PROJECT NAME - Where is Ada Lovelace?

Project Report

TEAM MEMBERS: Shayli Patel, Nosheen Masud, Lana Moroney, Michaela D'Mello, Catherine Phillips (Team Name - Globetrotters)

INTRODUCTION:

Aims, Objectives and Inspiration

Our team - The Globetrotters - have created an educational entertainment game for our Nanodegree final project, which we have named "Where is Ada Lovelace?". It is a walk-through text adventure game in Python, inspired by the 1985 computer game "Where in the World is Carmen Sandiego?", a popular mystery exploration game. The aim of our game is to find Ada Lovelace and find pearls of wisdom only she can provide. The objective of our game has the player travel to a city where they can find up to 3 clues at different landmarks, which direct them to the next location the protagonist, Ada Lovelace, has moved on to.

BACKGROUND

As the project is a text adventure, walk-through game, a lot of consideration was put into the logic and understanding of the rules for the game. Figure 1 demonstrates a flowchart of the logic of the location of the user in the game. The game begins in London, UK and the player is tested on their location-specific trivia skills throughout the game. The player can find up to 3 clues at different landmarks to determine the next location. If the correct location is deduced, the player is transported to the new location. If the player enters an incorrect location, they will be transported to that chosen location, and the clues will lead them nowhere, indicating they're in the wrong location. The player has the option to pick a different location - until the correct next location is selected.

This logic would continue throughout the game, until the player reaches the final, correct location where they would find Ada Lovelace.

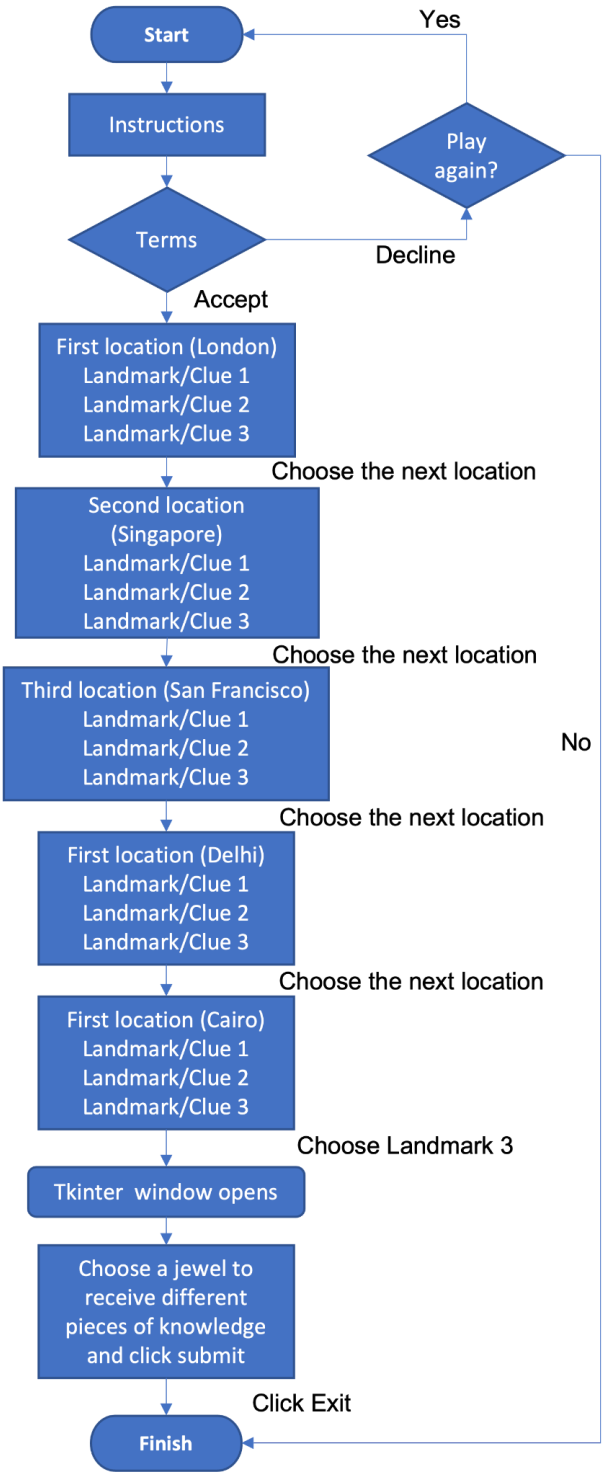

Figure 1: The logic and rules of the game were incredibly important aspects of the game and hence much consideration was put into such areas. The game was decided to start in a certain location. After choosing the correct location, the game displays 3 landmarks for the player to investigate. Each landmark contains a clue, which the user uses to determine the next location.

