Snake

◎ Work tips

- the point is to do personal work, even if you work together and discuss
- all useful notions will be given in class or in the document
- read the questions, most the answers are in them
- ask questions 😄

Context

The game of **snake** is one of the most popular games in the world[1]. The point of the game is to control a little snake. The snake can move in a given game area and has to eat little apples that are placed randomly in the area. Each apple the snake eats makes him grow, making the snake harder to control. The game ends when the snake's head exits the zone, touches another part of the snake, or if the snake fills the whole game area.

Idea

Do not forget that the code template gives you a lot of information on where to write the elements for the different steps. Use that to your advantage.

Game area organisation

A **position** in the game area is defined by a couple of numbers that indicate its distance to the top-left corner `(0, 0)`. The first coordinate is over the **width** of the window, the second is over the **height**.

We define a **unit** as a number of pixels that will allow to divide the game area in a **grid**. The first column (or row) is `0`, the last is `max - 1`.

Every **game element** is composed of a square of **unit** size, identified by the position of its top-left corner, corresponding to its row and column.

The snake will start in the **middle** of the game area.

The snake is composed of multiple squares of the same colour. When it moves, the head moves in a given direction and the last square disappears.

In fig. 1, you can see an example of a `30×20` game area. The snake is `5` squares long at the moment and the current apple is at coordinates `(14, 5)`. Of course, you should not display the coordinates on the actual game screen, this is just so you can imagine the system.

[1] at least in the '80s, don't judge

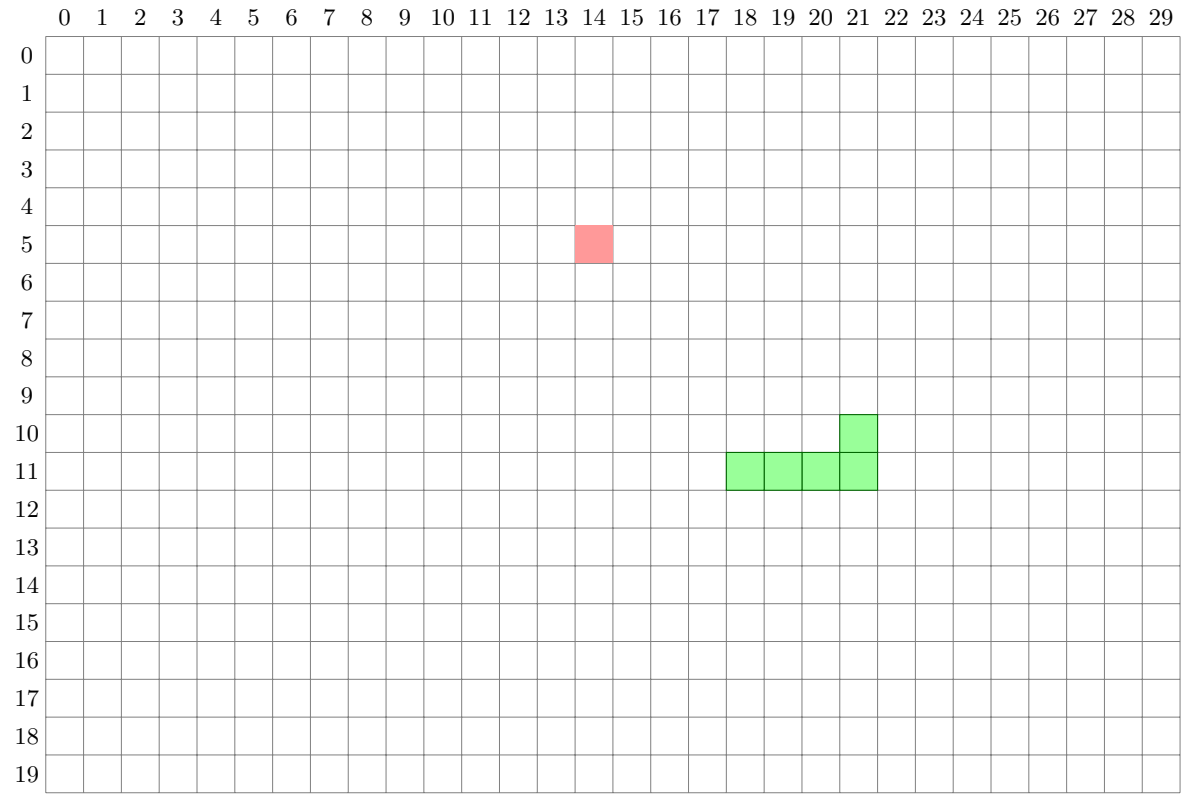

Figure 1: Example of a grid

Step 1. Defining the variables

As always, we have to define a couple of variables to set up the game area.

