ECE 122: Introduction to Programming for ECE- Spring 2023

Project 3: Fun with Games (OOP and Graphics)

Due Date: **See website, class policy and moodle for submission**

Description

The goal of this project is to practice OOP using multiple classes, instance variables, methods (instance and static), inheritance, and some encapsulation. You will also operate Graphics to develop multiple games.

Your project includes multiples files:
`Pixel.py`, `Grid.py`, `Snake.py`, `Tetrominoes.py`, and `Tetris.py`

the figure below represents the inter-relationships between all classes and applications:

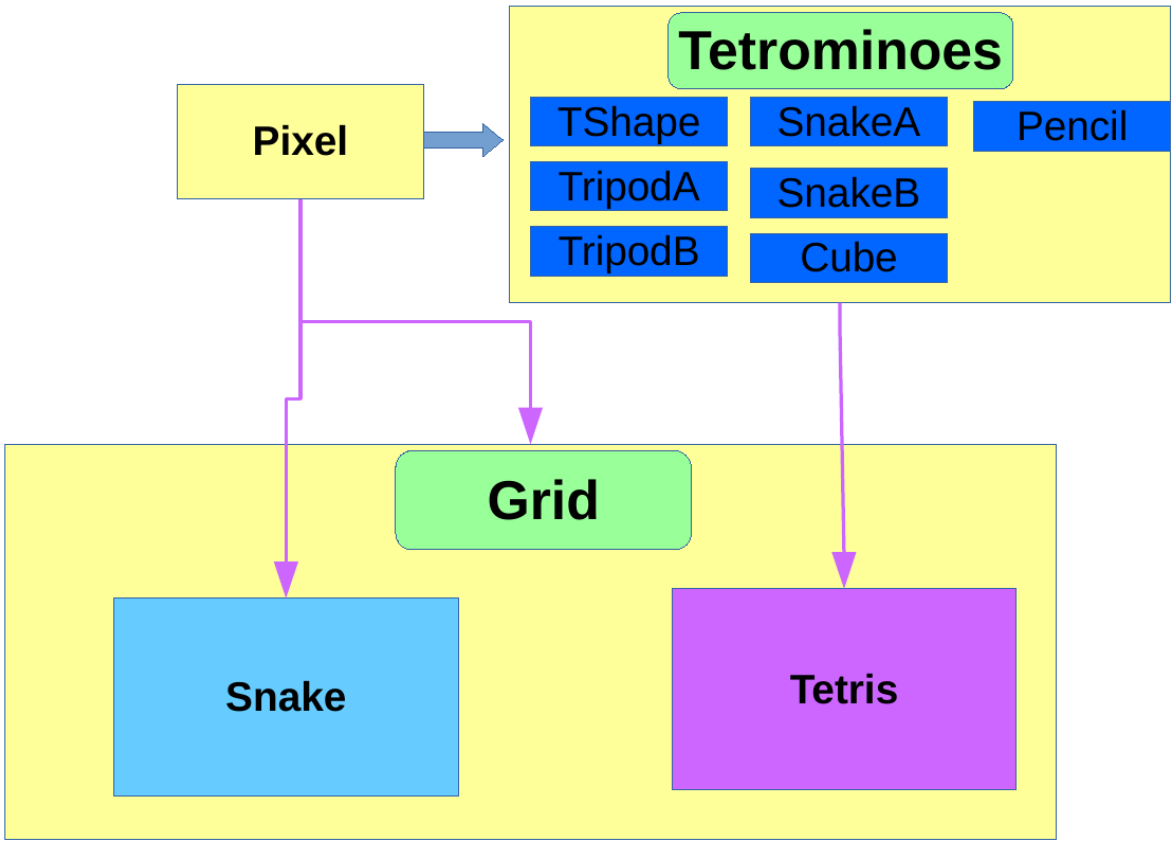

Figure 1: `Relationships between classes and Applications.`

Each class can be tested separately (they have their own main method and multiple test functions).

How to start

Once Pixel is implemented, you can implement Grid and Tetrominoes (any order). Once Grid is implemented, you can work on Snake (first game). If you implement both Grid and Tetrominoes, you can work on Tetris (second game). I would suggest to follow the order of the tasks that are presented below. There are a lot of things to complete and test, be patient and persistent so you can enjoy a nice rewarding experience.