SPECIFICATIONS AND DESIGN

Requirements for the Project

We decided to programme the game in Python to practice and showcase the skills we have been learning throughout the Nanodegree. For this reason we chose to focus on the back end of the game, as there was a mutual agreement of the appeal of the simplicity of a text game in Python.

To ensure the code is clear and tidy, we decided to adopt multiple functions and classes to ensure the code was easy to read and to avoid repetition.

We also implemented decorators into the game to emphasise parts of code that needed attention and enhance the user experience. Simple decorators are used for designing the output text which the user reads when running the game.

Design and Architecture

```
./Globetrotter_CFG_Project/
├── front-end/
│   └── globe.gif
├── functions/
│   ├── config.py
│   ├── decorators.py
│   ├── functions.py
│   └── location_data.py
├── tests/
│   ├── test_decorators.py
│   └── test_functions.py
└── main.py
```

Figure 2: The directory tree of the project can be seen to help visualise the design and architecture of the project. Multiple files were used to organise and tidy the code into appropriate files. The subdirectory 'tests' was used to ensure effective source tree directory structure.

The game would run inside the main.py file. Other files would be found in the repository of the project to contain specific segments of code that are imported into the main.py file. Figure 2 demonstrates the architecture of the project. This ensures the code is clear, concise and tidy. For example, the file decorators.py is used to contain decorators. By using this method, functions from different files can be imported into the main.py file. Figure 3 illustrates a section of the code inside main.py which imports functions from different files.

```
from functions.functions import introduction_qs, winners_graphics_box
from functions.decorators import instructions, london_welcome, san_fran_welcome, cairo_welcome, delhi_welcome, singapore_welcome
from functions.location_data import correct_route
```

Figure 3: Main.py is the file which is run for the game. Within the project repository, there is a subdirectory called functions. Inside the directory functions, there is a file called functions.py, location_data.py and decorators.py. The functions from these files are then imported into main.py.

IMPLEMENTATION AND EXECUTION

Development Approach

When developing the game, it was important to initially create a simple skeleton of the code. The skeleton allowed the player to start the game, be in a certain location, have the input choice for the next location, and a way to finish the game. This basic skeleton was an important first step to ensure everyone has a basic idea of how the game would be presented and developed.

After this simple skeleton, different roles were given to members of the team to ensure the development was efficient, and to ensure that team members could use their strengths when creating the game.

Initially, we discussed that functions and lists would be in the main body of the code. After first implementing this, we then decided to use classes in order to reuse and limit the repetition of code. We briefly considered using inheritance classes, however, ultimately we decided they would not suit our needs. We created objects by hard-coding the location data and used an API within a class method to read live weather data. Dictionaries and lists were then incorporated as part of the instance to eliminate these from the main body of the code and to minimise the importing of code into the main document, thus ensuring the DRY and KISS software development principles.

A third layer of development was to add GUI applications to the game using the tkinter library. We initially wanted to add this to the whole game, but due to time constraints we decided to add it as a celebration once the player has won the game, and in awarding them a reward for doing so. In creating the celebration of the win, a GUI application main window was created within a function. Various widgets were then added to this, with various dimensions, colours and fonts, plus a gif image. To keep it interactive, the reward for the win is to be chosen by the player, picking their prize from different jewels of treasure. The prize is a piece of advice to the player on programming. Each jewel has a different piece of advice, and a dictionary was used to achieve this.

Project Methodology

The project took an agile/scrum-like approach with weekly meetings. These weekly meetings were essential to discuss both progress and goals for that week, highlighting anything that had fallen behind or needed extra resources. These meetings were crucial as the methodology approach resulted in different members of the team working on different sections of the code, ensuring everyone was up-to-date with the project. In the last week, in anticipation for the fast approaching deadline, we held a daily scrum to update the group on progress, issues being faced and prioritising tasks to completion.

Changes:

When the project started, there was one large function inside the main.py file called start(). Within the start() function, a nested function (main_logic()) was coded inside. Initially, a few

errors would arise from this code (particularly when the play_again() function was called), so it was concluded that it would be best to break up the start() and the main_logic() functions. This resulted in the code running without errors, and it broke up the code neatly into subroutines, to illustrate the introduction separately to the main logic.

When importing functions into the main.py file, one issue which occurred was where the imported functions and decorators would automatically run without being called. The code was corrected by adding a statement into the file in which the function is defined, “if __name__ == ‘__main__’:”. The functions could then be called. This is seen in figure 4.

```
if __name__ == '__main__':
    instructions()
    london_welcome()
    san_fran_welcome()
    cairo_welcome()
    delhi_welcome()
```

Figure 4: When importing functions into the main.py file, sometimes they would automatically run. To avoid this issue, an “if __name__ == ‘__main__’:” line of code was added to avoid this issue.

