**Top-level structure of the report**

1. Abstract (in one paragraph, summarise the content of the report. It must be easily comprehensible to someone who has not read the rest of the report.)
2. Introduction (the scope of the mini-project; explain the theory underlying the algorithms, setting the scene for the remainder of the report.)
3. Algorithm details with sample computations (present the pseudocodes and show at least one concrete example detailing how the formulas are used)
4. Strengths and limitations, important practical considerations (discuss the theoretical strengths and limitations of the algorithm. Introduce practical considerations relating to algorithm implementation (e.g. appropriate data structure, software architecture, open source library, etc.).
5. Demonstration codes with comments.
6. User guide for the demonstration codes
7. Conclusion
8. References

int((len(self.body) \*\* 2) \* self.age / 1000)

return int(((len(self.body) \* 100) + self.age))

Sbody T age

**-Abstract**

Genetic Algorithm (GA)

**-Introduction**

The genetic algorithm models the genetic process that gives rise to evolution. In particular, it models biological reproduction, where both parents give some genetic information to their children. Hence, children have similarities with their parents, and there is lots of genetic inheritance. However, there are also random mutations, caused by copying errors when the chromosome material is reproduced, which means that some things do change over time. Real genetics is obviously a lot more complicated than this, but we are taking only the things that we want for our model.

The player controls a dot, square, or object on a bordered plane. As it moves forward, it leaves a trail behind, resembling a moving snake. In some games, the end of the trail is in a fixed position, so the snake continually gets longer as it moves. In another common scheme, the snake has a specific length, so there is a moving tail a fixed number of units away from the head. The player loses when the snake runs into the screen border, a trail or other obstacle, or itself. []

**-Source Code and Algorithm**

**Start-up Code and Algorithm Introduction**

In order to start up project quickly and focus on training snake neural network with GA implementations, we have utilise an open source project from <https://github.com/valentinmace/snake> as our start up code [], which has a playable snake game, neural network module, genetic algorithm module and game interface. However, this code has some potential bugs that not compatible with version, mistakes and lack of documentation. After some testing and code interpretations, we have fixed programme bugs, and improved user interfaces and GA algorithm efficiency.

In our project, we will utilise neural network to model the snake game by evaluating its fitness function which is the combination of snake’s body length and survival time. The GA will be applied to snake games neural networks for evolving through crossover and mutation of networks in population pool, in order to select network with best fitness in each generation.

Each snake has a neural network, which is an input layer of 21 neurons, 1 hidden layers of 16 neurons, and one output layer of 3 neurons.

In accordance with the input layer, the snake can see 7 directions which is front, front left, front right, left, right, back left and back right. In each of these directions, the snake looks for 3 things, which are distance to the food in the map, distance to the wall and distance to its own body from the snake’s head. In the output layer, the 3 outputs are snake move forward, turn left and turn right.

There are 1000 population (this figure can also be defined by user) of snakes created in each generation. All of the neural networks for each snake are randomly initialised in the first generation. Once the entire population is dead which means each snake game ends, each of the snakes will be calculated for a score based on its fitness function. The better snakes will be selected for reproduction to the next generation through crossover and mutation based on these fitness scores.

When two snakes neural network are selected for crossover, one of network components (that could be neuron connections or neurons on hidden layer) will be crossed with each other, which means part of one parent network is mixed with part of another parent network resulting a new child network. Some of the snake neural network will also be selected for mutation as per the pre-defined mutation rate, that one of network components will be replaced with another random number. Then, this process is repeated for 100 generations (this figure can also be defined by user).

**File Structure**

In this project, the source code contains:

-constants.py, main.py, game.py, map.py and snake.py are the playable snake game,

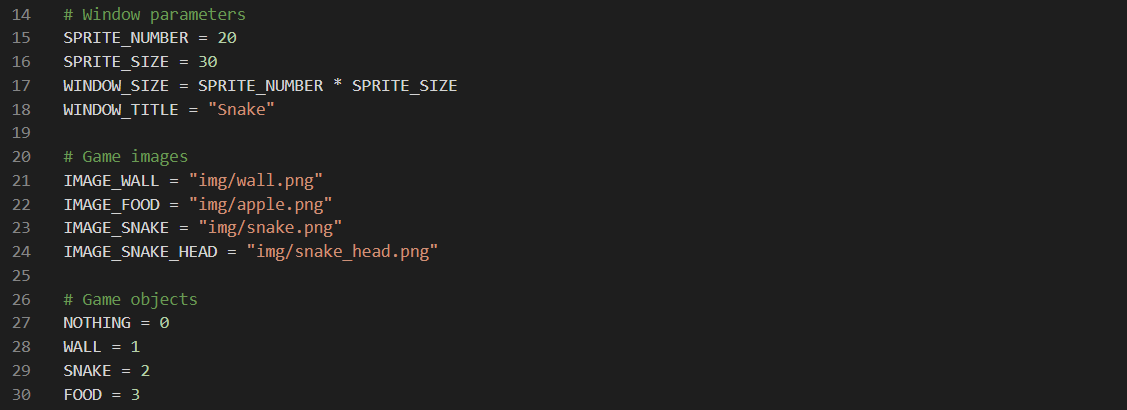
-neural\_network.py and genetic\_algorithm.py are the AI of snake movement.

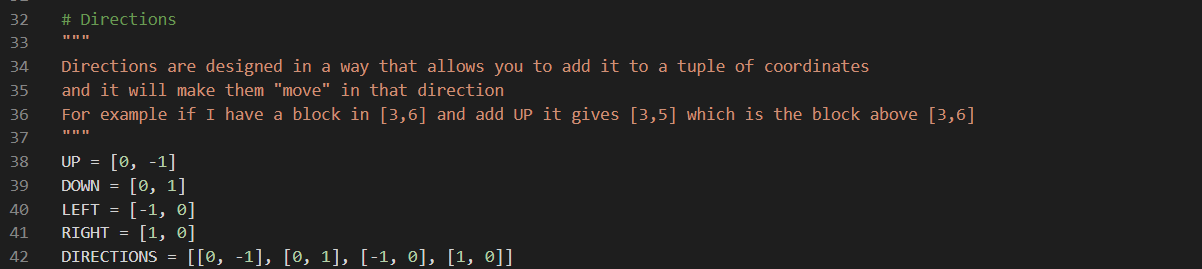
-/img folder is the snake, food and wall graphics for the game

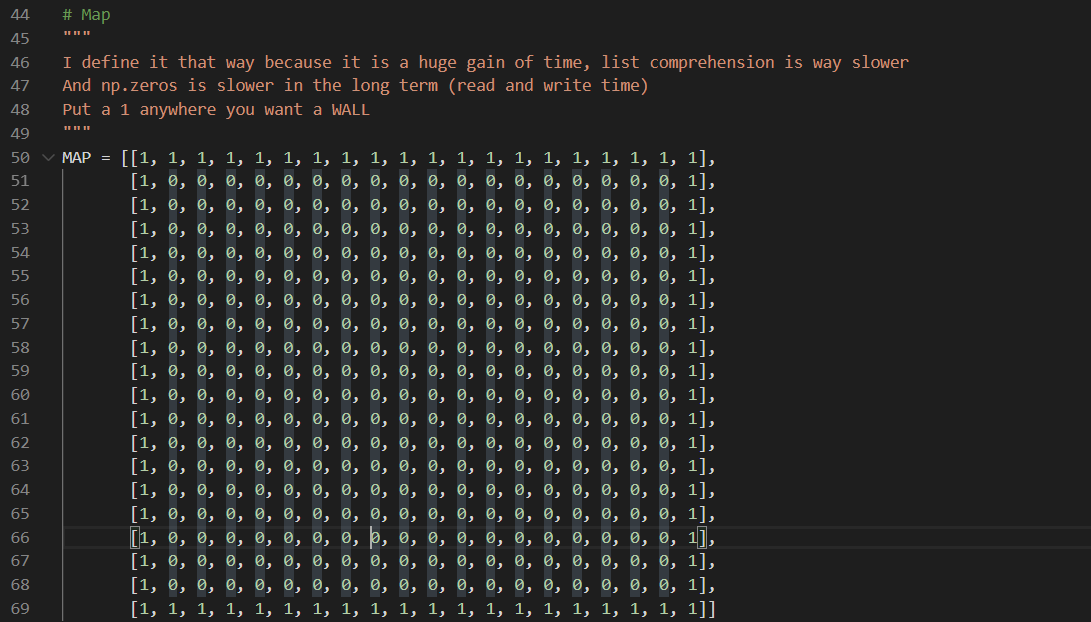
-/gen\_data folder is the saved AI data for each generation of GA.

***constants.py* file**

The *constants.py* file defines all the constant variables for the entire program, includes the size of displayed window of the game, objects images directory, data representations for game objects, snake movement directions, and map.

