# The Icarus Protocol

*Project Solution Approach*

Lincoln Middle School


Collin Nelson;

Anna Ueti;

10.3.22

# TABLE OF CONTENTS

# I. Introduction

This document, Project Solutions Approach, serves as a comprehensive overview of the inter-relational design of the gamified project commissioned by Lincoln Middle School. It will focus on three main designs, Architecture, Data, and User Interface. For Architecture, each subsystem within the overarching design of the project will be thoroughly explored and explained, with a focus on the subsystem's concepts, algorithms, and interface properties. For Data, any data type and database interaction and properties will be diagrammed and evaluated. And for User Interface, each page will have a mocked version and a detailed description of components and functions.

The purpose of this document is to create a guideline for the developers to follow throughout the development phase of this project. The developers will be able to perform verification tests during development based on the detailed description of

i. Subsystems descriptions

ii. Subsystem relationships

iii. Data type objects

iv. Data type and Database relationships

v. Database schema

vi. User Interface Mocked pages

Additionally, another purpose for this design document is for stakeholders. With these designs created and documented within this solution, stakeholders will be able to cross reference the prototype with the intended goals and design outlined here.

Our team aims to explore the potential of games as a tool for education by producing a fully featured game designed to teach basic programming skills to students at a middle school or early high school level with no prior programming experience. While games of a similar nature exist, they often fall into one of two traps which our team sees as pitfalls. Some, while employing the surface level appearance of a game, fail to truly embody the game design principles that make games powerful for learning; Others use a proprietary scripting language that has lessened impact in teaching actionable programming skills. In the construction of this project (working title: "Icarus Protocol") we aim to solve this problem by producing a game that is a genuinely fun and interesting experience, while also serving as an effective tool for teaching real Python programming.

# II. System Overview

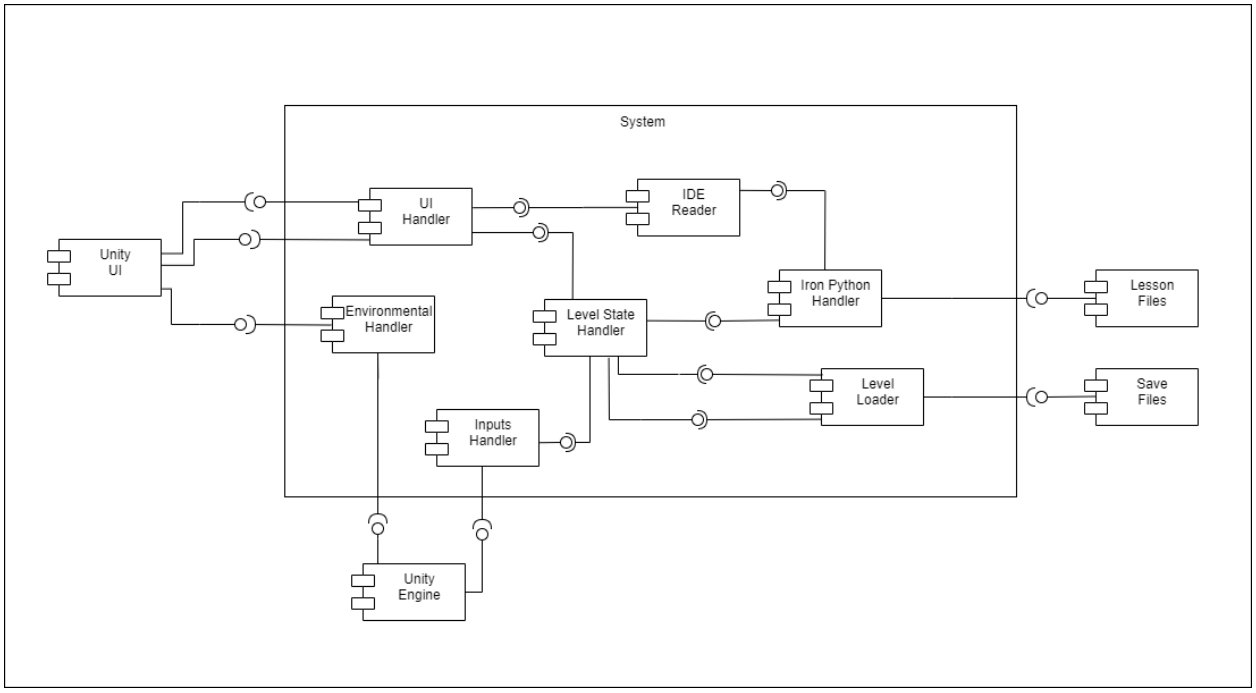
This project's main functionality surrounds the idea of teaching python to students with a novel and engaging experience, focusing on skill and knowledge retention through the avenue of