Class Pixel- [xxpts]

The main idea of this class is to be able to define a 'big' square pixel of size `scale*scale` (where `scale` is the number of physical pixels). Each big pixel have their own color, position, and direction associated with it. The main function is provided. It includes the creation of Tk window and the canvas. It also includes binding instructions between a particular event (an event could be any stroke on your keyboard with letter/number/symbol or mouse click) and a particular action. Here the actions of pressing 1 to 5 are associated with the test functions 1 to 5. To provide a bit more context, a binding can be set up as follows:
`root.bind("<Event>",lambda e:action())`
where "Event" represents a particular event such as `Button-1` for left click, or `Left` for left arrow, `Right` for right arrow, `1` for pressing 1 on the keyboard, etc. In addition, `action` stands for a particular method or function that will be called if the event happens. We note that `lambda` is a Tkinter keyword used to create a link between a Tkinter event `e` and a callback function.

When running `Pixel.py`, the main function of the program is executed. A square black window 800x800 should appear (800 is the number of physical pixels). In the code we are using big pixel of size `scale=20`, so the tkinter canvas could be used to represent a `40x40` (nrow x ncol) square of big pixels. If you press 1, the function `test1` will be executed and once your class Pixel would be implemented you should see Figure 2, and some info printed on screen.

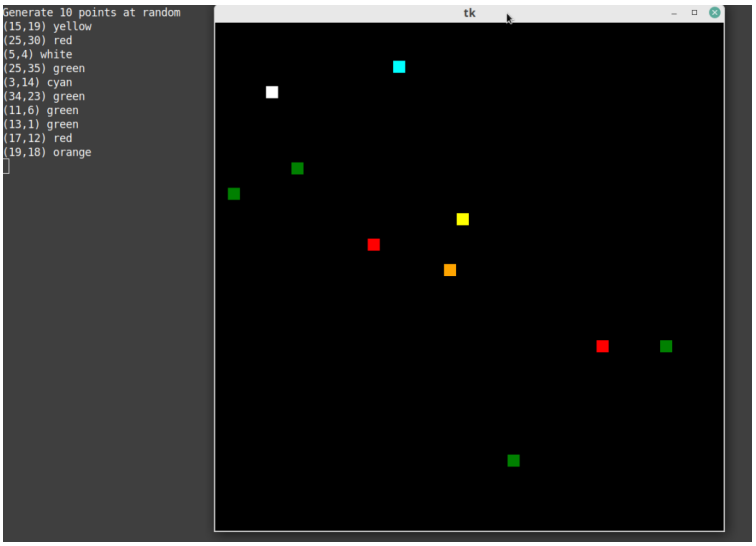

Figure 2: `test1 of Pixel.py`. Generate 10 random 'big' pixels. Display the i,j tkinter coordinates of the big pixel (i is the row number, and j the column number).

Remark1: Once a i,j coordinate is selected for a big pixel, it is is represented by a square in the physical canvas of size `scale*scale` (we use the top left corner of the pixel as origin). For example (to clarify), the big pixel (4,6) would give rise to a square in a physical world with a top left corner at 4*20=80 and 6*20=120 (and of size 20x20)

Remark2: You can clear the canvas at any time by pressing the key `d`.

Now, if you press 2, the function `test2` will generate 10 pixels as well. In addition to a new plot, you will see those info printed on screen (since the input coordinates were too large, we bring them back inside the window using the modulo `%` operator):

```
Generate 10 points at random (using modulo)
1326,208 -> (6,8) orange
1122,13 -> (2,13) brown
510,39 -> (30,39) red
238,299 -> (38,19) brown
510,312 -> (30,32) cyan
204,468 -> (4,28) blue
0,169 -> (0,9) purple
578,143 -> (18,23) purple
340,52 -> (20,12) red
1326,507 -> (6,27) brown
```

After clearing the canvas, you can try test3 where a red square will travel counterclockwise along a square in the canvas, and then disappear. Test 4 would have 4 square of different colors traveling a side of a square before disappearing. Finally, Test 5 would have a single yellow square originally fixed in the middle of the canvas. New tkinter bindings are associated with this test where you could use the left, right, up, and down keys to move the square. The square keeps moving in the direction you choose (it never stops), and once it encounters a boundary it appear on the other side (boundaries are periodic).

How to proceed?

