159.172 Computational Thinking

Programming Assignment 1: Snake Game

This assignment is worth 15% of your final mark. It will be marked out of a total of 25 marks. You are expected to work on this assignment individually and all work that you hand in is expected to be your *own* work.

In this assignment you will develop an animated application that uses several of the concepts that we have studied so far in 159172.

Go to Stream and download the file

```
snake.py
```

Set up a new project and add this file to it.

snake.py is a program that implements the beginnings of a simple animation for the classic "Snake Game". When you run this program, you will see a screen with a (very rudimentary) moving snake. The direction of movement of the snake is controlled by the UP, DOWN, LEFT and RIGHT keys. At the moment the snake just stops when it hits the edge of the screen, until the user presses a suitable key to move it in another direction. It is currently able to double back over the top of itself. In the finished game, either of these scenarios would cause the snake to meet its demise.

The **Snake** class has two attributes:

1. self.segments - an (ordered) list of the body segments that make up the snake, each of these is a *Sprite* object, defined in the *pygame.sprite* module.
2. self.spriteslist - a *pygame.sprite.Group* object, which is an (unordered) container to hold and manage multiple Sprite objects, this allows for drawing or collision detection methods to be applied on the whole group.

The **Segment** class is a subclass of *pygame.sprite.Sprite*. When a new segment object is created we first call the parent constructor to create a new Sprite instance, then assign *image* and *rect* attributes. The rect attribute is the bounding box for the image, we update the position of the Sprite by setting values for *rect.x* and *rect.y*.

The **move** method in the Snake class moves the snake one step, in the direction determined by the key pressed (UP, DOWN, LEFT or RIGHT.) It uses values assigned to the global variables *x_change* and *y_change* to decide where the new "head" of the snake should be, relative to the old head. It creates a new segment at this position and inserts it at the front of the segments list, then pops the last segment from the list. It also updates the spriteslist attribute, using *add* and *remove* methods. This gives the effect of moving the whole body, since every segment except the head should end up in the position of its predecessor at each step.

The main game loop updates the x_change and y_change variables, depending on the key pressed, then calls the move method on the snake object *my_snake* and draws the *my_snake.spriteslist* Group object.

The aim of the "Snake Game" is to have the player control the snake, using the direction keys, so that it "eats" food items by running over them with its head. Each item eaten makes the snake longer, so control becomes progressively more difficult, since the player loses when the snake runs into the screen border, obstacles other than food, or itself.

Your tasks

Before you start, go to Paul Craven's text "Program Arcade Games with Python and Pygame" using this link http://programarcadegames.com, and read Chapter 13 "Introduction to Sprites". See also https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Sprite for documentation on the pygame module *pygame.sprite*.

1. Add some food for the snake to eat - create two new classes

   ```
   class Food():
       ...
   class Food_item(pygame.sprite.Sprite):
       ...
   ```

   that mimic the *Snake* and *Segment* classes. Food_item objects should be Sprites, a Food object should be a Group of randomly-placed food items (doesn't need to maintain an ordering.) The number of food items provided at each point in the game is up to you. Create a new Food object and update the main game loop so that it displays both the snake and the food.

2. Update the main game loop to detect collisions between the head of the snake and any food items. Food items that have been eaten should be removed. You can use the method *pygame.sprite.spritecollide()* to find sprites in a group that collide with another sprite. The return value for this method is a list containing all sprites in a group that intersect with another Sprite. Usage example -

   ```
   hit_list = pygame.sprite.spritecollide(one_sprite,
   sprite_Group, True)
   ```

The True flag will remove each sprite from sprite_Group that collides with one_sprite.

3. Add a *grow()* method to the Snake class that grows the snake by one segment each time it eats a piece of food. Note that you will need to compare rect.x and rect.y positions of the last and second-to-last segments of the snake in order to determine the direction in which the snake's tail should grow. Create a new segment in the correct position and add it to the snake.