gaming. The functionalities of this project can be loosely divided into three major categories; User Interface and Python IDE, IronPython Integration, and Core Game Systems. Our design methodology is heavily influenced by the engine specific requirements caused by using the Unity Engine to drive development. The effect of this influence on our system architecture and component design patterns will be more thoroughly discussed in the Architectural design overview section. In regards to the design of our data structures, there are two major considerations that will be discussed. The first is the method of storing user progress in reliable and extensible save files. The second major consideration is the structure of our level data, which will have to be designed to be both highly flexible and easily modified without recompilation of the primary executable. The core consideration when designing our UI will be the strengths and limitations of the Unity UI system. While this system is flexible enough to be useful for our purposes, its limitations will have to be taken into account when creating UI layouts. The specifics of our component interfaces, data structures, and UI layouts will be discussed in the following sections.

# III. Architecture Design

## III.1. Overview

The Pluribus Doctrina Team has chosen to use an Object Scripting Model for this project. Since the application is developed in the Unity Engine, the architecture model must conform to the engine specifications. Given these requirements, the team has decided on an engine specific model that will utilize the strengths of the Unity Engine. Within the context of solely working on the game in the Unity engine, the team couldn't see another pattern that would take advantage of the Unity system as well. Below is a component diagram that illustrates the overarching architecture of our game. This diagram's purpose is two-fold, one, it serves as a guide for the developers to reference and base development around, and two, it provides a visual representation of the component architecture that is decomposed in the following section. Here is an overview of the components, as seen in the diagram below. Given the nature of working in the Unity environment, the player will start with the Unity User Interface. The Unity UI will then communicate with select system handlers. Such systems would be, one, the UI Handler, which will facilitate any UI element such as page layouts and icons, and two, the Environmental Handler, which modifies environmental properties such as camera positioning. From the UI Handler, there is a connection to the Level State Handler, which is the main subsystem that handlers the levels which the player interacts with. Level State Handler connects to the Iron Python Handler - facilitates the interpretation and simulation of the python code inputted and loads lesson files -, the Level Loader – saves and loads levels as the player progresses -, Inputs Handler – abstraction for user input from unity engine. Additional in-depth descriptions of these subsystems will be elaborated on in the following section.




## III.2. Subsystem Decomposition

### I.1.1. [UI Handler]

#### *a) Description*

The UI Handler subsystem manages any user interaction with the UI elements as well as UI details such as the displayed layout and icons. It handles inputs from the Unity UI external component such as actionable functionality that is tied to UI elements or player code entered into a UI element.

#### *b) Concepts and Algorithms Generated*

Similar to the structure of Unity's systems, the UI Handler is dispersed across many sub-classes. The most prominent of these sub-classes would be the UI-Layout class that dictates to the Unity UI which layouts should be displayed. The UI Handler also consists of the scripts tied to each UI elements' input events. In alignment with the Unity Engine's design philosophy, each button or UI element may have one or more atomic scripts which handle event actions. For the purposes of simplicity these are all considered to be part of the UI Handler subsystem. These classes will handle the interaction between the Unity UI elements and other subsystems.