- Implement the class Pixel by proceeding step by step with the tests from 1 to 5. Each test gives new clues on how to implement this class.
- The class `Pixel` has a constructor that should accept five arguments: the canvas, the i and j position of the big pixel (in its own coordinate systems), the number of row, columns, the physical size of the big pixel, the color ID (number), and a list of size 2 to specify the direction (starting from test3). Within the constructor, you will have to instantiate a rectangle that represents the big pixel (you may want to save it by creating a new attribute).
- Starting from test1, you will also need to define a __str__ method.
- A list of approved colors is defined as a class variable and using the ID color, you can retrieve the correct color by calling that list.
- The random module is used in a few place in the test functions, just a quick crash course:
  `random.choice(list_items)` returns a random item from the list `list_items`
  `random.randint(a,b)` returns a integer at random between a and b (inclusive)
  `random.seed(s)` (s integer) set up the seed for reproducibility (keep the same random order) and already set-up for you
- The direction attribute (named `vector`) works like a vector displacement and could take (in general) the values `[0,0]` (pixel is fixed by default), `[1,0]` (pixel is directed down), `[-1,0]`

(pixel is directed up), `[0,1]` (pixel is directed right), and `[0,-1]` (pixel is directed left). The +1 and -1 values conveniently represent the values of the displacement of your big pixel along i or j, if the pixel is instructed to move.

- The methods `right`, `left`, `down`, `up` can simply be used to change the values of the direction attribute, while the method `next` is achieving one actual move (representing the next step of the simulation). You can use the canvas methods `move` or `coords` (as seen in Lecture 3.4 to move the big pixel).
- You can delete a canvas object you created using the method `delete(myobject)` from canvas.

Class Grid- [xxpts]

The class `Grid` is using the class `Pixel` to generate big pixels into a regular **gray** grid background and perform new actions. If you execute the code (once completed), you should obtain a plot similar to the one in Figure 3 (your white pixels will likely be at a different cell positions since fully random (we do not use any random seed here).

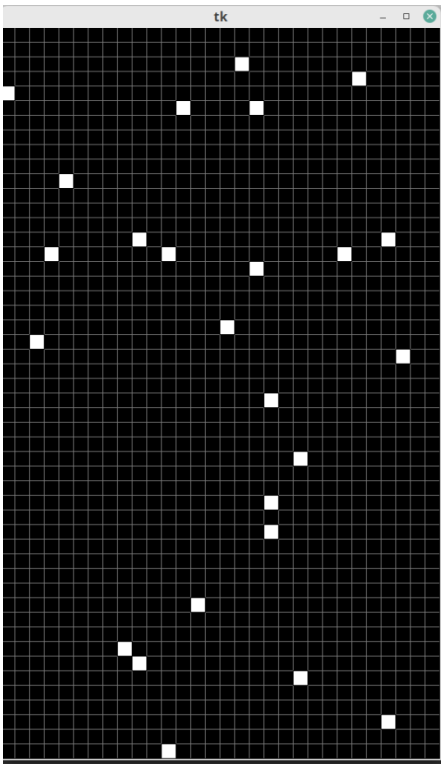

Figure 3: `Grid.py`. Original grid with 25 big white pixels (positioning is random).

Now if you left click on your mouse somewhere in that grid where there is no white pixel, a new white pixel would appear with info similar to this one (using 9 clicks here):

```
insert 176 608 30 8 0
insert 226 483 24 11 0
insert 87 410 20 4 0
insert 80 642 32 4 0
insert 144 781 39 7 0
```

```
insert 255 667 33 12 0
insert 543 632 31 27 0
insert 274 796 39 13 1
insert 412 889 44 20 1
```

The first two numbers are the x,y physical tkinter coordinates (the x,y location of your click in the physical tkinter canvas– Hint: x is toward the left and y toward the bottom, origin is top left corner). The two following numbers are the corresponding i,j coordinates of your cell grid. The last number is the color what was at this cell grid coordinate at the moment of clicking (0 for black, 1 for white). For the last two clicks, we left click on a cell that was already white (so nothing happens in the grid).

If instead you decide to right click on a white pixel, that pixel should disappear from the grid (example with 4 clicks here).

```
delete 409 890 44 20 1
delete 369 691 34 18 1
delete 231 310 15 11 1
delete 354 717 35 17 0
```

Attention, if you decide to right click on an empty cell (black, like the last click in the example above), something very different would happen. The entire row must disappear using a fun "animated flushing way" (see video). As a result, the entire row would be flushed, and all the cells that are placed above this row will shift one cell down.

How to proceed?

