A description of your simulation. This just needs to specify and explain choices of the grid size, number of turns per simulation, the world type etc.

For my grid size choice, I decided to keep the grid size as the default 24 as it was convenient, World type: I decided to keep the world type as world 1 mainly for simplicity and due to my ability I felt as if I could get something working in world 1 and I still had time, I might give something in world 2 a go, but world 1 proved to be a bit of a mission for me. For the number of generations, I found that for my implementation, 450 generations showed a positive trend in survival rate for the most part. This was paired with a number of turns at 150.

A description and reasoning for the choice of the model of the agent function.

The agent function is rather straight forward. If percept[4] has food on it that is safely edible (red strawberry), eat the food regardless of energy. If the percept has food on it but it is not food that is ready to be eaten (green strawberry), I set up a random uniform distribution between 0 and 1, and if the resulting number is greater than 0.7, eat the green strawberry. This doesn't happen very often as the odds of getting a value over 0.7 are 0.3, and there also has to be a green strawberry under the creature.

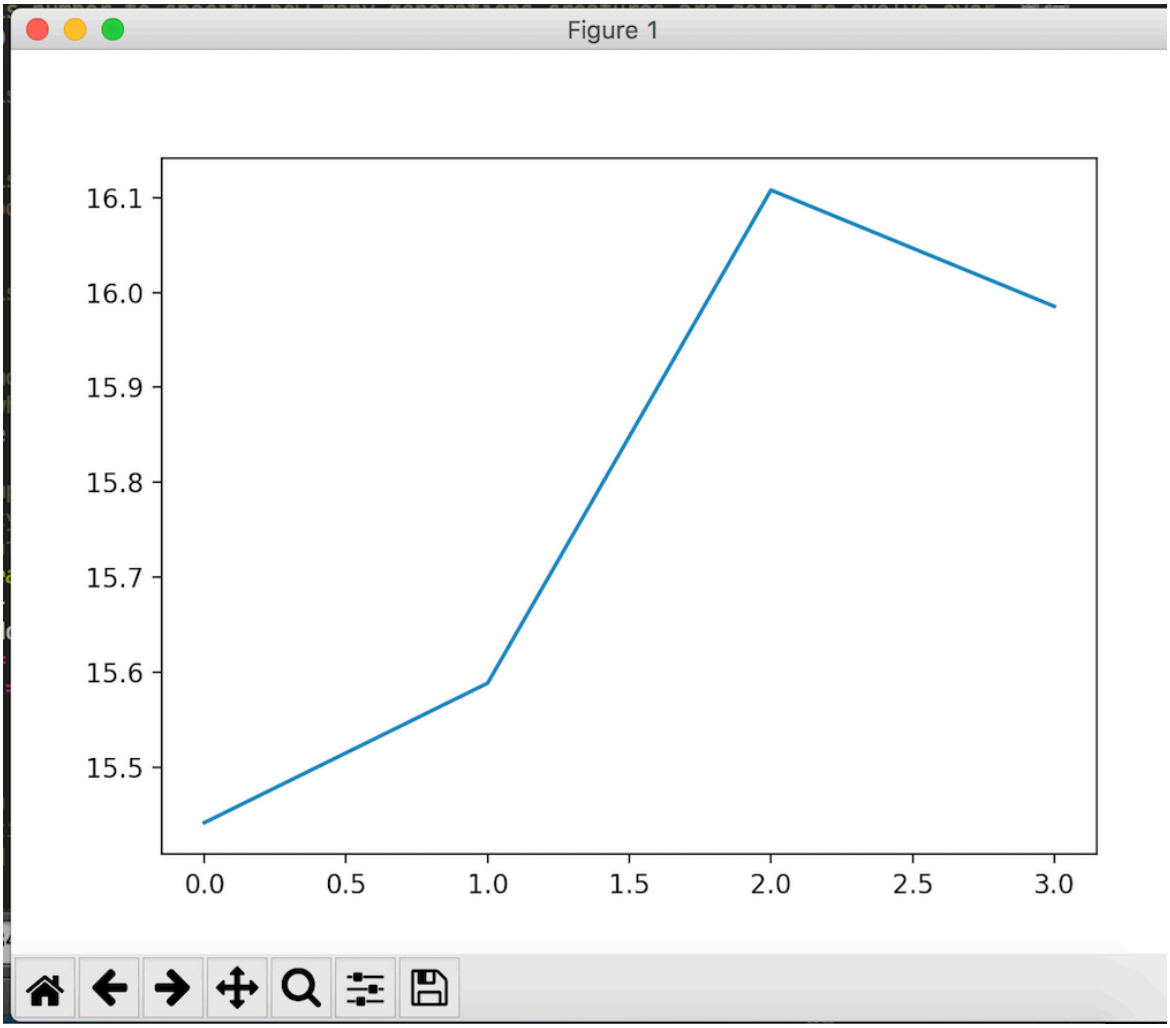
The rest of the agent function is about avoidance, and how to react when there are monsters within the area around the monster. By connecting actions to chromosome values, I make the creature move in the opposite direction to which the monster is in so that it can stay as far away from the monsters as possible, and try and have the creature survive as long as possible. An example of this is if a percept[0] == 1, then a monster is at the bottom left relative to the creature, so the actions array has actions[8] set to self.chrome[8], so the creature knows to move to the top right most tile from its current position.

