mage licensing:

Plant images: used under CC-BY-SA 3.0+ license. Created by bluecarrot16, Daniel Eddeland (daneeklu), Joshua Taylor, Richard Kettering (Jetrel). Commissioned by castelonia. Sourced from: https://opengameart.org/content/lpc-crops

Player images: used under CC-BY 4.0 license. Created by wind astella. Sourced from: https://opengameart.org/content/stella

See https://creativecommons.org/licenses/by/4.0/ for details of this license.

Note: if you choose to open-source your solution (which you may only do after you have received your A3 grade) and you include the plant or player images, please ensure you credit the authors appropriately under the respective licenses.

Farm Game

Assignment 3

Semester 1, 2023

CSSE1001/CSSE7030

Due date: 26th May 2023 16:00 GMT+10

1 Introduction

In Assignment 3 you will implement a graphical user interface (GUI) for *Farm Game*. *Farm Game* is a simple farming video game in which the player can plant and harvest crops, tilling and untilling soil, and buying and selling seeds and produce. An example of a final completed game is displayed in Figure 1.

Figure 1: Example screenshot from a completed *Farm Game* implementation. Note that your display may look slightly different depending on your OS.

As opposed to earlier assignments where user interaction was handled via calls to `input`, user interaction in assignment 3 will occur via key-presses and mouse clicks.

Your solution will follow the Apple MVC design pattern covered in lectures. To assist you in your implementation, the model classes have been provided along with some extra support code and constants; see Section 4 for further details. You are required to implement a series of view classes as well as the controller class.

2 Setting Up

Aside from downloading and unzipping `a3.zip`, to begin this assignment you will also need to install the Pillow library via pip. Instructions on how to install Pillow can be found here[1].

3 Tips and hints

You should be testing **regularly** throughout the coding process. Test your GUI manually and regularly upload to Gradescope to ensure the components you have implemented pass the Gradescope tests. Note that Gradescope tests may fail on an implementation that visually appears correct if your implementation is wrong. You must implement your solution according to the implementation details from Section 6. Implementing the game using your own structure is likely to result in a grade of 0. Note also that minor differences in your program (e.g. a few pixels difference in widget size) may not cause the tests to fail. It is your responsibility to upload to Gradescope early and often, in order to ensure your solution passes the tests.

This document outlines the required classes and methods in your assignment. You are *highly encouraged* to create your own helper methods to reduce code duplication and to make your code more readable.

Except where specified, you are only required to do enough error handling such that regular game play does not cause your program to crash or error. If an attempt at a feature causes your program to crash or behave in a way that testing other functionality becomes difficult without your marker modifying your code, comment it out before submitting your assignment. If your solution contains code that prevents it from being run, you will receive a mark of 0.

You **must not add any imports**; doing so will result in a deduction of **up to 100% of your mark**.

You may use any code provided from the teaching staff of this course **in this semester only**. This includes any code from the support files or sample solutions for previous assignments **from this semester only**, as well as any lecture or tutorial code provided to you by course staff. However, it is your responsibility to ensure that this code is styled appropriately, and is an appropriate and correct approach to the problem you are addressing.

For additional help with tkinter, you can find documentation on effbot[2] and New Mexico Tech[3].

[1] `https://pillow.readthedocs.io/en/stable/installation.html`
[2] `https://web.archive.org/web/20171112065310/http://effbot.org/tkinterbook`
[3] `https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/index.html`

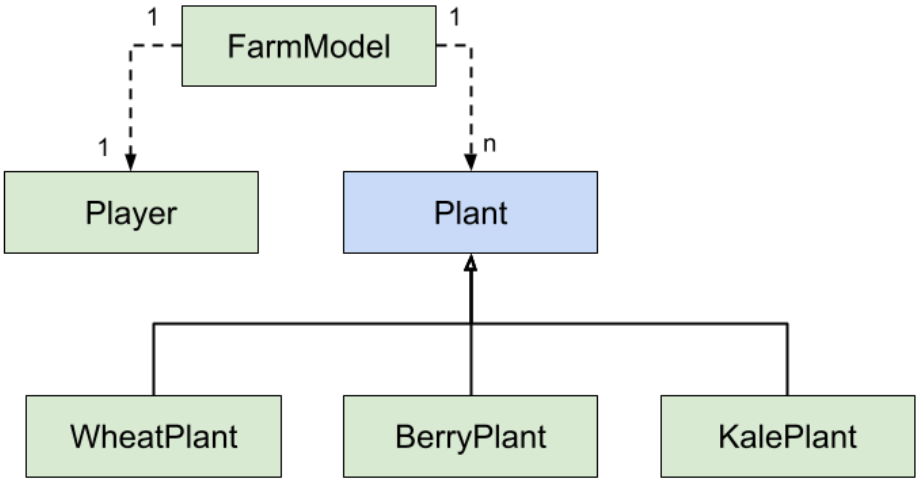

Figure 2: Class diagram for `model.py`.

4 Provided Code

This section provides a brief, high-level overview of the files provided for you in `a3.zip`. For further information, please see the documentation within each file.

4.1 model.py

The `model.py` file provides modelling classes for the plants and player, as well as the overall model for the game. Figure 2 presents a class diagram of the classes provided in this file. You will only need to instantiate `FarmModel` in your code. However, you will still be interacting with instances of the other classes via the `FarmModel` instance.

4.2 a3_support.py

The `a3_support.py` file contains support code to assist you in writing your solution. In particular, this file provides the following:

1. `get_plant_image_name(plant: 'Plant') -> str`: a function to help map between Plant objects and the names of their corresponding image files.