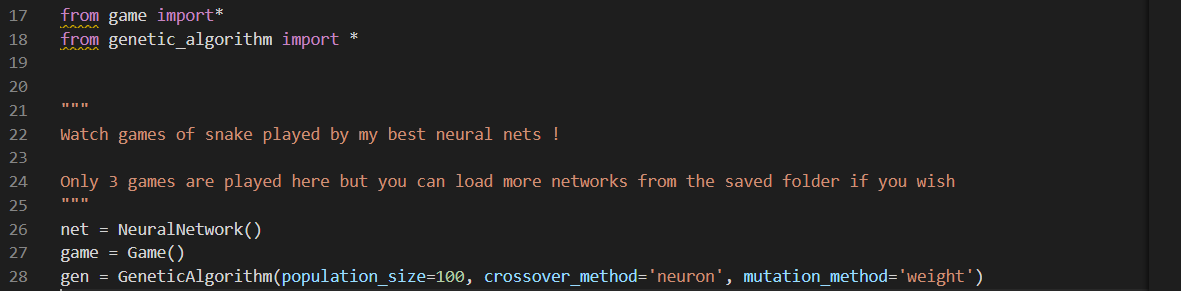






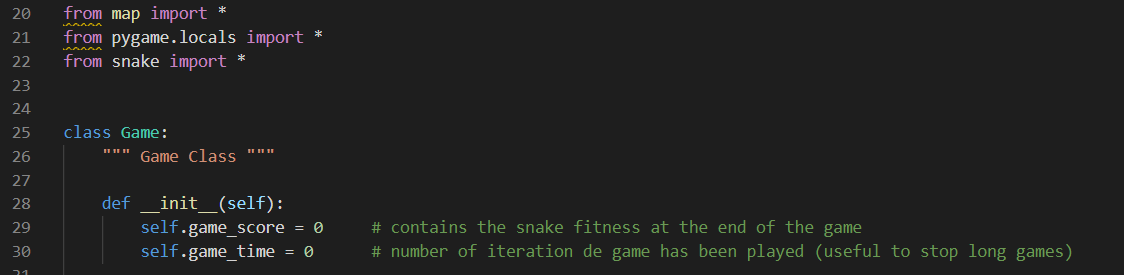
***main.py* file**

*main.py* file is the entry interface to start snake game program, it needs firstly to import *game* and generic\_*algorithm* dependencies , and also define *NeuralNetwork(),Game()* and *GeneticAlgorithm()* objects. Then we can choose to start playing a snake game manually, or watching plays for trained GA model and use GA to train a new model for the snake game.

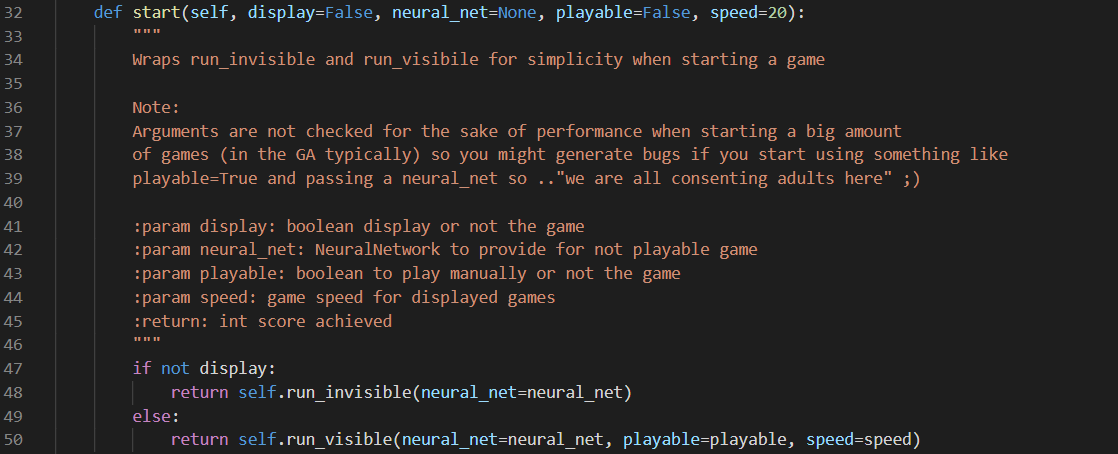


***game.py* file**

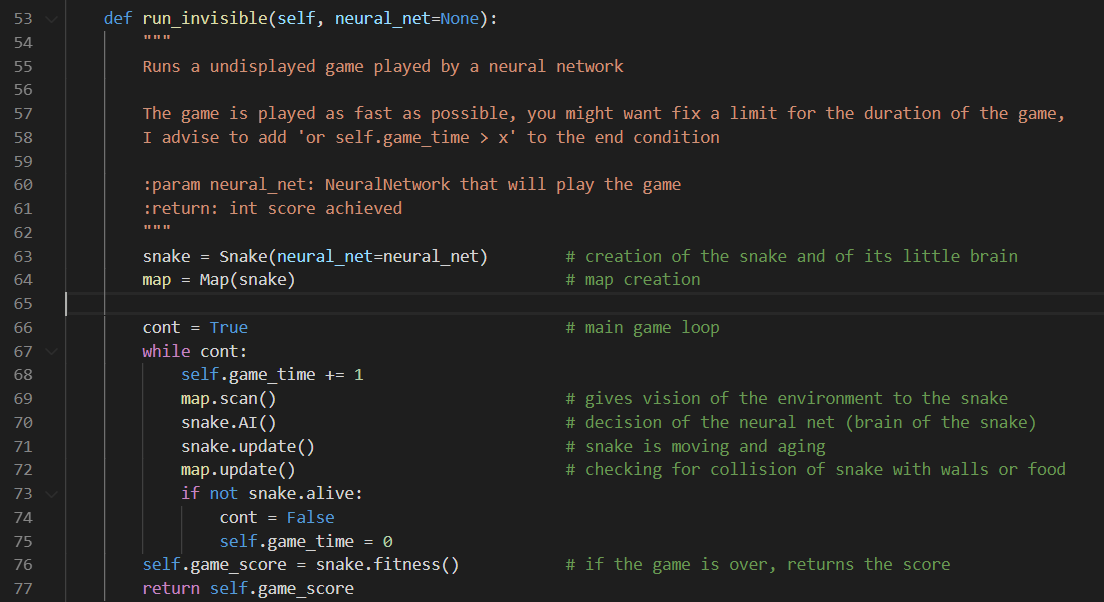
The *game.py* file is to control display of the snake game. The Game object has *start(), run\_invisible(), run\_visible(), inputs\_management()* and *render()* functions.



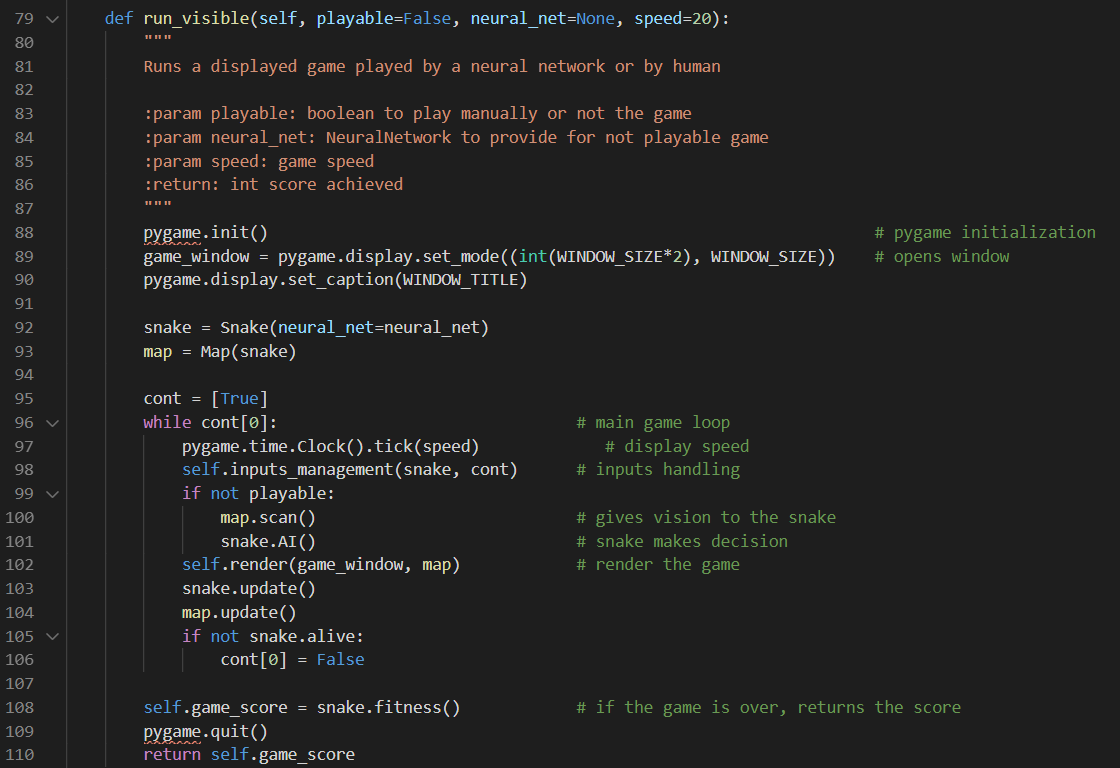
The *start()* function is used to choose if snake game was run visible or invisible after starting the game through calling *run\_invisible() or run\_visible()* functions.