#### *c) Interface Description*

Services Provided:

1. **Service name:** UpdateLayout
   **Service provided to:** Unity UI, Level State Handler
   **Description:** The UpdateLayout service will allow the Unity UI or the Level State Handler to call for an update to the page layout, this will occur for the transitions between the home

page, the level select page, the manual page, and the level page, with any variations due to level phase.

2. **Service name:** BundlePlayerCode
   **Service provided to:** IDE Reader
   **Description:** The BundlePlayerCode service will package the code written in a UI element into a compiled data object that is passed to the IDE reader, this will happen once the player has selected simulate.

Services Required:

1. **Service name:** ModifyUI to UI Handler
   **Service provided from:** Unity UI

#### I.1.2. [Environmental Handler]

***a) Description***

The Environmental Handler system manages all the interaction between the Unity UI and the application system. It also handles the interactions between the internal environmental objects and the Unity Engine. It will handle services pertaining to both the Unity UI and Engine systems, such as camera direction and scope.

***b) Concepts and Algorithms Generated***

Following a similar structure to the UI Handler, the Environmental Handler is decomposed into many sub-classes. However the one largest class is the Camera controller class. Given the strengths of working in a Game Engine, the game objects, such as layouts and UI elements, will always be on screen and the camera will display or obscure these objects as needed.

***c) Interface Description***

Services Provided:

Services Required:

1. **Service name:** ModifyCamera to Environmental Handler
   **Service provided from:** Unity UI
2. **Service name:** ModifyEvents to Environmental Handler
   **Service provided from:** Unity Engine

#### I.1.3. [Inputs Handler]

***a) Description***

The Inputs Handler's responsibility is to abstract the inputs form the Unity Engine from the internal application system environment. It will handle inputs from the Unity Engine and provide a sanitized version to the level State Handler.

***b) Concepts and Algorithms Generated***

Following the design philosophy of the Unity Engine, the Inputs Handler will also consist of several smaller classes. These classes will be tied to the script that is paired with game objects. The responsibilities of the Inputs Handler, while on any game object, is to act as a sanitizer for the raw inputs from the Unity Engine. It will then pass the inputs to the Level State Handler.

### *c) Interface Description*

Services Provided:

1. **Service name:** GetPlayerInputs
   **Service provided to:** Level State Handler
   **Description:** The GetPlayerInputs service will abstract the interaction process of the player's input and the internal application's properties. This process will take the input from the Unity Engine as a result of player actions and will facilitate and sanitize what is provided to the Level State Handler

Services Required:

1. **Service name:** GetPlayerInput to Inputs Handler
   **Service provided from:** Unity Engine

#### I.1.4. [IDE Reader]

### *a) Description*

The IDE Reader, similar to the Inputs Handler manages packaging the user's input, specifically the input in the Python text editor UI element. The UI Handler will provide the IDE Reader the raw text from the player, which the IDE Reader will package into a compiled object in the internal application system. Afterwards, the IDE Reader will provide this object to the Iron Python Handler. This adds a layer of abstraction which allows for sanitization of the user's text so as to avoid unintentional interactions between the code written by the player and the code written by the developers.

### *b) Concepts and Algorithms Generated*

The IDE Reader is responsible for the sanitizing and packaging of the player code, written in the UI element for the Python IDE, into a compiled object. Additionally the IDE Reader is responsible for providing this object to the Iron Python Handler for compilation. The IDE Reader will consist of one main class that is tied to the script of the UI element for the Python IDE. While most of this project's architecture parallels that of the Unity Engine, the IDE Reader is only used in an instance of the Python IDE UI element within a level. Something to note here, is that this component is only used in the script of one UI element, however variations of the UI element will be created for every level. So, in the greater scheme of things there will be multiple classes that compose the IDE Reader but they are multiple instantiations of this component.

### *c) Interface Description*

Services Provided:

1. **Service name:** CompilePlayerCode
   **Service provided to:** Iron Python Handler
   **Description:** The CompilePlayerCode service will take in the raw text input of the player, compile it into a data object, which is then provided to the Iron Python Handler. This will occur once the player has selected to simulate.