A description and reasoning for the choice of the chromosome that governs the model parameters.

By default, the chromosome was initialised to the random values, with the values being reassigned based on whether the creature had to either eat, and what it was eating, or the way in which it had to move in relation to the monsters around it.

The reasoning behind this was because it was a good way to get the creatures working to where they would eat what is below them if it is food that is able to be eaten and weight the choice as to whether or not it should eat an unripe strawberry, or avoid monsters in order to survive.

A graph showing how average fitness of the population changes as evolution proceeds.



The values on the left of the graph is survival rate percentage per generation, and the values along the bottom are the generations grouped into 100s. So 0.0 is the first 100 generations, 1.0 is generations the next 100 generations average and so on.

A description of your genetic algorithm - the method and parameters of selection, cross-over and mutation.

The parent selection was done through the use of tournament selection. I create a 2D list that is the size of old_population and add the fitness and index of the individual to the list. This allowed me to easily reference the creatures, as before they were stored as objects which makes it rather hard to reference to breed from. From here, the 2D list was then used in the selection of five random creatures from old_population. This sub-selection of creatures was then sorted, and the highest two of those five random creatures were chosen to be parent and parent1 for the new offspring to be created from. The offspring's chromosome was default assigned to one parent's chromosome, and then a random number between 0 and 10 was selected to be the crossover point. From 0 up until this crossover point, the offspring's chromosome was set to be that of the other parent. The offspring of these two creatures was then created with numCreaturePercepts, numCreatureActions, and its new chromosome. This was then appended to the new_population list that would be returned at the end of the newPopulation method. This was repeated for the size of the whole of old_population, in this case 34, so that each generation would have all new creatures each time, based on the survivors of the previous generation. As for mutations, I couldn't get mutations to work well and I ran out of time, and since it has to be working code that gets handed in, I decided to leave it out.

A discussion of the results and how the evolution shaped your creatures' behaviour.

The above results are at 450 number of generations, and 150 number of turns per generation. In the beginning, with the low survival rate, there would have been a small amount of survivors. This graph shows a low starting survival value, with some learning over the generations, with some regression near the end, but there is some learning in there. These were the smartest creatures, and from these creatures, my various methods that I implemented got them to breed from what should have been a good mix of random selection from those who had good genes and survived, whilst not relying on two elites and just breeding heavily from them. The final average survival rate over 450 generations was 15.9215682745109%, or 16%, which I am semi happy with. I would have liked a higher survival rate, but the more turns that occur per generation make getting a higher survival rate harder to achieve.

Compilation note: I did have some issues with getting it to work on my personal machine with the 343 environment but going through terminal worked fine. I tried it on a lab machine when trying to fix another issue I was having and it did work fine then, but just on the off chance that it didn't work in PyCharm, it should work through terminal.