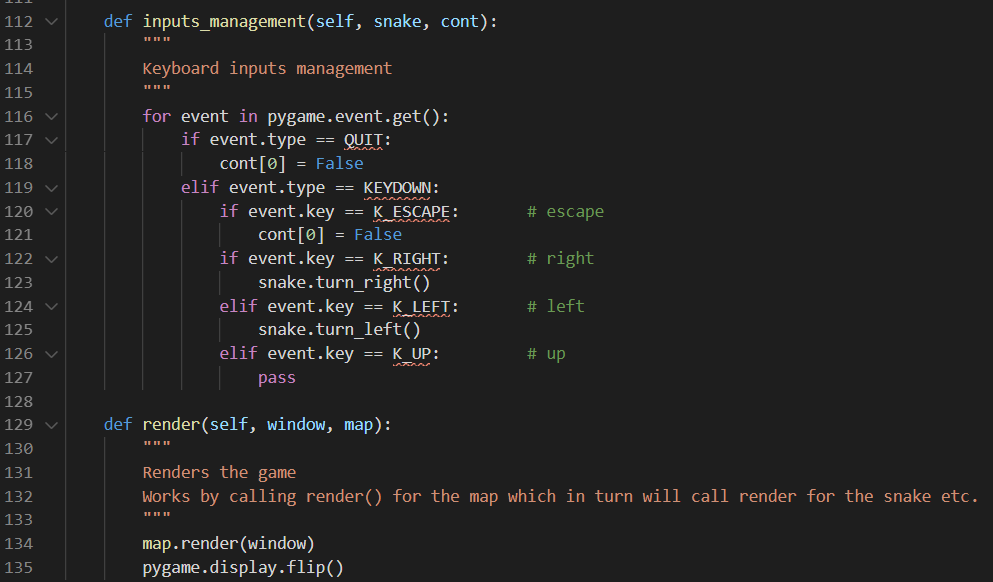
The *run\_invisible()* function is designed for the GA training for a single neural network with random initialised weights and biases. The snake game will run in background, and the snake movement is based on the input of neural network which is distance to the wall, food and snake self-body with 7 directions. Once the game is end, the return value is the game score based on the fitness function of the snake.



The *run\_invisible()* function can make user either display the play for an AI based snake on trained GA neural network or just play manually by setting its playable variable.

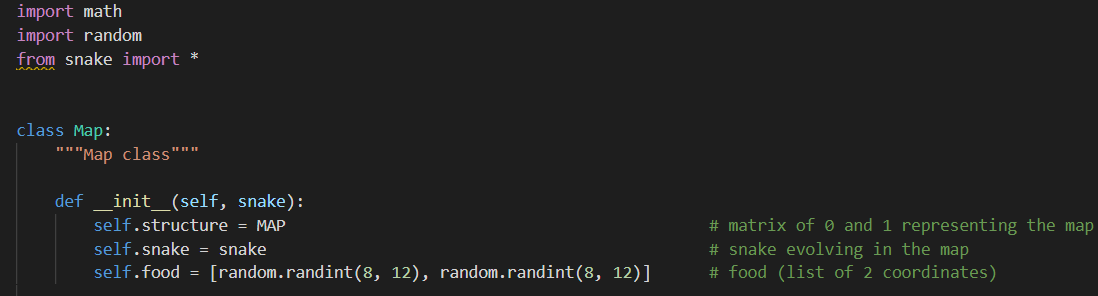


Lastly, *inputs\_management()* function is used to set up short keys for the game and *render()* function is for rendering the visual display of the entire snake game.

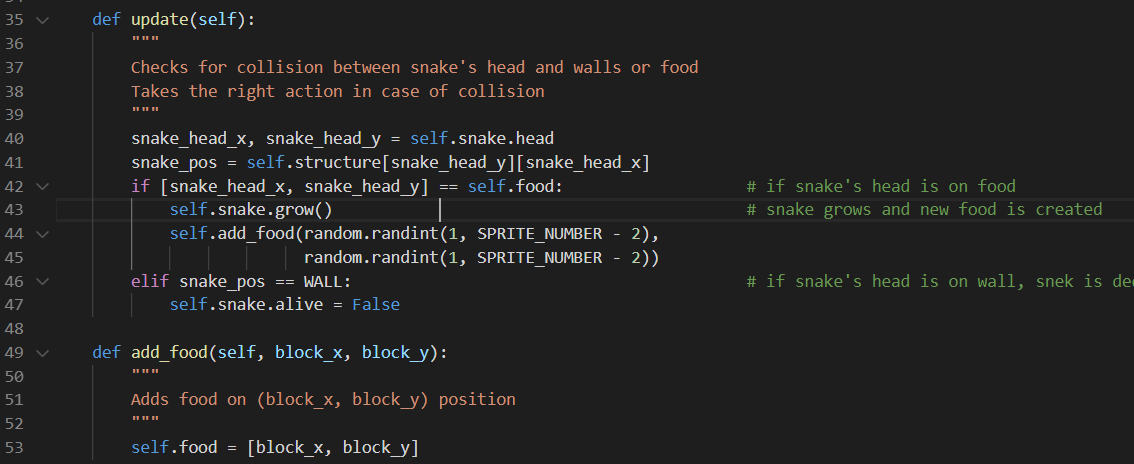


***map.py* file**

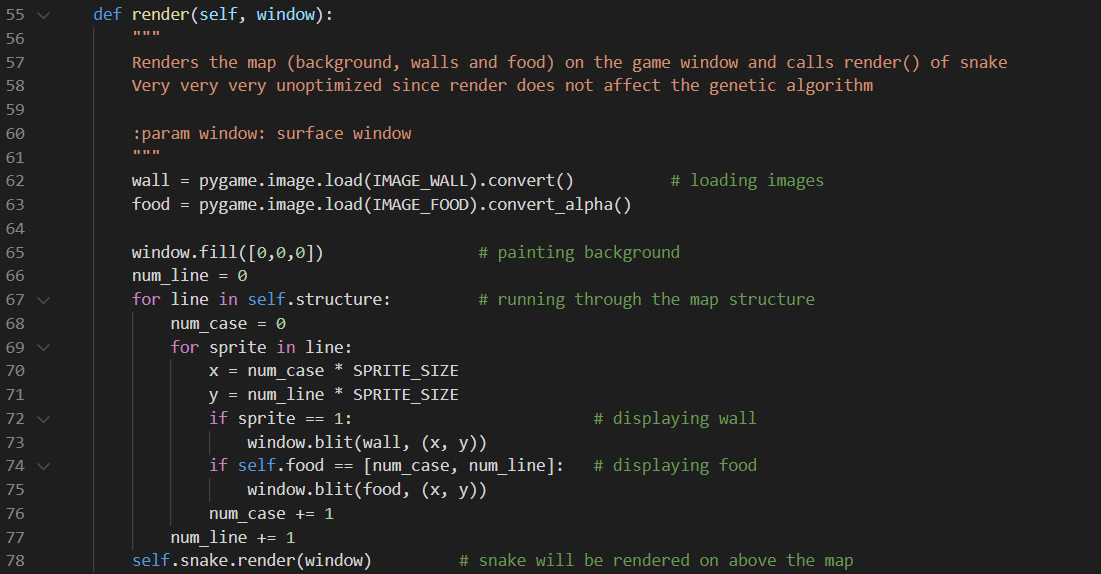
In *map.py* file, it has defined a Map object with snake, food and MAP instances to generate the map for the snake game. The *update(), add\_food(),render()* and *scan()* functions for Map object are used to update map plus to calculate inputs for snake’s AI.



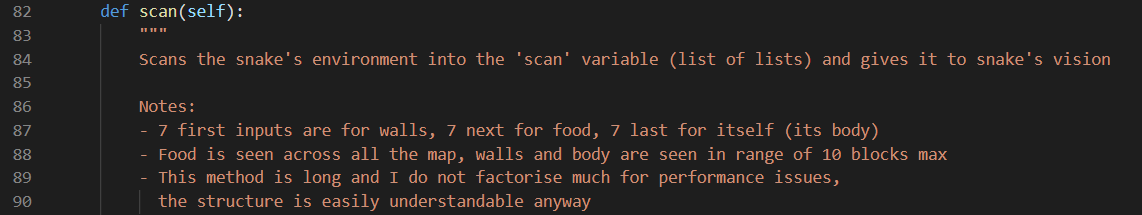
The *update()* function will update the game play dynamically, which checks if snake’s head had eatern the food, and then it will respawn another randomly located food. If snake has eatern the food , it will grow length of one block. In addtion, the game ends if snake hit the wall or its self-body.



The *render()* function will call pygame package API to display the snake movement , growing , and eating food visually in the game play interface.