Services Required:

1. **Service name:** BundlePlayerCode to IDE Reader
   **Service provided from:** UI Handler

#### I.1.5. [Iron Python Handler]

##### *a) Description*

The Iron Python Handler is responsible for the interaction between the python code that the player has written and simulation of that code. This component will take input from any instantiation of the IDE Reader, as well as any given lesson file, which will consist of an external python file for the specific lesson. It will then provide the Level State Handler the results of the Python simulation of the player's code.

##### *b) Concepts and Algorithms Generated*

Diverging from the similarities of the previously explored components, the Iron Python Handler will consist of only one class. This one class will take inputs from two places, one, the IDE Reader class which will provide a compiled data object to run the Iron Python simulation on, and two, a lesson file, as mentioned before, the lesson file will consist of python environmental objects and methods specific for the lesson. The Iron Python Handler will also provide to the Level State Handler the results of the Iron Python Simulation.

##### *c) Interface Description*

Services Provided:

1. **Service name:** SimulatePlayerCode
   **Service provided to:** Level State Handler
   **Description:** The SimulatePlayerCode service will simulate the compiled data object from the IDE Reader, and provide the results to the Level State Handler. This occurs after the player selects to simulate.

Services Required:

1. **Service name:** LoadLesson to Iron Python Handler
   **Service provided from:** Lesson Files
2. **Service name:** CompilePlayerCode to Iron Python Handler
   **Service provided from:** IDE Reader

#### I.1.6. [Level Loader]

##### *a) Description*

The Level Loader subsystem is responsible for the loading and creation of save files. An important aspect of this project given the nature of video games is saving and loading. Players should be able to save their progression within the phases of a level and load their previously completed data for levels. The Level Loader is responsible for transposing the internal level data into a JSON file that can be accessed and read at a later time. It is also responsible for the reversal action, where given a save file, the Level Loader will read and interpret the save file and update the internal level data to reflect the progress within the save file.

##### *b) Concepts and Algorithms Generated*

Similar to the Iron Python Handler, the Level Loader subsystem will consist of a single class. This class will take an input of the internal class data from any instance of the Level State Handler and will transform the given data to a JSON file format as a Save file. Additionally, the Level Loader will take a JSON file input and transpose it into compiled data objects that provide the internal level objects with previously saved states.

### *c) Interface Description*

Provide a description of the subsystem interface. Explain the provided services in detail and give the names of the required services.

Services Provided:

1. **Service name:** SaveFile
   **Service provided to:** Level State Handler
   **Description:** The SaveFile service will take the input of the Level Loader's data objects progression then it will output the data transformed into a JSON file format and save it to a local storage location.
2. **Service name:** LoadFile
   **Service provided to:** Level State Handler
   **Description:** The LoadFile service will take the input of the JSON file from the Save Files subsystem, then it will transform the JSON data into a compiled data object and provide it to the Level State Handler. Afterwards, the Level State Handler will update its internal data objects based on the compiled JSON object.

Services Required:
**Service name:** GetLevelData to Level Loader
**Service provided from:** Level State Handler

#### I.1.7. [Level State Handler]

### *a) Description*

The Level State Handler is the main gameplay subsystem that is responsible for managing the services from the UI Handler, Iron Python Handler, Level Loader, and Inputs Handler to create a cohesive and fun level. It will handle displaying the level layout, running the player python code, the simulated results, and saving level progress at key milestones.

### *b) Concepts and Algorithms Generated*

The Level State Handler subsystem will consist of a single class attached to the Level UI Component. Similar to the IDE Reader, the Level State Handler will have multiple instances for each Level that the player can play, such as main, challenge, and boss levels. Additionally, it will provide Level data objects to the Level Loader in order to create or update a player's save file, which can be loaded in later.