Challenges:

- Ensuring the player's input was accepted when the correct answer was given.

An issue found was that sometimes the correct answer - which was inputted from the user - was not always accepted. This problem is particularly noticeable with case sensitivity. For example, initially if the player inputted ‘Yes’, it would not be perceived as correct as Python is case-sensitive. This was resolved by converting the user’s input into the format we used in the code of the game. Figure 5 portrays how the .lower() and .strip() methods are used to convert the format of the player’s input into the same format as the code for the rest of the game to ensure this problem would not happen again.

```
if answer.lower().strip() == 'yes' or answer.lower().strip() == 'y':
```

Figure 5: Case sensitivity was a common problem, particularly for players inputting anything into the game when playing. To solve such a problem, the .lower() and .strip() methods are used as a function to convert the player’s input into a format which the code uses.

- If the player inputs a location which is not the correct answer.

There was a lot of consideration for the logic of the user selecting the wrong location for the game to travel to next. It was decided that if the player incorrectly selects a location, the player would be taken to this location. Once at this location, the user would be given useless clues. These useless clues would hint that the player was at the wrong location. The player proceeds to choose another location from the list. There are some issues and things to consider with this approach. To limit the number of tries a user may enter a location, it was decided a list of

locations would be provided so the user picks one from the list, instead of only guessing from the clues. This is more helpful for the player playing, to avoid them consistently typing the wrong location again and again, never progressing in the game.

The logic of the game was also considered for what happens once the player inputs the wrong location. Once in the incorrect location, 3 useless clues (i.e. “I haven't seen who you are looking for.”), will be provided. The player is then asked from the previous list of locations to choose from, where to head to next.

- Making the output of the game easier to read.

As the game was designed to be a text adventure, walk-through game, with the game running through the output terminal. It was essential to ensure the game was clear and easy to follow along. To ensure this was maintained, decorators were used to wrap certain features around certain bits of text. Different techniques were included to ensure the presentation of the output terminal, including using ‘\n’ and importing the colorama library to highlight and use different coloured text.

TESTING AND EVALUATION

Testing strategy

Once code has been written, it is incredibly crucial to test and evaluate the code, to ensure all possibilities of the code works. User testing was undertaken to ensure the functions used worked. This resulted in more information and evaluation of when exceptions and error handling could be raised within the code to ensure no errors popped up during the user running the game.

```
while True:
    try:
        choice = input('Where would you like to go? ').title()
        if choice.upper() == 'NEXT':
            break
        elif int(choice) in dict.keys():
            print('{}: {}'.format(list[int(choice) - 1], dict[int(choice)]))

        else:
            raise Exception
    except:
        print('Invalid landmark, please try again!')
```

Figure 6: From attempting user testing, it was rather noticeable where try and except statements could be added into the code to ensure no errors arose during the game. This was carried out, resulting in consistently running code.

Moreover, another important type of testing which was conducted was unit testing. Unit testing tends to be used to check what the function is returning, however for the project it was more suitable to check the output value. It was important to import io, sys, and unittest for the unit tests to take place. This would allow the program to create a StringIO object which can be redirected. For the functions which had

decorators, it was decided that ensuring the output has a value was more important. Hence, self.assertEqual() was used to check if the output was what would be expected.

CONCLUSION

Analysis:

One aspect of the project which went really well was the ability to work as a team in an agile way. It was advantageous to work in such a way where different team members were working on different parts of the code at the same time, as the short deadline for the project meant that the team would have to work in a time efficient way. To ensure this technique was successful, weeking (and then daily) meetings were held to update the team on what was done, and what was the next focus. We collaborated on our work over GitHub, initially starting with an ideas branch and then when we had a substantial amount of workable code, moving over to the main branch.

The project was a success, with the player being able to complete the whole game. In addition, we found our project developed our skills in programming and the software development life cycle, and allowed us to implement the concepts we have learnt. It also further developed skills in new areas such as the libraries of tkinter and colorama, which were external features outside of the course.

Extension:

One of the biggest benefits of doing a walk-through game is the flexibility to expand it. Given more time, there are several avenues to make the game more complex that we thought would make good extensions to our project:

- Increase the number of locations - a simple extension to lengthen the game.
- Use Tkinter for the whole game - instead of using the Python output, it may be interesting to use Tkinter to display the entirety of the game.
- Randomise the correct order of locations for every game so it's new every time - including the start and end location. Each time the user begins the game, the correct route to find Ada Lovelace would be different from before.
- After finding Ada Lovelace, the player can go on to find another person for advice to extend the game further. Or a villain character, who makes the game more difficult.
- Add an aspect of currency into the game. The currency would be used for the player to take flights and move from location to location. A currency API would be used to convert different currencies in different countries.