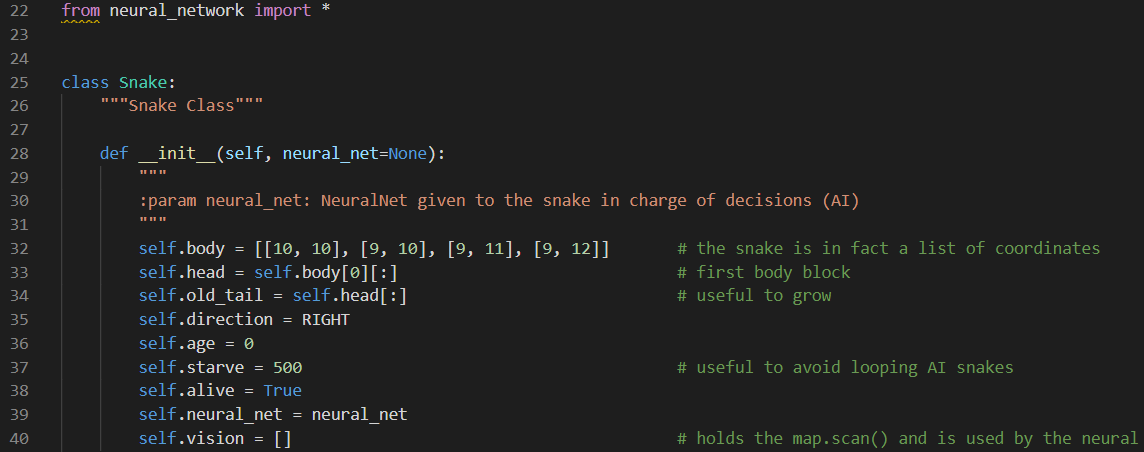


The *scan()* function has *scan\_wall(), scan\_self()* and *scan\_food()* sub functions that detecting the 10 step distances for wall and snake body towards snake’s head position, and the food distance towards snake’s head position. This function will return a 21 x 1 vision array that being the input for snake’s AI which is a neural network.

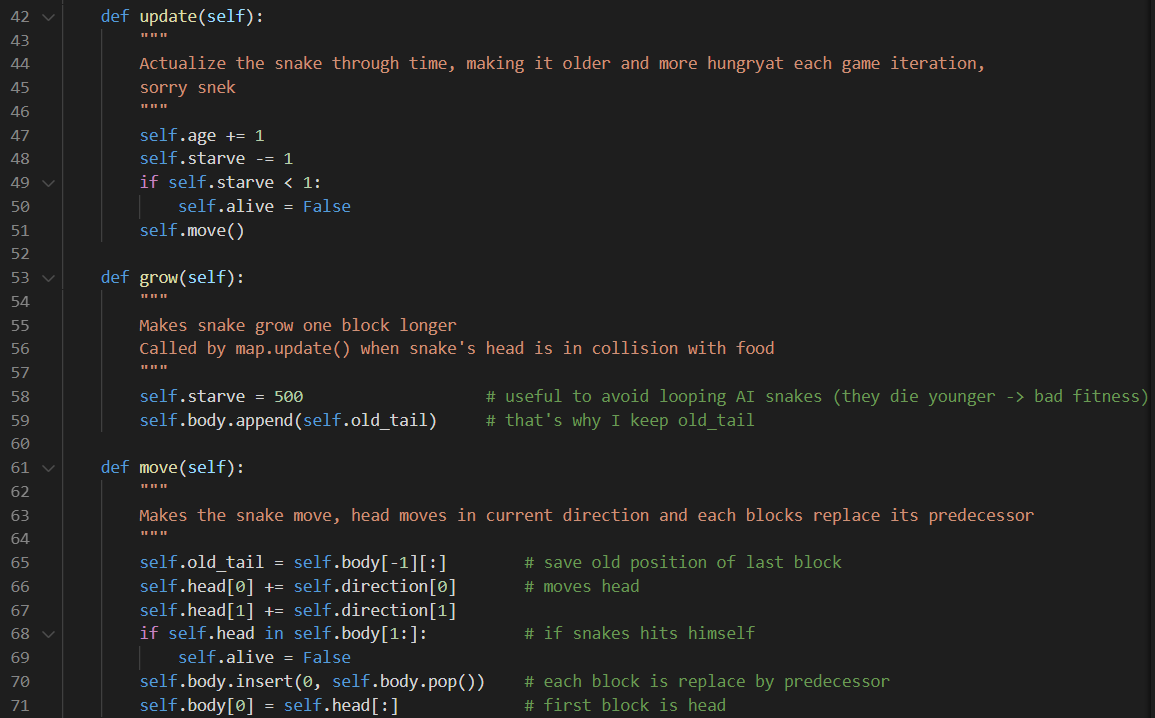
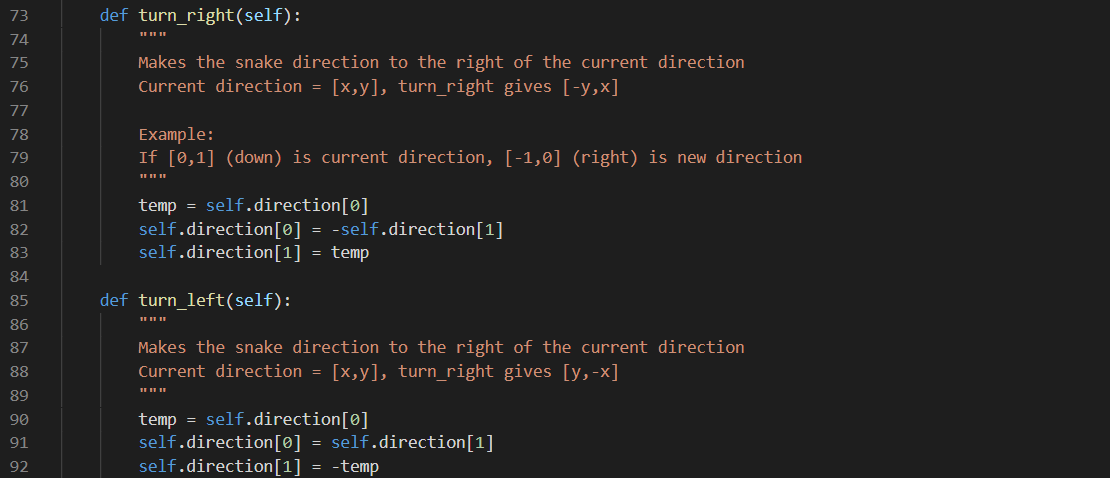


***snake.py* file**

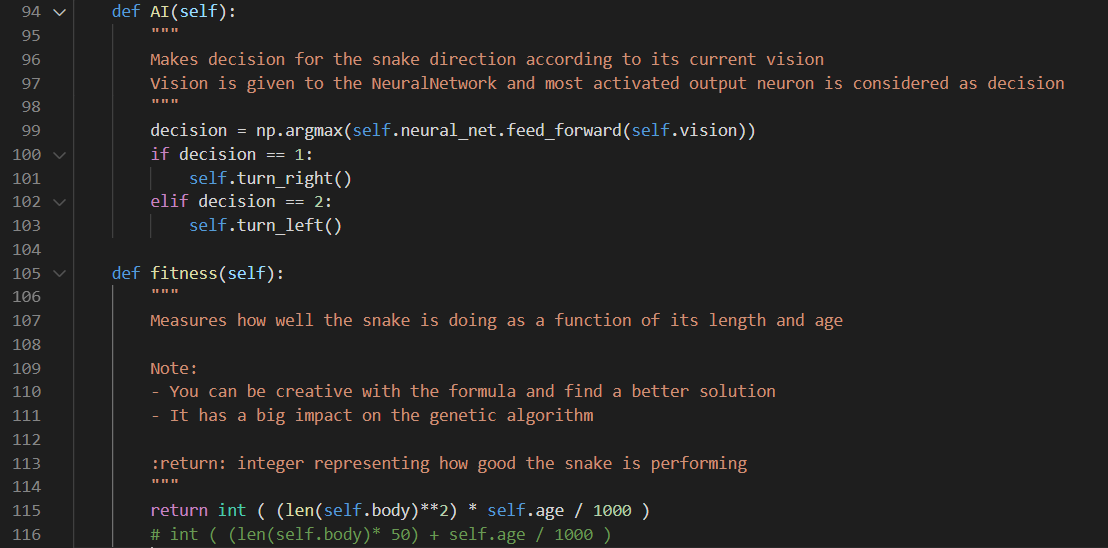
In *snake.py* file, a Snake object has been declared with varies parameters to represent its head and body positions, movement direction, survival age , starve (the value will decrease overtime until 0, then snake is dead and game ends), the neural network object for snake’s AI and vision object for neural network’s input.