### *c) Interface Description*

Services Provided:

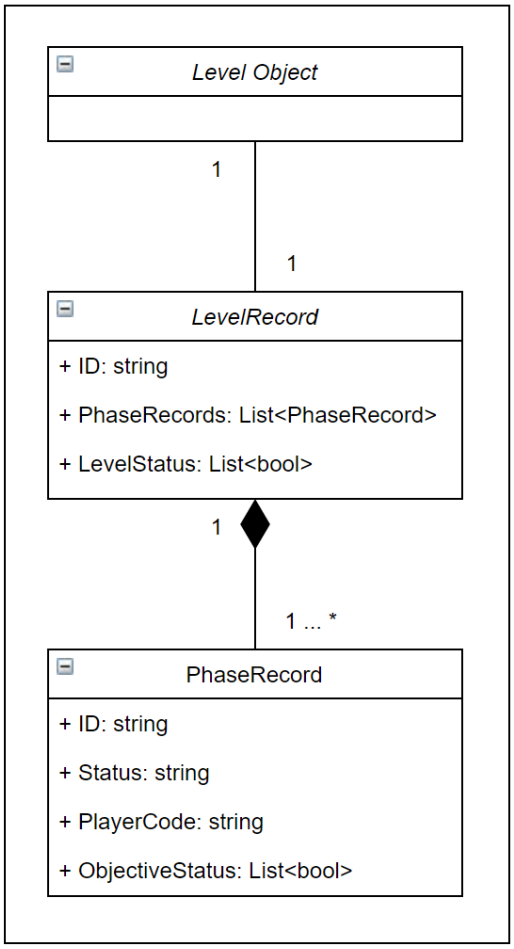
1. **Service name:** GetLevelData
   **Service provided to:** Level Loader
   **Description:** The GetLevelData service will provide the data objects of the level's progress to the Level Loader. This Level data will then be converted into a JSON file and saved to a local location.

Services Required:
**Service name:** UpdateLayout to Level State Handler

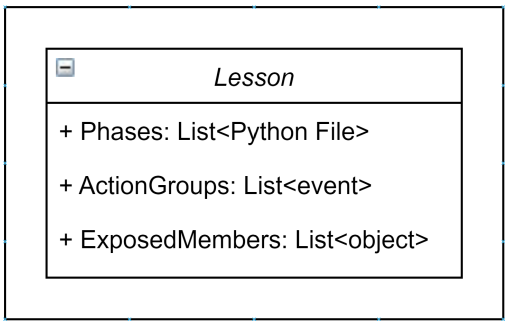
**Service provided from:** UI Handler
**Service name:** GetPlayerInput to Level Loader
**Service provided from:** Inputs Handler
**Service name:** LoadFile to Level Loader
**Service provided from:** Level Loader
**Service name:** SimulatePlayerCode to Level Loader
**Service provided from:** Iron Python Handler

## IV. Data design

For this application, there are two main data structures of concern, both which interact with external subsystems. These two structures pertain to the interaction with Lesson files and Save files. For more information about the internal subsystem interactions see the System Architecture diagram and description above.




(Save File)




(Lesson File)

For the Save File data structure, there will be a Level Object that functions as an abstraction for the LevelRecord object. This LevelRecord will consist of three main properties: ID, PhaseRecords and LevelStatus. The ID will be a string that contains information pertaining to

the level's creation as to distinguish levels that have been heavily modified from a base state. The LevelStatus object will contain a list of boolean values that correspond with the completion of each phase, this will be used to establish how much progress has been completed and which phases have been or need to be completed. The PhaseRecords will be a list of PhaseRecords. The PhaseRecord object will contain four properties: ID, Status, PlayerCode, ObjectiveStatus. The ID property is very similar to the ID property of the LevelRecord, and will serve to uniquely identify the phase. The Status will be a string that indicates the completion of the phase, this could be incomplete, partially complete, or complete. The PlayerCode will be a string representation of the player's code that was last compiled, this will allow players to continue from the last line of code they wrote at a later time. And lastly, the ObjectiveStatus object will be a list of boolean values, each that correspond to the completion status of the objectives for this phase. Given this structure, the LevelRecord is a composition of the PhaseRecords, where every LevelRecord contains at least one PhaseRecord.