- The main method is provided. The tkinter `root` that is instantiated, is now becoming an argument of the `Grid` constructor, along with the number of rows, columns and scale of the grid cells.
- In the `Grid` constructor, you can instantiate the canvas (which becomes an instance attribute) and pack it. You can then plot the entire mesh grid using Gray color. At this point, you could use an empty list as another instance attribute that can be used to keep track of the pixel objects that you are going to create. Finally it will become handy to initialize a `nrow*ncol` numpy **integer** matrix (as another instance attribute) that could be used to record if a cell is empty or not. All the elements of the matrix can be initialized to "0" since the cells are empty to start with. Later, for example, if you decide to create a red pixel at (23,4), your matrix could record this information as `matrix[23,4]=3`, 3 being the red color as specified in the Pixel static variable.
- Create a method `random_pixels` that will accept as input the number of pixels to create at random positions, and their color number (same color for all the random pixels created). Here it is useful to call (and create first) a method `addij(i,j,c)` that create a single pixel at the position (i,j) with color number c. This method could simply instantiate a Pixel object at this position, and this pixel object could be appended to your list of pixel objects. Note that

you should do this only if the color is not black (> 0). In addition, if you create a non-black pixel object, you want to update the correct matrix element with 1 (white color) to make sure the corresponding cell of the grid is now occupied.

- As you can see from the main code, we are binding the left click of you mouse with the method `addxy` while the right click is binded with the method `delxy`. The Tkinter binding is creating for you an event `e` where the attributes `e.x` and `e.y` are the (x,y) coordinates in the Tkinter canvas where the click occurs. They are used as input arguments for both methods. In both `addxy` and `delxy` methods you need to convert those x,y coordinates as i,j for the mesh cell (see examples above). Once done, in the `addxy` method you would need to display some info (again see example above), and you could call your `addij` method since you have the i,j coordinate. Once the (i,j) coordinated are obtained in the `delxy`, display some info on screen, and then proceed by calling a method `delij`.

- The method `delij` must consider two options:

  1. if the color of the pixel you want to erase is not black (the color number is obtained by looking at the i,j element of the matrix), you need first to update the matrix element to zero. Now, instead of deleting the corresponding pixel, we propose to adopt another strategy (that would be useful later for one application game). You need to implement and call a method `reset` that would delete all the pixel objects in your list of pixels (all the pixels will be removed and the grid would be empty, Hint: you have implemented the `delete` method in `Pixel`). Then this method will scan your entire matrix and call the `addij` method if matrix element is different than zero. You will then redraw this way all the pixels. This approach is instantaneous so it will give the illusion that only one pixel has been removed.
  2. if the color of the pixel you want to remove is black (so empty cell), you should call (and implement) a method `flush_row(i)` where `i` is the row number that is going to be flushed. Watching the video you will see that three big purple pixels will first start appearing on the extreme left and extreme right of the row. You can create a list to store those big pixels (you could use 2 lists or 1 list to store the 6 pixels). Note that you know the location and the direction of each pixels (pixels from the left should have a right direction vector attribute, and pixels from the right should have a left direction vector attribute). At this point you can look at `test4` and `test5` from the `Pixel` class, to implement your own animation of those pixels. I am using a sleep time of 0.02s to make the animation faster (you can play around with this number to slow down the animation to debug if needed). The animation must stop when the pixels reach around the center of the row. Now you can delete all those purple pixels (Hint: you have implemented the `delete` method in `Pixel`). You then need to shift your matrix. As a hint and example, if you want the first 3 rows of a matrix A to go down, you could do something like `A[1:4,:]=A[0:3,:]`, and do not forget to nullify the first row of the updated matrix. As you may have guessed, you could just use your `reset` method implemented above to obtain the new grid configuration.

Class Tetrominoes- [xxpts]

A little bit like the class `Pixel`, this class comes with its own set of test functions. If you execute the code, a black canvas of 900x600 size should appear. You can press either 1, 2, 3, 4 or d (delete all), to perform to some action. Each test functions from 1 to 4, provides clues on how to implement the code.

Test1

Using test1, you should obtain a plot like the one in Figure 4.

