use the function from Part 1). The function must return lists snake and fruits, and the score.

Task 5. (1 point) Write a function `update`. It must take three arguments: the LED matrix, the snake, the fruits, and recolor the LED matrix accordingly.

Task 6. (2 points) Write a function `snake`. This will be the main function of this part. Initialize your snake by a list of three lists of length 2 corresponding to the first three pixels in the zeroth row. Initialize the fruit list as an empty list. Call the function `game_starts` (re-use the function from Part 1). Add a cycle while. In this cycle, you must do the following: check if the joystick was moved and update the direction of the snake's movement accordingly, move the snake (call the function `move`), update the fruit list, and update the LED matrix. Don't allow to change the direction from left to right (and vice versa) or from up to down (and vice versa). If the snake bites into itself, stop the game and call the function `game_over`. Recall that the score is equal to the number of fruits eaten by the snake! Don't forget to add sleep(1) at the end of the while cycle so that the game is not too fast.
Introduction to programming: Final project

**Part 3: Marble maze**

In the third part, you will develop a clone of the Marble maze game. The game is simple: you have a maze with a sink and a marble ball. Your goal is to move the maze so as the ball ends up in the sink. On the figure below, the walls of the maze are red, the marble is white, and the sink is green.

![Marble maze game](image)

**Task 1.** (1 point) Write a function `generate_maze`. The function receives no arguments. This task has two versions: an easier one and an a harder one, you can choose any of them up to your taste. Easier version: create three mazes, and return one of them at random. Harder version: create a maze at random. A hint here is that you can represent all pixels that are not a part of the maze as a graph. Your task is to ensure that the graph is connected, otherwise, the user will not be able to win. The function must return a list representing the LED matrix.

**Task 2.** (1 point) Write a function `generate_sink`. It must receive one argument, a list representing the maze, and return a position of the sink (chosen at random not to coincide with any of the walls).

**Task 3.** (1 point) Write a function `generate_marble`. It must receive two arguments, a list representing the maze, and the position of a sink, and return a position of the marble (chosen at random not to coincide with any of the walls or the sink).

**Task 4.** (3 points) Add a function `move_marble`. It receives four arguments, pitch, roll, marble, maze and returns the new position of the marble. The pitch and the roll take values between 0 and 359. If there is a wall on the way of the marble, it should not move. Making the movements of the marble realistic is not that easy. Here is a suggestion:

- If $10 < \text{pitch} < 170$, the y-coordinate of your marble can decrease by one, and if $190 < \text{pitch} < 350$, the y-coordinate of your marble can increase by one (of course, only if this does not make the marble to leave the borders of the LED matrix).
- If $10 < \text{roll} < 170$, the x-coordinate of your marble can decrease by one, and if $190 < \text{roll} < 350$, the x-coordinate of your marble can increase by one (of course, only if this does not make the marble to leave the borders of the LED matrix).

You now need to decide where your marble goes. For example, let $[x, y]$ be the old coordinates of the marble. Suppose that $x$ can increase by one (become $x + 1$), and $y$ can decrease by one (become $y - 1$). Then you have three choices for the new position of the marble: $[x, y + 1]$, $[x + 1, y]$, and $[x + 1, y + 1]$. Check each of them in order. If there is no wall at this position, move the marble there with probability 0.4.

Hint 1: If you need to create a copy of a list `my_list`, use `list1 = list(my_list)` (if you use `list1 = my_list`, all changes to `list1` will be repeated in `my_list` as well).
Introduction to programming: Final project

Hint 2: To change the coordinates with probability 0.4, generate a random float number in the interval \([0, 1)\). If this number > 0.6, then change the coordinates.

Hint 3: You can come up with your own solution. The main thing is for the movements to be realistic!

Task 5. (1 point) Add a function \texttt{check\_sink}. It receives two arguments: marble and sink. If their positions coincide, the function returns True and otherwise False.

Task 6. (1 point) Add a function \texttt{redraw}. It must receive four arguments: the LED matrix, the maze, the marble, and the sink, and re-draw them. You can re-use the function \texttt{flatten} that you wrote for the first part.

Task 7. (1 point) Add a function \texttt{game\_over}. When called, if the time < 200, it must display a message GAME OVER. YOU WIN and otherwise GAME OVER. YOU LOST.

Task 8. (2 points) Add a function \texttt{maze}. This will be the main function of this part. Initialize your maze by a list of eight lists of length 8, the marble as a list of length 2, and the sink is a list of length 2 as well. Initialize time \(t = 0\). Call the function \texttt{game\_starts} (re-use the function from Part 1). Add a cycle while. In this cycle, you must do the following: check if you need to move the marble, and if the marble is in the sink. Add \texttt{sleep(0.05)} at the end of the while cycle (note that the pause is different!)