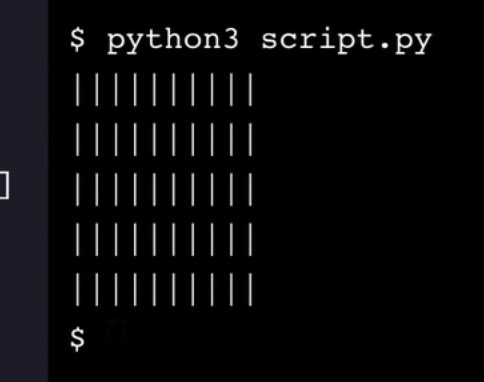
Tarek El-Hajjaoui

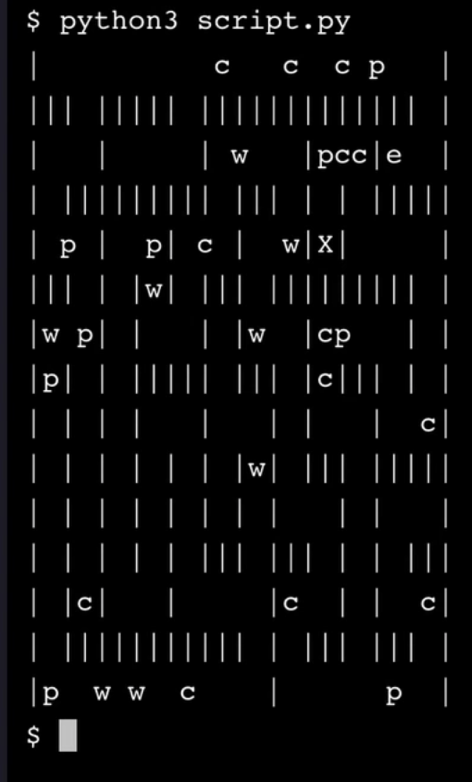
12/4/2018

**Codecademy Final Corn Maze Project Presentation**

For this application, we created a maze that moves two cells at a time. Before we started writing code related to the movement of the “X”, the Kernel Maze looked like this:

****

Then we integrated our move functioning code, which made it so that the X moves two cells at a time. The reason we did not move the X one at time is because “every single wall would be removed, and it wouldn’t be so much of a maze as an empty field” (Codecademy). If the code was designed to move the X three spaces each time, it would “make for a more sparse maze” (Codecademy). In other words, the maze would have to be designed differently to accommodate for that type of movement because otherwise the X would destroy walls, or the maze already made would even be too small. Two moves per pass by the X perfectly prevents the X from destroying walls and it is a complimentary amount of space to move within the given grid. This also demonstrates the importance of consistency. It is vital the X consistently moves at two spaces every time to prevent destroyed walls.

****

Above is a picture demonstrating what the maze looks like when completed. When the script runs, an ‘X’ is randomly populated within the maze. This X represents the starting point of the maze. The objective of an algorithm for this maze would be to find a path for the X to travel to its goal, the “e” (also randomly inputted somewhere in the maze). There are multiple algorithms that could be used to solve this maze.

A simple algorithm, linear search, can be used to solve this problem. The way a linear search (or sequential search) algorithm works is by sequentially checking whether a value of an element (from a list of elements) matches a given value. Linear search checks every element in a given list in order from start to end until it finds the given target value. To solve our problem, we could design a linear search code that sequentially reads through every element in the Kernel Maze to search for the “e” (represents our end goal). The program would then be able to tell which exact coordinates represent the “e”. However, we would need more search algorithms to finish solving the problem. Next, we could integrate code that uses a search algorithm to find all of the “|” (represents walls) positions so that we can discover which exact coordinates for the “X” to avoid moving into. Another search algorithm could be integrated to search for any letter except for “e” to find the positions of the random letters of the maze (which represent swag for the X to pick up as it moves in the maze), but it is not necessary. The program would then know which coordinates it cannot possibly move to (the wall), which coordinates it could travel through to get swag (the letters), and which exact coordinate it is intended to reach (the “e”). The problem can be solved by the program.

An alternative solution would be to use a binary search algorithm to solve our Kernel Maze problem. Binary search is contingent on the given data to be sorted prior to the algorithm operating. The reason being is that binary search takes advantage of the ordering of sorted data to solve problems more efficiently than a linear search algorithm. With the given scattered Kernel Maze, it would be vital for us to somehow sort the array of data in a manner for us to use binary search. One thing to consider when manipulating data to be sorted is to maintain immutability. In other words, if we want to sort the Kernel Maze, we should not directly manipulate the Kernel Maze data. Instead, we should make a new copy of the Kernel Maze data. With immutability in consideration, the first step would be to sort the Kernel Maze in manner such that the walls would be grouped together so that the algorithm can focus on sorting through the letters. If we can organize the Maze data list in a manner that does not remove the wall data, but somehow allows the program to ignore it, the program can focus on the letters next. We would then have to define to the program how ordering of letters (in alphabetical order) is meant to be. Once the program understands alphabetical order, we can use binary search to sort the array of letters with the consideration that the “e” is our target. The binary search algorithm would then run until it finds the e target.

The time complexity for linear search is O(N) and the time complexity for binary search is O(log N). Although in general binary search is more efficient than linear search, it is not necessarily the right choice. A programmer would have to take in the considerations discussed above to use a binary search algorithm to solve this Kernel Maze problem. I am confident that it is possible to write a code that takes in those considerations and works, but actually writing the code could pose more unforeseen issues such as a complicated code to maintain or dealing with bugs.

A search algorithm was used to generate the Kernel Corn Maze. It is helpful to understand which algorithm was used to generate the maze in order to determine the best algorithm to solve it. This is the case because since we know that a search algorithm was used, we know that our maze is a list (or array) of unsorted data consisting of walls and letters. Knowing this simply makes it obvious that it would be optimal to use a search algorithm to search for our target, “e”.

As the “X” is moving through the maze, it will collect some swag items (the letters other than e). One method to store the swag items collected would be to first define a variable that represents our swag items (maybe called ‘swag\_list’). The variable will initially equal an empty array. As the X grabs a swag item, the program will know to know to push the swag item into the swag\_list array.

If the farmer were to ask me to sort the items collected before leaving the maze, I would consider using a quicksort algorithm. The quicksort algorithm “is an efficient recursive algorithm for sorting arrays or lists of values” (Codecademy). This algorithm is ideal for an array with many values that are out of order. The worst case runtime for a quicksort algorithm is O(N^2), but quicksort is an unusual algorithm and normally is thought of in its average runtime, O(N \* logN). Given a criterion of how the farmer wants the items sorted, the quicksort algorithm can be used to sort the data.

The quantity of the data of the Kernel Maze list must be taken in consideration. If there is not so much data to be sorted, a quicksort algorithm may not be the best choice, but perhaps a linear search would be more efficient and therefore more ideal. The type or variety of swag must also be taken into consideration because certain algorithms may be less efficient given a large variety of swag data. If there are many different types of data to be sorted, the quicksort algorithm may not be ideal because it is a comparison algorithm and therefore would require some sort of metric (in the code) to compare the different types of data.

**Side note:** *I was unable to make my maze code functional despite me carefully following the walkthrough numerous times and carefully looking for mistakes. I am a MERN Full Stack Developer, so I learned Python just to take this course. For me, it is much more important to focus on the concepts. I hope you can see from my write up, that I am learning and understand the important material. As for my code, I believe that I simply have a problem with my spacing (or perhaps syntax) that I cannot find. If I need to fix this to get the certificate, please let me know. Thank you!*