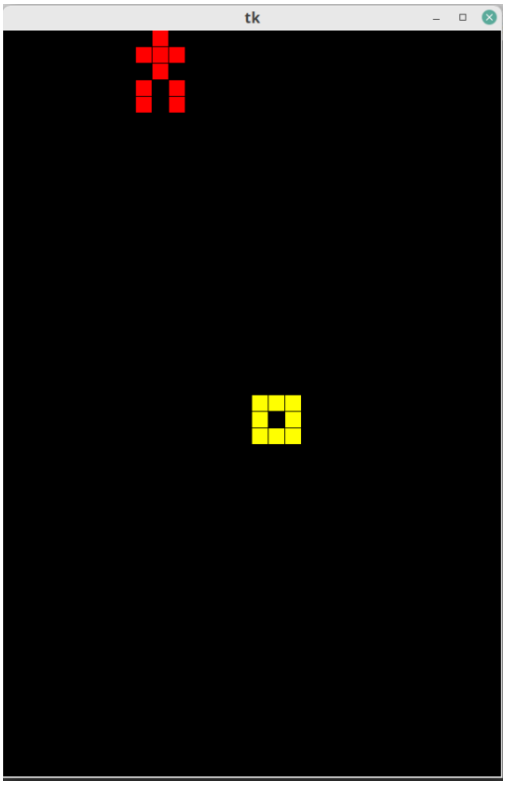

Figure 4: `Tetrominoes.py`. Result from test1. The red fellow appear at random on the top (so could be in different "y" position at the top for you).

In addition, the following info would appear on screen as well:

```
Generate a Tetromino (basic shape)- different options
Tetro1 Basic
  number of patterns: 1
  current pattern:
 [[2 2 2]
 [2 0 2]
 [2 2 2]]
  height/width: 3 3
  i/j coords:   22 15

Tetro2 Custom
  number of patterns: 1
  current pattern:
 [[0 3 0]
```

```
[3 3 3]
[0 3 0]
[3 0 3]
[3 0 3]]
 height/width: 5 3
 i/j coords:   0 8
```

This `test1` will help you implementing the constructor of `Tetrominoes` and the methods `activate` and `get_pattern`. The last two input arguments of the constructor are optional and should be the color ID (set to the value 2 –yellow– by default), and a pattern_list. The pattern_list consists of a list of numpy matrices (e.g. `patterns=[A0,A1,A2,...]` where the `Ai` are numpy matrices. Stated otherwise, each item of this list is a 2D numpy array that represents a particular Tetromino pattern of the same family. For `test1` you will consider only a single item in the list that could be either the default pattern (Basic shape) or the customized red fellow pattern. To make things clear, if there is no pattern_list provided to the constructor, you will create a default pattern_list (a list with only one item) where the single item `A0` is a 3x3 numpy matrix filled with 2 (2 is the default ID color) but in the center (0 value in there). It would look like the yellow empty square you have in the plot. Do not forget to initialize the attribute `name` (default value is "Basic") and attributes `h` for height and `w` for width (which are 3 and 3 for the basic shape, but look into how do extract those using numpy methods). For this project, we are assuming that even if you add more matrix patterns for the same Tetromino family into the list, they should all have the same height and width. It is also useful to keep track of the current pattern you are working with (index of the pattern in the list), and this `current` attribute should be set to zero in the constructor (representing the first item in the list). In the method `get_pattern` you will be able to return the current matrix pattern (which is right now `A0` the first item of the list of pattern, since the `current` attribute is set to zero at this point). Attention, the constructor is just initializing some attributes but it should not be used to plot anything. For this, you need to use the method `activate(i,j)` where (i,j) are the locations of your top left pixel of your Tetromino pattern (same coordinate framework used for the `Pixel` class). If (i,j) are no specified, the default value for i should be 0 (first row), and it would be a random number for j taken from 0 to `ncol-w` (indeed, you need to account for the width of the Tetromino to not step beyond the right boundary). Requirement: You must save the those i,j coordinates as new attributes (will be useful for the Tetris game). While scanning the matrix of your current pattern, you will be able to define a list of big pixels (and create a new attribute). Since you will instantiate all the non-black pixels, your Tetromino will appear on screen. Note that a 3x3 matrix should produce a shape of 3x3 big pixels.

Test2

If you made it that far with no issue, you can try out `test2`. This test uses two 3x3 matrix blue patterns. To be able to switch from one pattern to another, you need to implement the method `rotate`, the method will first need to delete all the list of pixels created within the `activate` method (actually it could be useful to implement and use a dedicated method `delete` to just do that), then you need to change the value of the `current` attribute (increment by one, modulo the length of the list of pattern so you can cycle back), finally call `activate` again. When you execute `test2` the Tetromino should oscillate (rotate) between its two patterns (look at the video), it will do that ten times and display the following info on screen (just the beginning is shown here):

```
Generate a 'square' Tetromino (with double shape) and rotate
My Tetro
  number of patterns: 2
  height/width: 3 3
  i/j coords:   22 15
  current pattern:
 [[0 0 4]
 [0 4 0]
 [4 0 0]]
  current pattern:
 [[4 0 0]
 [0 4 0]
 [0 0 4]]
...
```

Test3

You need to define seven new child classes that all inherit from the parent class `Tetrominoes`. All those classes have their own list of patterns but all of them of matrix size 3x3.