Additionally, for the Lesson data structure, there are three main elements that it consists of. One, Phases - a List of Python files -, two, ActionGroups - a List of Events -, and three, ExposedMembers - a List of Objects. The Phases property contains python files that dictate what functions and variables the player has access to in the UI Python IDE element and can contain additional python files depending on the amount of phases in a lesson. The ActionGroups holds events that trigger modifications in the Unity UI subsystem. And the ExposedMembers list allows control over class members that are exposed to the user to interact with.

## V. User Interface Design

The Pluribus Doctrina Team has created a partial UI for the game, with a focus on the main layouts for each page that the player will interact with. For the User Interface Design, the team gave heavy consideration for the nature of the application, a video game, and the audience, middle school students. As development continues, the team has decided to follow a minimalistic format with bold colors to draw attention to important items. From the images in the Appendix, the player is first greeted with the start screen (Image 1). This page displays four items, one of which is the title of the game and the other three will be intractable buttons. These buttons consist of one, a start button, which will move to the player to the Level Select page with no previous progress, two, a continue button, which will prompt the player to choose a save file to load after choosing one, the player will be moved to the Level Select page with the previously saved progress added, and three, a quit button, which will terminate the executable, quitting the game. The second page in the player game loop is the Level Select page (Image 2), this page has two main elements: level selection, level description. The level selection, positioned on the left, will indicate which level is selected and the level description, positioned on the right, will give a brief description of the level, display level progress, provide a begin or continue button depending on the state of progress made in that level, as well as a restart button, that will reset any progress made. Once a level is selected, the player will then be moved to a Level page (Image 3). This layout will have three main elements: Python IDE, Simulation, AI response. The Python IDE will let the player write code within it and will contain a button to initiate code simulation, the Simulation will provide visual feedback to the player regarding the outcome of the python code they wrote, i.e. if it worked or if it did not, and the AI response will provide instruction, tips, and feedback to the player. The final page is the Manual page, which is accessible in either the Level Select page or the Level page. This page contains two main elements, similar to the Level Select page. While the left positioned element is the same as the Level Select page, the right positioned one, will instead contain information about the lesson

selected. This would include a written explanation of the python documentation and examples of what was taught in that lesson. The final image in the Appendix, Image 5, is the overlay screen for the game, the two elements that it contains are one, a menu button that will allow the player to choose to quit the game or to save their progress, and two, a manual button that will open the manual layout.

Some additional aspects to note, this is a partial UI mockup there are still parts of the UI that are undermined by both the team and the client, one such example is the background imagery. In the partial UI, there is a background of stars, however, this is temporary pending confirmation of the team and client. Additionally, this partial UI mockup doesn't denote any animations for layout change, actionable events, or simulations. However, these aspects will both be handled in the team's video demonstration farther in development.