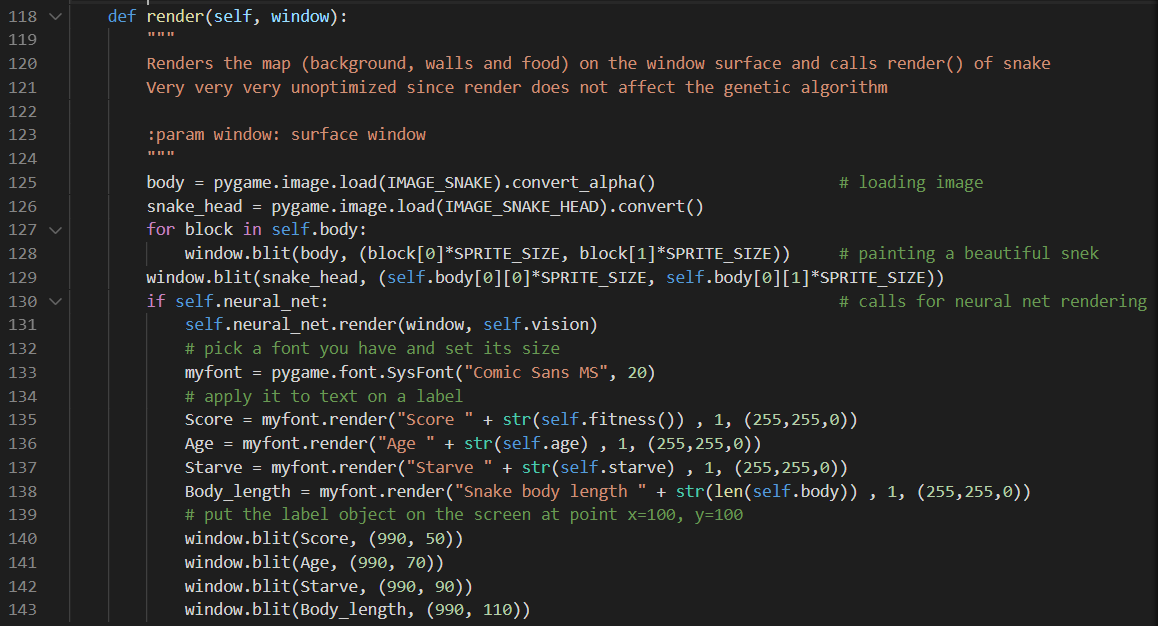
*turn\_left(), turn\_right(),update(),grow() and move*() functions let control snake movements and varaibles updates in the game.



The *AI() f*unctions let snake make decision that either turn left or right based on its neural network’s output for each step. The *fitness()* function returns a formula to calculate the performance for each neural network, which is crucial for the result of GA. As the selection of good populations for evolving in each generation are largely depends on this value , we have also tried out a couple of different formula’s to test out the performance for this GA algorithm.

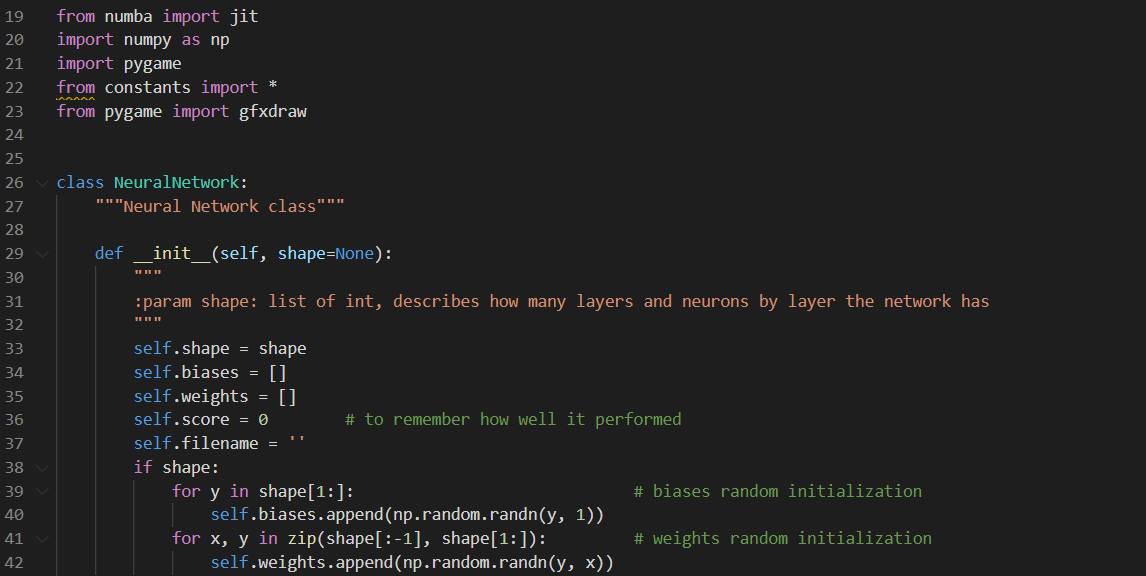


Lastly, the *render()* function will visualise snake movement in the game interface. In addition to the start-up code, we have also imrpoved snake’s graphics and user interfaces that will display the latest figures for the snake game in the game interface, i.e. Score, Age, Starve and Body Length values.

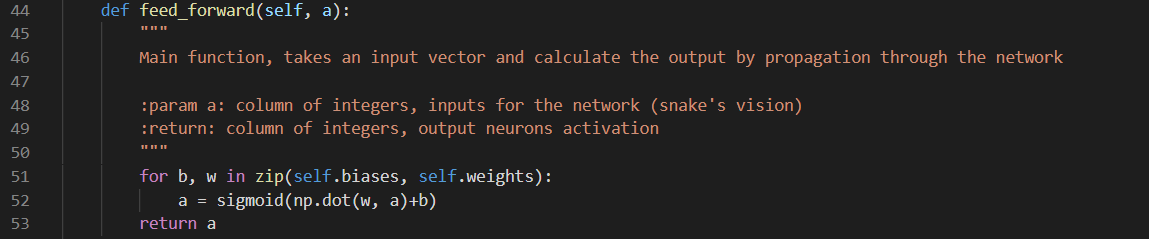


***neural\_network.py* file**

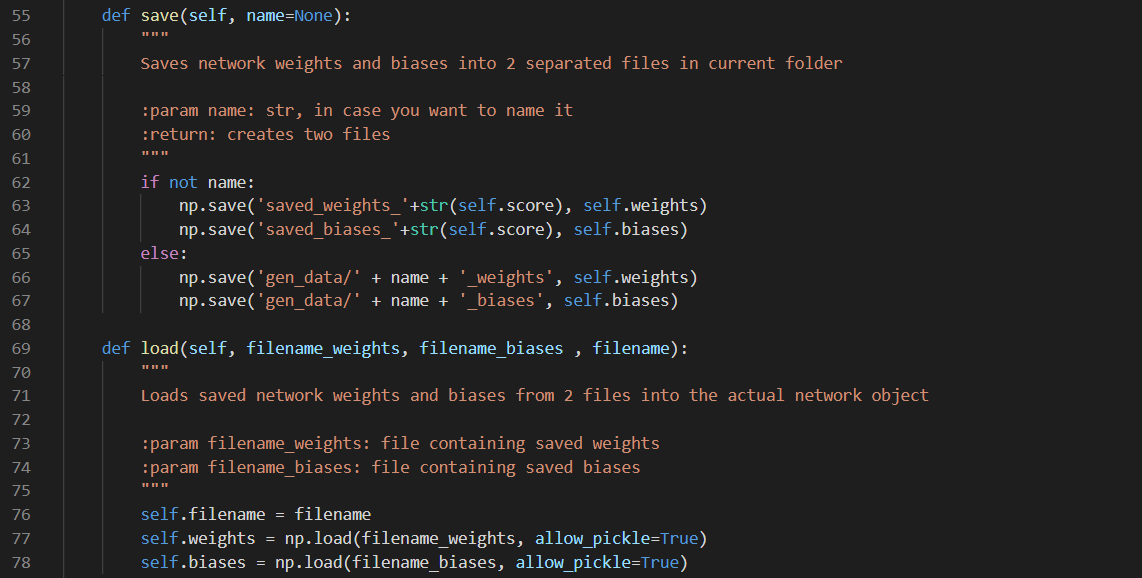
In the *neural\_network.py* file, it declared a NeuralNetwork object which is a single hidden layer architecture contains 1 input layer, 1 hidden layer and 1 output layer. The weights and biases values that connecting input , hidden and output layers are initialised randomly. The NeuralNetwork object has *feed\_forward(), save(),load()* and *render()* functions, which is used to compute snake’s movement , save and load trained neural network model, and render the neural network’s live activations in the game interface. As the neural network will be trained with GA algorithm , there is no backpropagation function required for this object.



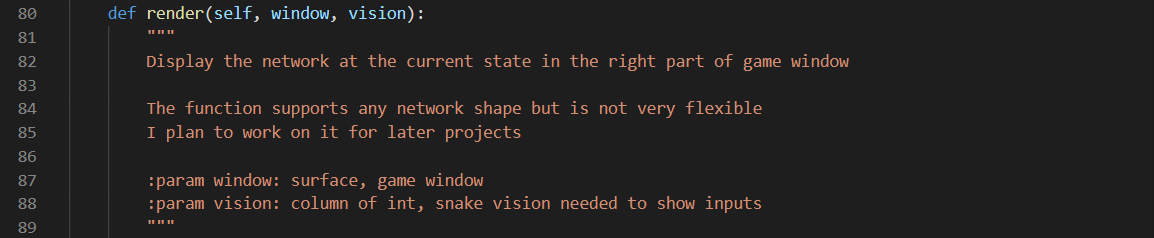
The *feed\_forward()* function uses classic sigmoid function for neural network activation.



The *save()* and *load()* functions can save and load copy of neural networks weights and biases in the file directory.