Here are the classes name:

- `TShape`, which contains 4 different patterns with red color.
- `TripodA`, which contains 4 different patterns with blue color.
- `TripodB`, which contains 4 different patterns with green color.
- `SnakeA`, which contains 2 different patterns with orange color.
- `SnakeB`, which contains 2 different patterns with purple color.
- `Cube`, which contains 3 different patterns with brown color.
- `Pencil`, which contains 4 different patterns with cyan color.

Figure 5 shows the original pattern (first item of the list of patterns) associated with each class (including the parent class in yellow).

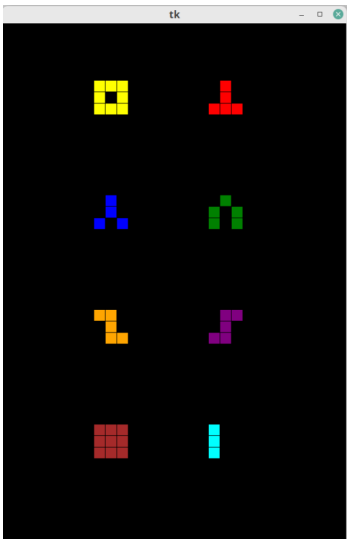

Figure 5: `Tetrominoes.py`. Result from test3. Original pattern for the 8 classes (the one of the top left is the parent class).

The function `test3` is instantiating an object for each one of those classes and then create a list of all objects. Note that the child classes inherit all attributes and methods of the parent class, such as the `activate` method and the `rotate` method. In the function, we are also binding the "space bar" to one particular rotation for all those shapes. If you keep pressing the "space bar" you will rotate between all the possible patterns for each class. You can watch the video to make sure you can identify all the correct 3x3 patterns for all the subclasses.

How to proceed?

- The constructor `__init__` method is the only method needed for all the child classes.
- The child constructors should start by defining their list of patterns and then call the parent constructor using the `super` operator.
- The child constructors can then modify some attributes if needed (like `name`).
- That's it.

Test4

Here a Tetromino is selected at random (the static method to select the Tetromino is already implemented for you) and positioned at random at the top. In addition to the "space bar" that can still be used to rotate the shape, the function is binding the left/right/up/down key arrows to move it. You can watch the video to get the idea.

You need to implement the methods `left`, `right`, `up`, `down` to perform a single move (one pixel move). These methods belong to the parent class so they can be accessible by the child classes. For example if you choose `right` all the pixels that composed the shape will move right by 1 pixel. Since you already have an attribute representing the list of the pixel objects that composed the pattern, it should be straightforward to call the `right` method of each pixel followed by its `next` method to achieve the move. For this example, do not forget to also increment by 1 the attribute of the j coordinate of the shape (modulo the number of columns).

Class Snake- [xxpts]

You need the class `Pixel` and `Grid` to implement this Snake game. Here is what to expect:

- When the game starts you should see a grid canvas with a random set of white pixels and a random set of red pixels.
- In the middle of the grid, there will be a snake of 4 pixel length, where the head is represented by one blue pixel and its body is represented by three green pixel. The snake orientation is toward the right and as soon as the game starts it is moving toward the right.
- You are allowed to change the forward moving direction of the snake, for example if the snake is moving right, you are allowed to press the up or down keys (pressing the left or right keys would have no effect). If you decide to press the down key (for example), the cell where the head of the snake is located will be recorded somehow and every pixels will be forced to change their direction down if their reach this cell.
- Like for the Pixel class, the system is periodic so the snake can leave the right boundary and reappear to the left (same for up/down).
- The white pixels represent some obstacles that the snake should not bump into, otherwise it will be game over (the game stops and a Game Over message should appear).
- The red pixels are strawberries that the snake wants to eat. If the snake eats all the berries, you won the game (the game stops and a Winning message should appear). The problem is that the body of the snake keeps growing by one pixel at each time it eats a berry!
- To manage your nerves, you will consider implementing a pause option (by pressing the p key).

Figure 6 represents a snapshot of what could happen.