You will need:

- a **unit** size, set at **10** for the moment
- a **area_width**, set at **60** units
- a **area_height**, set at **40** units
- a **food_colour**
- a **snake_colour**
- a **background_colour**

Step 2. Preparing the zone

In this step, we prepare the game area to be displayed.

1. update the instruction to create the window, so that it uses the **area_width** and **area_height** variables
2. add a background colour to the window

3. write the body of the **draw_square** function. This function takes 3 parameters `x, y, colour`. It will draw a square of **unit** size at the grid at position (x, y).

Step 3. Randomness

To add the food at random points in the grid, we have to be able to generate random positions.

1. write a **random** function that takes a parameter **max** and returns a random integer between 0 and max (excluded).
2. write a **random_position** function that returns a **valid** position (a couple of coordinates) in the game area. You should use the first random function you defined of course.

Idea

A **coordinate** will be represented by a couple (tuple) of integers separated by commas inside of brackets. ex: `(14, 5)`

Step 4. Showing multiple squares

Write a function named **draw_square_list**, that takes as parameters:

- a list of coordinates
- a colour

and draws for each coordinate in the list a square by calling **draw_square** correctly. All the squares in the list should be in the same colour.

Idea

You can access elements in a tuple in the same way as a list.

Code

```
coord = (5, 6)
print(coord[0]) # 5
print(coord[1]) # 6
```

Step 5. Game elements variables

These variables will allow us to store useful game information.

You will need:

- the **snake_positions**: the list of coordinates for the squares composing the snake. At the start, the snake is one square in the middle of the game area.
- a boolean **game_end** to know if the game is finished. Of course, it's `False` at the start.
- the current direction **snake_direction**, which will be an empty string at the start.
- coordinates for the food **food_position**, which will be chosen by calling the relevant function.

Step 6. A game turn

We will now write the actions that occur each turn.

1. write the **call** to the **draw_square** function so that food appears correctly
2. write the **call** to the **print_message** function to show the current score at the top left of the game area. The current score is computed as the length of the snake minus 1.
3. write the **call** to the **draw_square_list** function to draw the snake
4. use the **game_end** variable for the loop's condition

Step 7. Direction(s)

The snake has to be able to move in multiple directions. Write a function **direction**, that takes two parameters: the pressed key and the current snake direction.

If the pressed key is:

- `UP`, then the x-coordinate does not change, and y goes up by 1
- `DOWN`, then the x-coordinate does not change, and y goes down by 1
- `LEFT`, then x goes down by 1 and y does not change
- `RIGHT`, then x goes up by 1 and y does not change

The function will **return** a couple of numbers that represent the changes in x and y. If another key is pressed / something else, nothing changes.

Step 8. Reacting to an event

Two events may occur during the game:

1. the user presses the exit button, in that case the game is over. A variable has to change.
2. the user presses a key: then the current direction might change. Use a function you have defined earlier.

Idea

`event.type` indicates the type of an event

Step 9. Computing the new position

Write a **new_position** function that takes as a parameter the current direction and the snake. It returns the coordinate that is the new position for the head of the snake.

You compute the new coordinate from the direction. You can add the new position at the end of the list with `append`; the head will therefore be the last position in the list each time.

Step 10. Position in the game area

Write a **in_zone** function that takes as a parameter a position. It then returns `True` if the position is inside the game area, and `False` if not.

Idea

You should use the variables that define the window, of course

Step 11. Snake progression

Compute the next position for the snake's head. If the position is inside the game area, it is added to the snake. If not, the game is over.

If the snake's head is on the food, you have to define a new position for food. If not, then the tail of the snake is removed from the list.