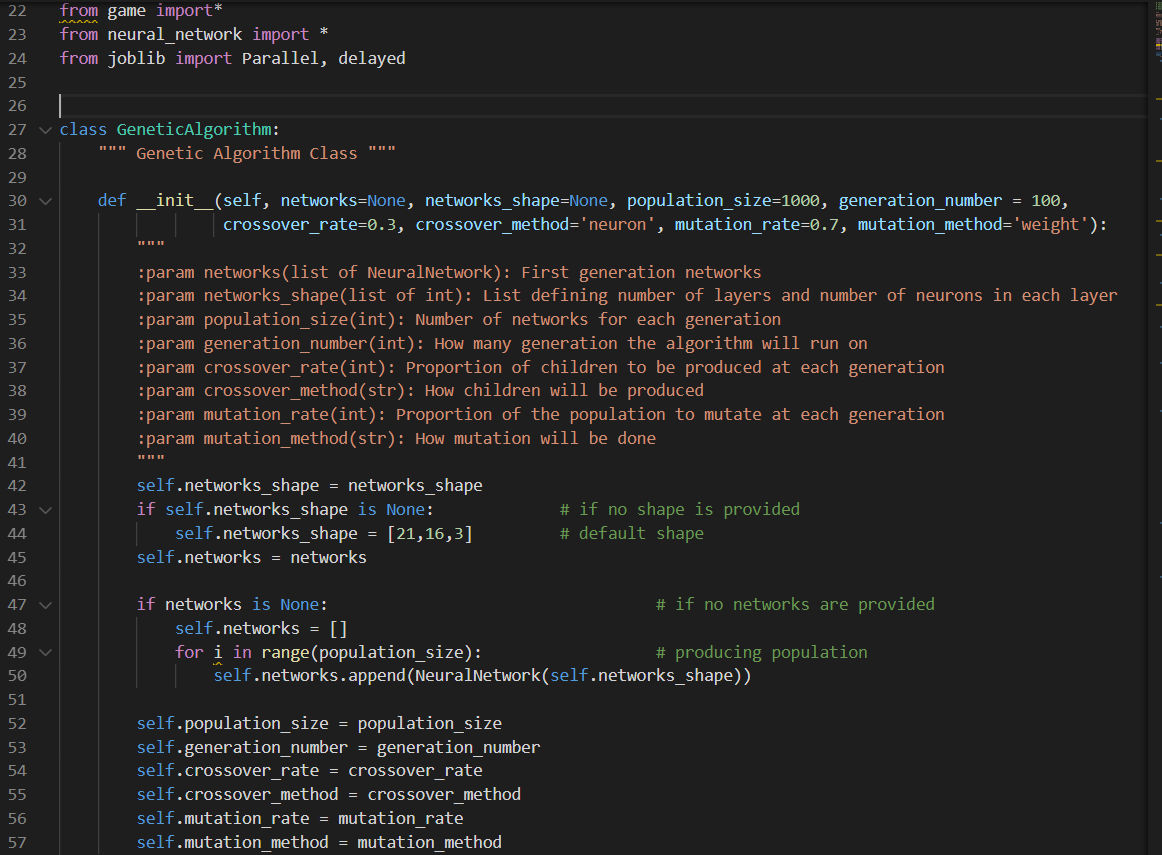


Lastly, *render()* function will visualise the neural network latest status in the game interface.

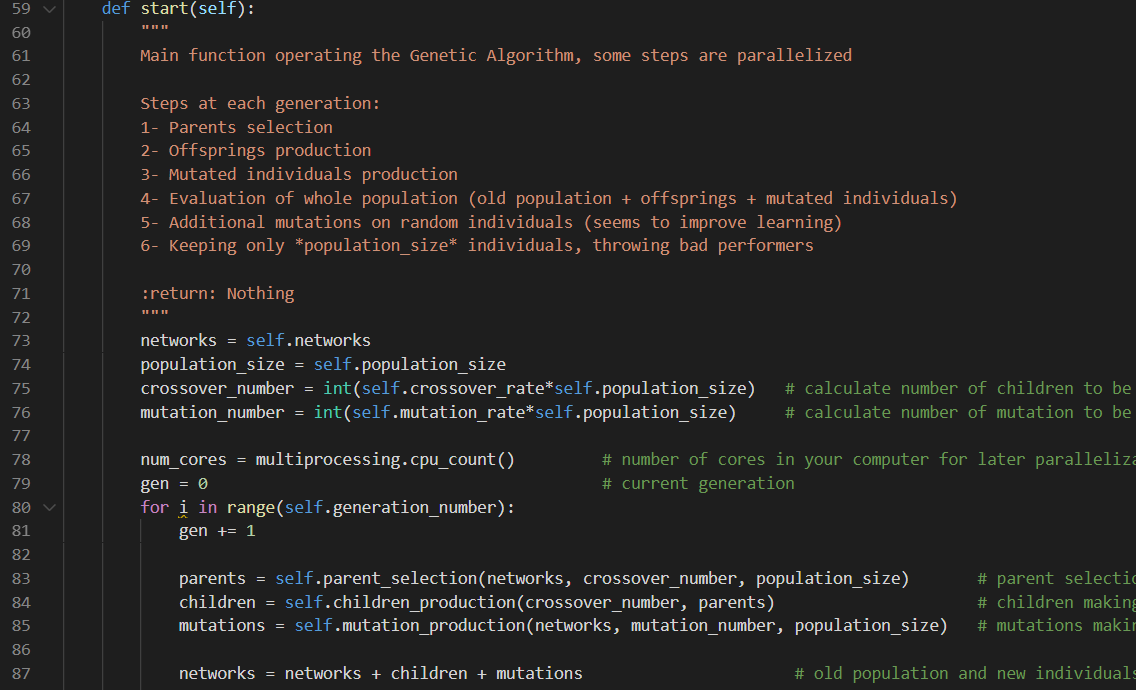


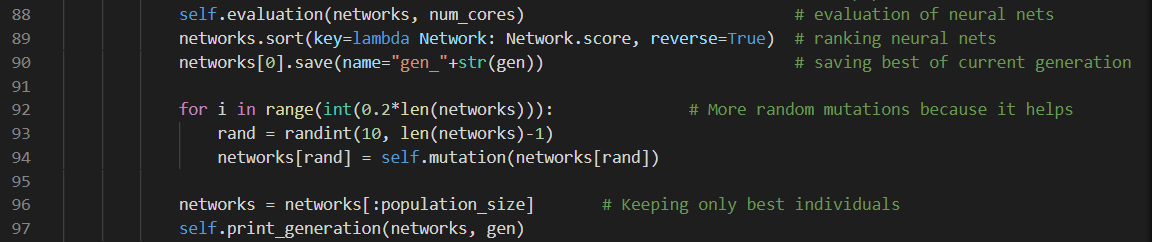
***genetic\_algorithm.py* file**

The *genetic\_algorithm.py* file contains GeneticAlgorithm object that including all the details for training snake’s neural network with GA. It is initialised with variables for population size, generation number, crossover rate and method, mutation rate and method. This object has start(), parent\_selection(), children\_production(),crossover() , tournament() , mutation\_production(),mutation() and evaluation() functions implementing GA algorithm through populations crossover and mutation in each generation. In this project, many neural networks (21 inputs, 16 hidden nodes and 3 outputs) with population size number will be initialised for snakes to play the game.

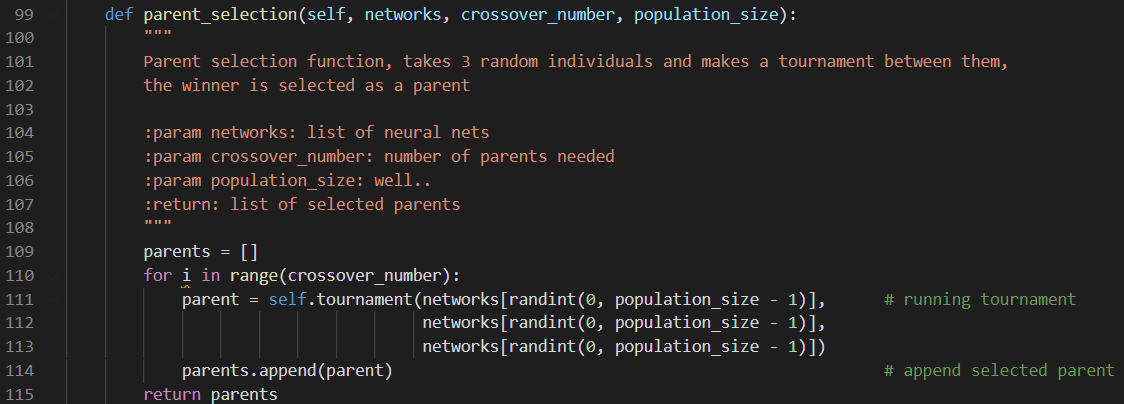


The *start()* function has the procedure for entire GA algorithm, it startes with initialising crossover\_number, mutation\_number and population\_size. In each generation , it will choose parents to produce children through *parent\_selection()* and *children\_production()* functions in order to cross over at different layers for different neural networks. A few neural networks will also mutate at one of their layers thorugh *mutation\_production()* function. Once extra neural networks have been produced through cross over and mutation processes , they will be put into the entire pool for evaluation by *evaluation()* function which run multiple snake games at the same time. The neural network with best fitness will be saved at local directory by *save()* function. Then, a few extra mutations will be processed to improve evolution for the entire generation.The top fitness neural networks , which is the population size number , which be selected for the evolution of next generation.