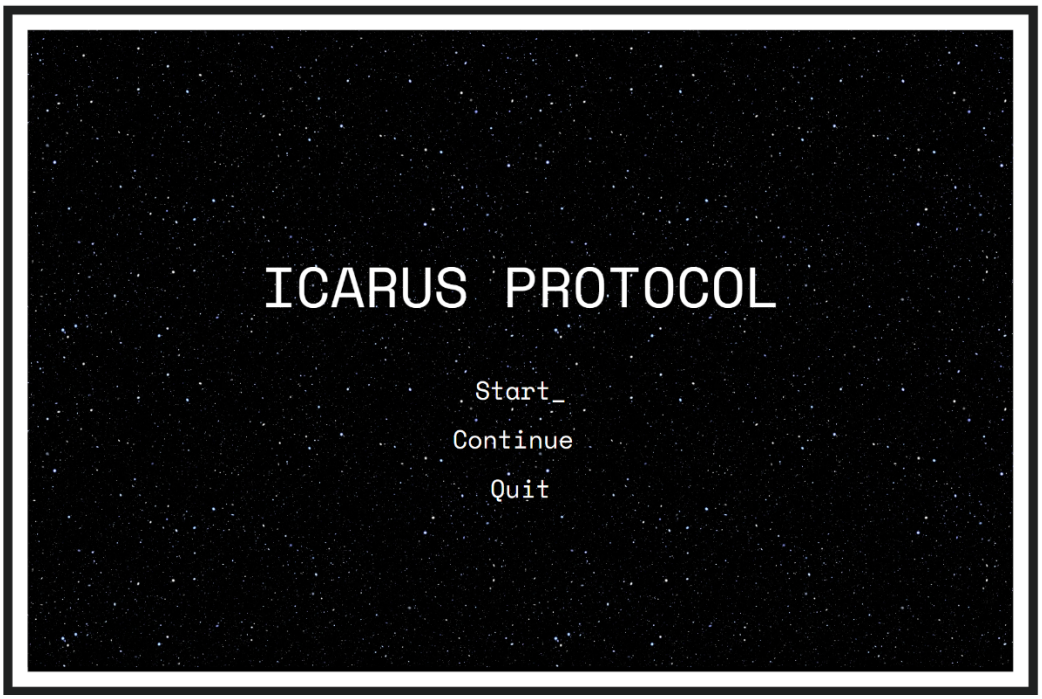
## VI. Glossary

HUD: Heads Up Display

IDE:

## VII. References

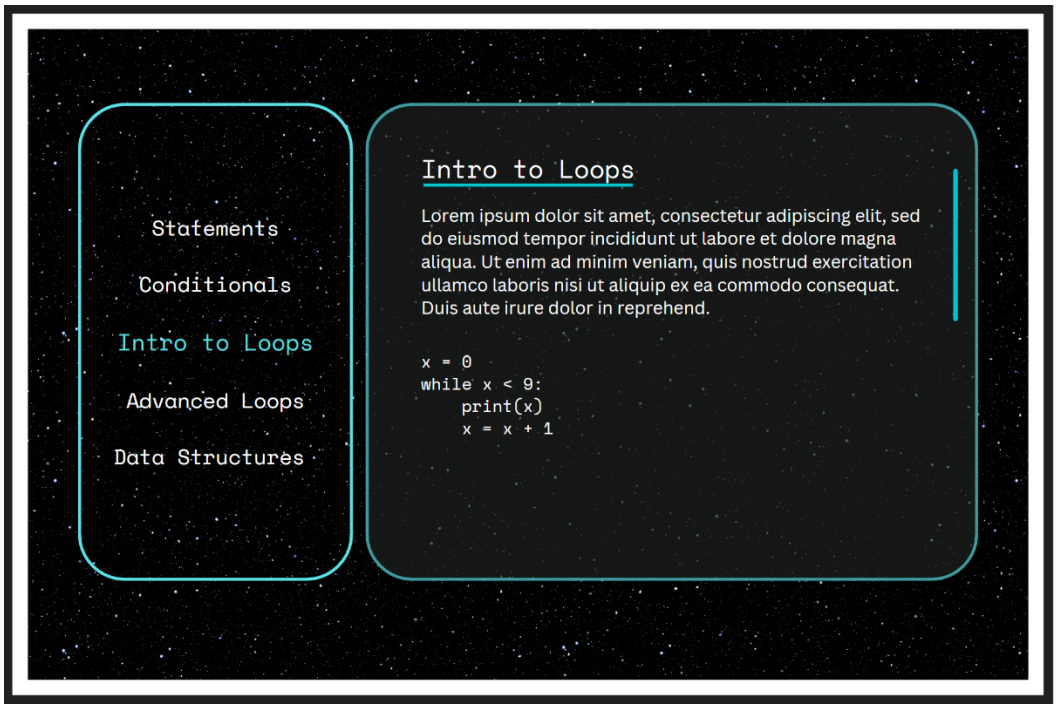
## VIII. Appendices




(Image 1)


(Image 2)


(Image 3)




(Image 4)

(Image 5)