4. Add a *replenish()* method to the Food class, to replace food that has been eaten with new randomly-placed food items. Each time the snake eats a piece of food the Food object needs to be replenished. The pattern that this takes is up to you, one for one (recommended, to begin with), or increasing/decreasing amounts, or a new batch provided when the last has completely run out. Make sure that new food items do not land on the snake itself (we didn't worry about this to begin with.) To do this, you can use the method *pygame.sprite.spritecollide()* again, but using a False flag. Usage example -

   ```
   hit_list = pygame.sprite.spritecollide(one_sprite,
   sprite_Group, False)
   ```

   The False flag will ensure that each sprite from sprite_Group that collides with one_sprite is not removed from the Group. If the hit_list is not empty, you can recursively call the replenish() method to try again.

5. Add some obstacles for the snake to avoid - create a new class

   ```
   class Obstacle():
       ...
   ```

   whose instances are randomly sized and placed Sprites that the snake must avoid. Create a new *pygame.sprite.Group()* object and add some obstacles to it. Make sure that obstacles don't collide with the snake in its initial position. Update the main game loop so that it displays the snake, the food and the obstacles.

6. Add code to implement a score for the game. Each time the snake eats a piece of food, increment the score. You can make this a bit more interesting by adding *type* and *points* attributes to the Food_item class. Not all food has to be created equal! (or displayed identically.) Extend the screen so that there is a banner along the bottom (separate from the main game environment) where you display the score.

Displaying text in pygame is a bit tricky, you can do it using these five steps (change the font/message to your liking):

```
# create a Font object from the system fonts
  font = pygame.font.SysFont("comicsansms", 48)

# create an image (Surface) of the text
  text = font.render('Score = ' + str(score), True,
(255, 0, 0))

# get the bounding box for the image
  textrect = text.get_rect()

# set the position
  textrect.centerx = 200
  textrect.centery = 700

# blit (copy) the image onto the screen
  screen.blit(text, textrect)
```

Update the main game loop so that it displays the snake, the food, the obstacles and the current score.

7. Add code to end the game. The player loses when the snake runs into the screen border, obstacles other than food, or itself. If any of these things happen, clear the screen and display a "Game Over" message in the bottom banner area. Note that adding a *game_ended* Boolean flag, and setting it to True here, rather than setting *done = True*, will allow for the display to stay live until the player decides to Quit.

8. **Challenge Exercise:** Add an "enemy" snake. This second snake should be controlled by the program, independently of the player. It should start off at a location somewhat away from the original snake and should move one step on each iteration of the main game loop. You will need to create a new method in the Snake class - *ai_move()* - to control the movement of the enemy snake. You need to make sure that the enemy snake doesn't double back on itself and that it doesn't change direction too often. You can make this snake as smart as you like. For example, before making a move, it can check if this will make it hit the screen border or an obstacle, and if so it can make a different move. The enemy snake doesn't need to eat food, it just obstructs or chases the original snake. If the original snake collides with the enemy snake, the player loses.

- **Extensions (not marked):**

  1. Add code to implement a "high score" ladder, that is initialised and then retained from one play to the next. This will require you to store information in a text file, separate from the game program. When the game begins, ask the player for a name. When the game ends, display the information from your file - high scores along with player names. If the current player has beaten one of these, store the name and the score achieved in your text file.
  2. To make everything look nicer, use graphical sprites rather than filled blocks. For a very simple introduction, see http://kidscancode.org/blog/2016/08/pygame_1-3_more-about-sprites/

Submission

Submit the assignment in Stream as a single zipped file containing:

1. Your completed code, contained in the file *snake.py*.
2. A word document containing annotated screen shots demonstrating the behaviour of your program.
3. If you have completed any extensions to the program, submit your extended code as a separate module, or modules, and include a separate word document demonstrating the extended program behavior.

Marking Scheme:

- Completion of each of the first seven tasks: 3 marks each
- Annotated screenshots demonstrating the behaviour of your program - 2 marks
- Implementation of the Challenge Exercise - 2 marks
- **Total – 25 marks**

Late submission:

Late assignments will be penalised 10% for each weekday past the deadline, for up to five (5) days; after this no marks will be gained. In special circumstances an extension may be obtained from the paper co-ordinator, and these penalties will not apply. Workload will not be considered a special circumstance – you must budget your time.