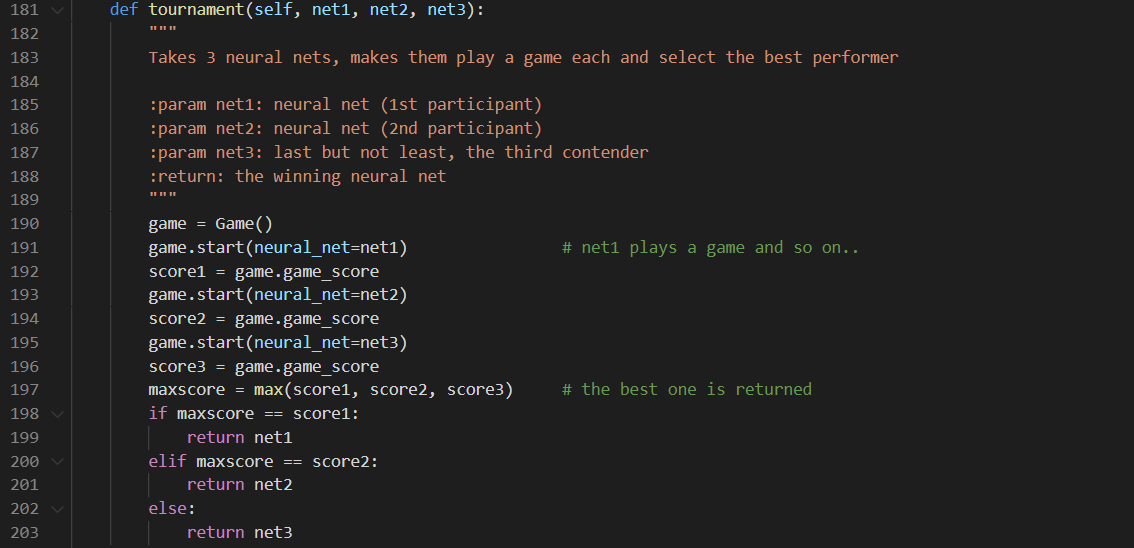




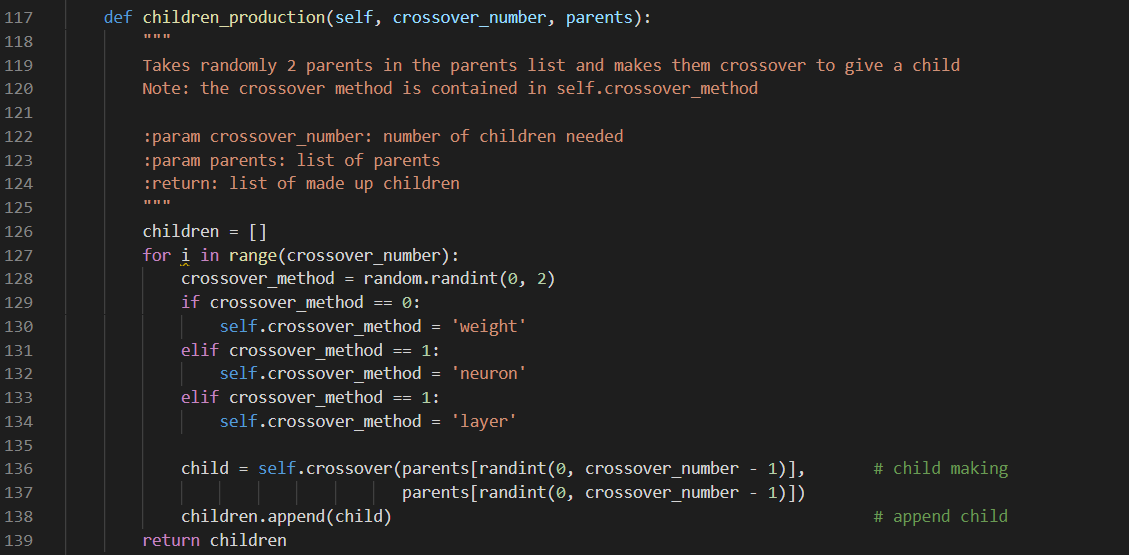
The *parent\_selection()* function will randomly choose 3 neural networks from the population pool for competition by tournament() function and the best fitness network will be kept as the parents for crossover.



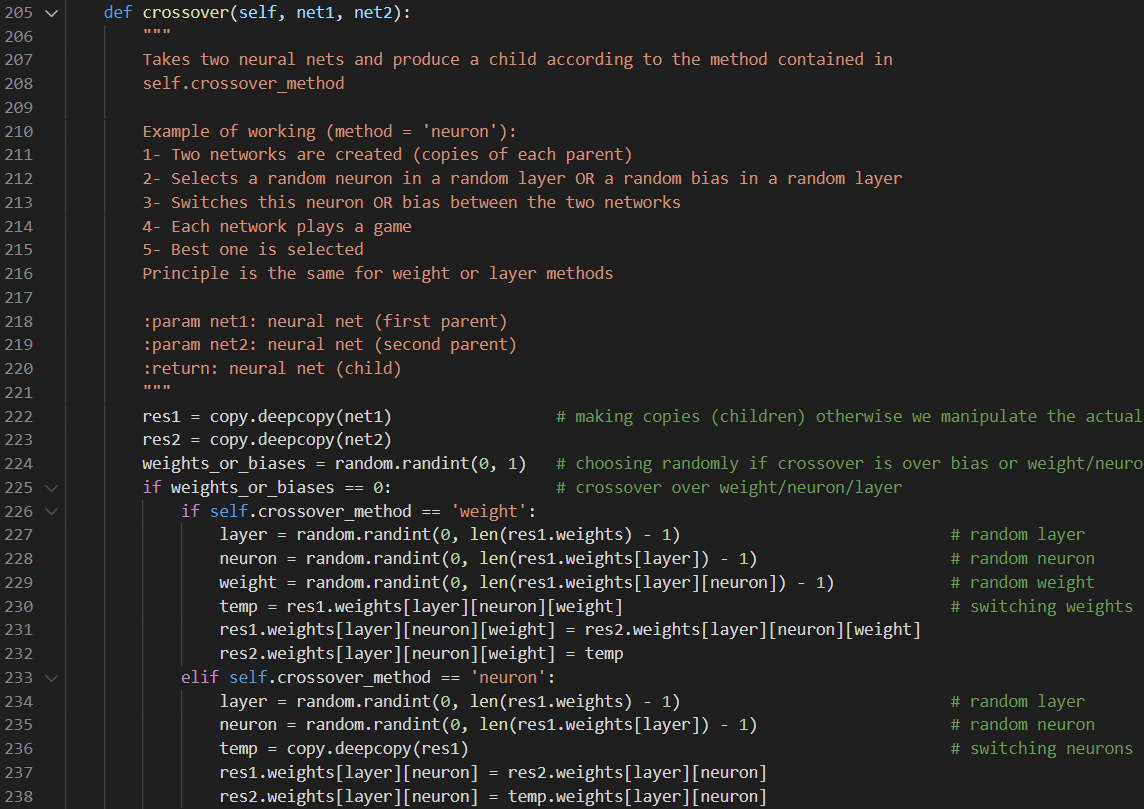
In *tournament()* function, 3 snake games with different neural networks will run sequencially and the best one will return among them.

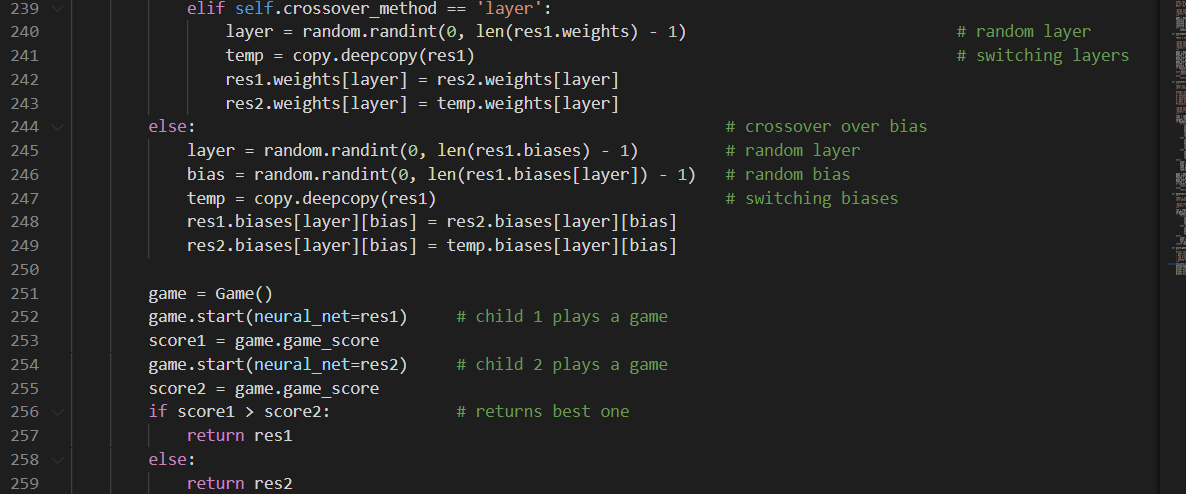


Once the parent neural networks have been selected , they will start cross over by pairs in *children\_production()* function. In this algorithm , two of them will crossover at weight, neuron or layer between their network to produce a child network. The total number of children networks is *crossover\_number* input varaible.