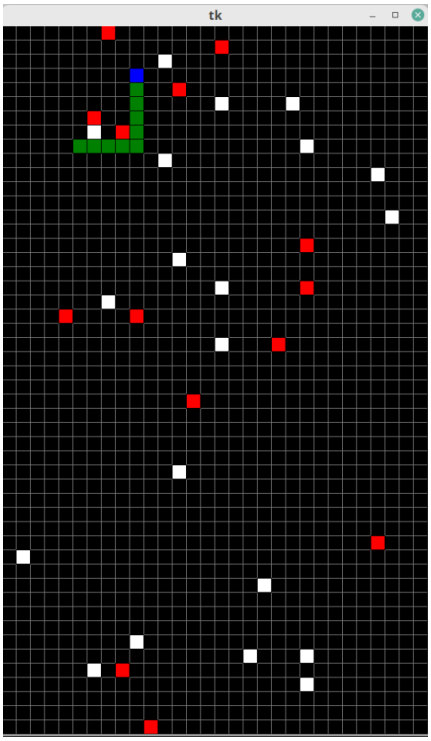

Figure 6: `Snake.py`.
The Snake game in action!

How to proceed?

- The `Snake` class must be a child class of the `Grid` parent class, and you will start by calling the `super` operator.
- Since it inherits from all the methods in `Grid`, you can use the `random_pixels` method to generate both white obstacles and red fruits. You may also want to define an attribute that represents the number of fruits (so you can check later on if you win the game).
- The constructor should also contain a list attribute to store all the pixels representing the snake. The last item/pixel of this list (index -1), is representing the blue head snake. All pixels have directions `[0,1]` to start with. That's it for the constructor (for the time being).
- It is time to implement the method `left`, `right`, `up`, `down`. To be able to perform a turn, we are going to use some trick strategy (which was suggested by your TA Braegan). First, make sure you can retrieve the (i,j) coordinates of the head pixel (index -1 in your list of pixels defining the snake....also you have access to i and j since they are attributes of the pixel object). Second, check the direction vector of the head pixel, if it is going toward the right and the method you are calling is `left` or `right` then nothing should happen and you should leave the method (and so on and so forth for the other options and methods). At this point of the simulation, the matrix representing the grid (also an attribute of `Grid` that you have access to) should automatically contain a bunch of value 1 (representing the white obstacles) and values 3 (representing the fruits). Let us now simply use a sentinel value (also called turning point) that we are going to assign to the (snake head) (i,j) matrix element. I am using -1 to signal an allowed right turn, -2 an up turn, -3 a left turn and -4 a down turn (the negative value guarantees that the code would not be mistaken that value with a color ID value). All done.
- Now, you need to implement the method `next`. Let us proceed step by step:
  - You need to scan all the list of pixels representing the snake and extract their (i,j) coordinates.
  - For each pixel, check if the corresponding (i,j) matrix element is a turning point, if so change the direction of the pixel appropriately. For example if the matrix element is -3, you can call the `left` method (in the class Pixel) on this particular pixel.
  - For each pixel, call their `next` method. Do not forget that if the coordinate of the last pixel of the body has reached the turning point (you can check if the matrix element at this coordinate is negative), you can remove the turning point from the grid (by setting the matrix element to zero).
  - After all pixels have moved, extract the new (i,j) coordinates of the head of the snake. Check if the corresponding matrix element is equal to 3 (it is fruit!) and if so, you will need to insert a new green Pixel in the list of snake pixels (you need to insert at index 0). The direction of this new pixel should be the same than the one that was the last pixel of the body. Do not forget to use the `delij` method to remove the fruit from the grid and decrement your fruit counter. Remark: the way the Grid class is set up, you may need to modify your `addij` method to make sure that no pixel is added if the color is not "greater than zero" (in case you used "different than zero" previously...). Indeed turning points are negative and when you replot the grid using the `reset` method, the color cannot be negative...however it must be possible to still add the negative value into the matrix grid.

- If the new matrix element at the head is equal to 1 (it is an obstacle!) or if the number of fruits reaches zero, it is game-over. In one situation, you can print the message "GAME OVER", in the other situation you can print the message "YOU WON". In both cases, you need to be able to signal that the game is over. Here you will use a **private** attribute that you can set to False in your constructor, and it would become True if the game is over. As you can see, the main method is calling the method `is_game_over` that should return the value of this private attribute. Remark: we use a private attribute since we do not want want a user to hack the game and win without even trying...that would be cheating.
- Similarly to the attribute for signaling if the game is over, you could use another **private** attribute called "__pause" and set it to False in your constructor to stat with. You need two new methods `is_pause` that returns the value of the attribute, and a method `pause` that is binded with the "p" key in the main code. This method will change the private attribute to True if False, or False if True.
- Have fun with your game!

Class Tetris- [xxpts]

Tetris celebrates its 40 years! I thought it would be a good idea to acknowledge this innovative game for this year project.
To implement this class, you need the class `Pixel` and `Grid` and `Tetrominoes`. Here is what to expect:

- Like the original Tetris game, some Tetrominoes appear from the top at random, and you have the possibility to rotate the shape.
- Unlike the Tetris game, our Tetrominoes are a bit different (already defined previously), but they all define within a 3x3 square pixels (even our pencil shape rotates within a 3x3 square).
- Like Tetris, you can move left and right the Tetrominoes as they fall down. If you move down, it will automatically go into place.
- If you form a filled row or filled rows, they will be flushed (and you can earn some points...but we are not keeping track of the score in this project).
- If the grid becomes to full you loose.

Figure 7 represents a snapshot of what could happen.

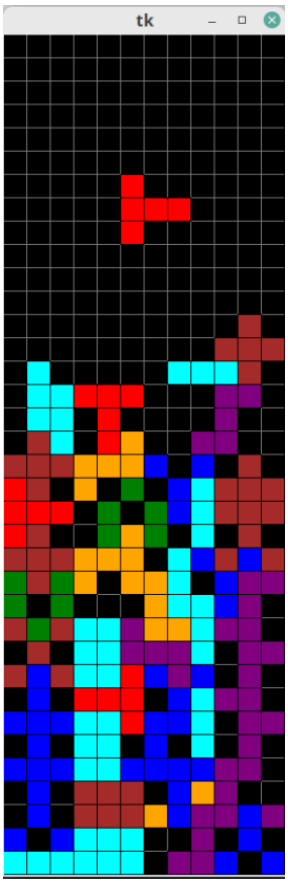

Figure 7: `Tetris.py`.
The Tetris game version **1.2.2** in action!...I am badly loosing.

How to proceed?

- The `Tetris` class must be a child class of the `Grid` parent class, and you will start by calling the `super` operator. Define an attribute name that would represent a given falling Tetromino (let us call it `block` to ease the description below) that you will need to set to `None` to start with. In this Tetris game, we are going to consider one falling block at the time.
- Let us first focus on the method `next` (the most difficult one), which represents the next iteration of the simulation (where the block is falling). Let us proceed by steps:
  - First, if the value of your `block` is `None`, it means that no block is active and you need to instantiate `block` as a new Tetrominoes, for this, you can use the method provided `random_select` that selects a Tetromino at random. You will need to `activate` this `block` (it will place it on top of the canvas)
  - Second, just call the Tetromino method `down` for this `block` (the block should move one cell down). You would need to retrieve the new (i,j) coordinates of this block.
  - Check if the block has reached the bottom, if so the block is in place.
  - Check if the block is on top on another block, if so the block is in place. This part is a bit trickier, and to accomplish this, I created a method `is_overlapping(self,ii,jj)` where the coordinates (ii,jj) represents what would be the coordinate of the block after an additional move (here `ii=i+1` and `jj=j` since the block is falling down...so one more row represents the next move). In this method, you could check if the **non-zero** pattern of the Tetromino overlaps in any way with 3x3 extracted matrix elements from the grid at the ii,jj position. If an overlaps exists, it means that your block is already in place.
  - If you find that your block is in place, you first need to proceed as follows: (i) save the current block pattern, (ii) delete the block (you should have the method `delete` already implemented from Tetrominoes), (iii) call the `addij` method of the Grid class to insert the pixels at the pattern location (this way we are transfering the pattern of the block in-place to the Grid, and delete the block itself).
  - Once the block is in-place, you need to scan all the rows of your matrix grid to see if a value 0 is at least present. This way, you would be able flush the rows that are completely filled (use your method `flush_row` to do that). You may also want to check if there is a non-zero matrix element present in the the first three rows of your matrix, if so, you lost the game!
  - Before leaving the method and if the block is in place, you can set its value to None (so a new block could be created if you call `next` again).
  - Finally, use a **private** variable to signal that the game is over. See the instructions from the `Snake` game on how to proceed, for implementing the methods `is_game_over`, `pause`, and `is_pause`.
- if you made it that far, all your blocks will be able to fall. Now, you need to implement the `up`, `down`, `left` and `right` methods. The `up` method is used to rotate the block (between the different pattern of the Tetrominoes... just call your `rotate` method). The `down` method will keep calling `next` until the block is not `None`, so it can drop in-place like a rock. Using the `left` and `right` methods, you will be able to move the block by one pixel left or right. You cannot go beyond the edge boundaries, and you are not allowed to overlap on the side with another Tetrominoes already in-place (Hint: the `is_overlapping` method above should be useful here).
- Have fun with your game!