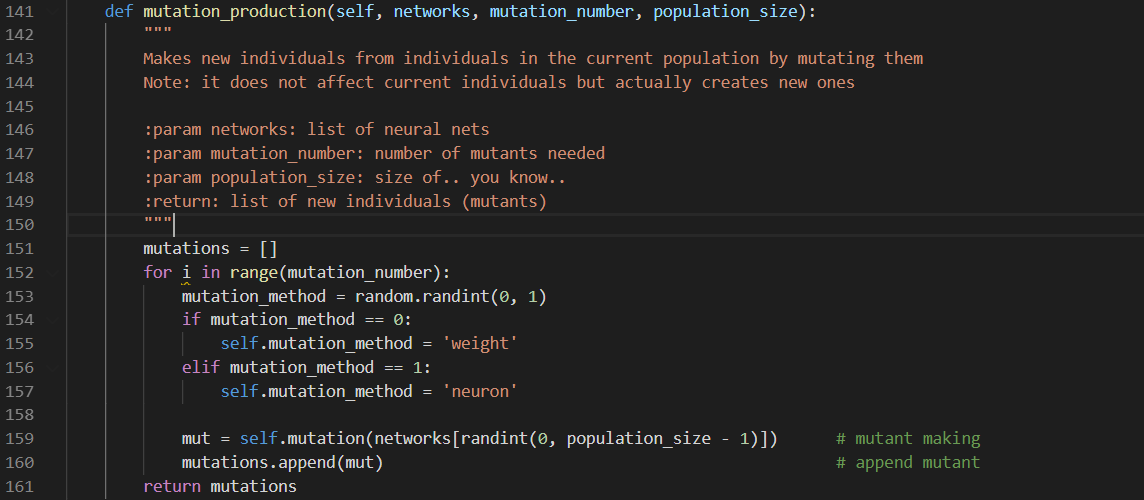


The *cross\_over()* function will crossover 2 neural networks based on thecrossover\_method variable of the GeneticAlgorithm object. One of weight, neuron, layer and bias value at 2 neural networks pairs will be exchanged to produce two new chidren from their parents. Lastly, these 2 children will run 2 games and better fitness network will be returned as the child in this function.

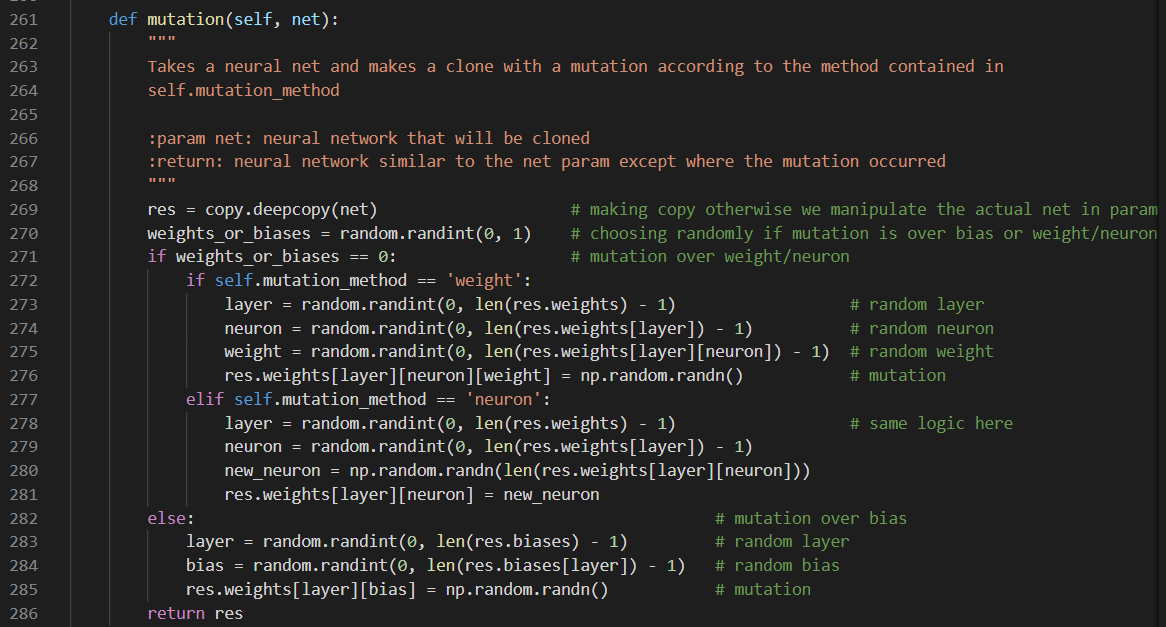




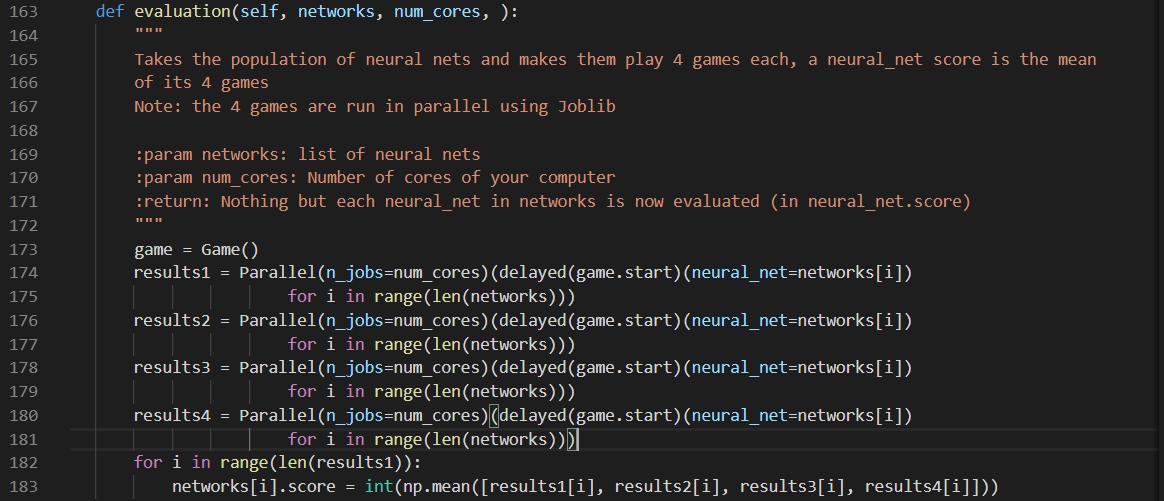
The *mutation\_number* networks in population pool will be chosen in *mutation\_production()* function. The selected networks will mutate at either weight or neuron randomly in the networks. A list of networks will be returned after mutation process.



The *mutation()* function will choose one of weight, neuron and bias value to mutate based on the mutation\_method varaible in the object. The chosen component in the network will be assigned a new random value.



Lastly, children and mutation population will be appended to the original population for evaluation of their fitness by running *evaluation()* function. It takes population of neural networks and makes them play 4 games each, the mean score of these 4 games will be added into the network array which is the network population pool. Also, there are parallel processes to accelerating the evaluation process based on *num\_core* variable which is the number of CPU core for the machine running this program.



**-User guide for the demonstration codes**

**-Demonstration codes with comments.**

**-Strengths and limitations**

There are lots of good things about genetic algorithms, and they work amazingly well a lot

of the time. However, they are not without problems, a significant one of which is they can

be very slow. The main problem is that once a local maximum has been reached, it can

often be a long time before a string is produced that escapes from the local maximum and

finds another, higher, maximum. In addition, because we generally do not know anything

about the fitness landscape, we can’t see how well the GA is doing.

A more basic criticism of genetic algorithms is that it is very hard (read basically impossible)

to analyse the behaviour of the GA.We expect that the mean fitness of the population

will increase until an equilibrium of some kind is reached. This equilibrium is between the

selection operator, which makes the population less diverse, but increases the mean fitness

(exploitation), and the genetic operators, which usually reduce the mean fitness, but increase the diversity in the population (exploration). However, proving that this is guaranteed

to happen has not been possible so far, which means that we cannot guarantee that

the algorithm will converge at all, and certainly not to the optimal solution. This bothers

a lot of researchers. That said, genetic algorithms are widely used when other methods do

not work, and they are usually treated as a black box—strings are pushed in one end, and

eventually an answer emerges. This is risky, because without knowledge of how the algorithm

works it is not possible to improve it, nor do you know how cautiously you should

treat the results.

**-Conclusion**

**-References**