2. `get_image(image_name: str, size, cache=None) -> Image`: a function to create, resize, and optionally cache images based on the name of their image file. Returns the image object, which can be rendered onto a tkinter Canvas. ***Note:*** **you need to retain references to either all the images, or to the cache**. Tkinter will delete an image as soon as all references to it have been lost. ***Note also:*** **use of this function in creating images is mandatory.**

3. `read_map(map_file: str) -> list[str]`: Reads a map file into a list of strings, where each string represents one row of the map. The first item represents the top row, and the last item represents the bottom row.

4. `AbstractGrid`: `AbstractGrid` is an abstract view class which inherits from `tk.Canvas` and provides base functionality for multiple view classes. An AbstractGrid can be thought of as a grid with a set number of rows and columns, which supports creation of text and shapes at specific (row, column) positions. Note that the number of rows may differ from the number of columns, and may change after the construction of the AbstractGrid.

4.3 constants.py

The `constants.py` file contains a set of constants that will be useful in implementing your assignment. Where a detail is not specified in this document (e.g. size or colour of a widget, etc.), please refer to `constants.py`.

5 Recommended Approach

As opposed to earlier assignments, where you would work through the task sheet in order, developing GUI programs tends to require that you work on various interacting classes in parallel. Rather than working on each class in the order listed, you may find it beneficial to work on one *feature* at a time and test it thoroughly before moving on. Each feature will require updates / extensions to the controller, and potentially additions to one or more view classes. The recommended order of features (after reading through all of Section 6 of this document) are as follows:

1. `play_game`, `main`, and title: Create the window, ensure it displays when the program is run and set its title.
2. Title banner: Render the title banner at the top of the window.
3. `InfoPanel` and Next day button (including being able to increment the day on the info panel).
4. `FarmView`:
   - Display basic map from file.
   - Display and move player.
   - Till and untill soil.
5. `ItemView`:
   - Basic display (non-functional)
   - Binding command for selecting item. After this is implemented, you can implement planting, removing plants, growth of plants on a new day, and harvesting.
   - Binding commands for buying and selling items.

6 Implementation

You must implement three view classes; `InfoBar`, `FarmView`, and `ItemView`. Additionally, you must implement a controller class - `FarmGame` - which instantiates the `FarmModel` and all three view classes, and handles events.

This section describes the required structure of your implementation, however, it is not intended to provide an order in which you should approach the tasks. The controller class will likely need to be implemented in parallel with the view classes. See Section 5 for a recommended order in which you should approach this assignment.

6.1 InfoBar

`InfoBar` should inherit from `AbstractGrid` (see `a3_support.py`). It is a grid with 2 rows and 3 columns, which displays information to the user about the number of days elapsed in the game, as well as the player's energy and health. The `InfoBar` should span the entire width of the farm and inventory combined. An example of a completed `InfoBar` in the game is shown in Figure 3. The methods you must implement in this class are:

- `__init__(self, master: tk.Tk | tk.Frame) -> None`: Sets up this `InfoBar` to be an `AbstractGrid` with the appropriate number of rows and columns, and the appropriate width and height (see `constants.py`).
- `redraw(self, day: int, money: int, energy: int) -> None`: Clears the `InfoBar` and redraws it to display the provided day, money, and energy. E.g. in Figure 3, this method was called with day = 1, money = 0, and energy = 100.

Figure 3: InfoBar after redrawing with information from a new *FarmModel*.

6.2 FarmView

`FarmView` should inherit from `AbstractGrid` (see `a3_support.py`). The `FarmView` is a grid displaying the farm map, player, and plants. An example of a completed `FarmView` is shown in Figure 4. The methods you must implement in this class are:

- `__init__(self, master: tk.Tk | tk.Frame, dimensions: tuple[int, int], size: tuple[int, int], **kwargs) -> None`: Sets up the `FarmView` to be an `AbstractGrid` with the appropriate dimensions and size, and creates an instance attribute of an empty dictionary to be used as an image cache.
- `redraw(self, ground: list[str], plants: dict[tuple[int, int], 'Plant'], player_position: tuple[int, int], player_direction: str) -> None`:
  Clears the farm view, then creates (on the `FarmView` instance) the images for the ground, then the plants, then the player. That is, the player and plants should render in front of the ground, and the player should render in front of the plants. **You must use the `get_image` function from `a3_support.py` to create your images**.

6.3 ItemView

`ItemView` should inherit from `tk.Frame`. The `ItemView` is a frame displaying relevant information and buttons for a single item. There are 6 items available in the game (see the `ITEMS` constant in `constants.py`). The `ItemView` for an item should contain the following widgets, packed left to right:

- A label containing the name of the item and the amount of the item that the player has in their inventory, the selling price of the item, and the buying price of the item (if the item can be bought; see `BUY_PRICES` in `constants.py`).
- If this item can be bought, the frame should then contain a button for buying the item at the listed buy price.
- A button for selling the item at the listed sell price (all items can be sold).

There are three kinds of events that can occur on the `ItemView`: a left click on an items frame or label (indicating that the user would like to select the item), a button press on a buy button (indicating that the user would like to buy one of those items), and a button press on the sell

Figure 4: Example of a FarmView partway through a game.

button (indicating that the user would like to sell one of those items). The callbacks for these buttons must be created in the controller (see `FarmGame`) and passed to each `ItemView` via the constructor.

The methods that you must implement in this class are:

- `__init__(self, master: tk.Frame, item_name: str, amount: int, select_command: Optional[Callable[[str], None]] = None, sell_command: Optional[Callable[[str], None]] = None, buy_command: Optional[Callable[[str], None]] = None) -> None`: Sets up `ItemView` to operate as a `tk.Frame`, and creates all internal widgets. Sets the commands for the buy and sell buttons to the `buy_command` and `sell_command` each called with the appropriate `item_name` respectively. Binds the `select_command` to be called with the appropriate `item_name` when either the `ItemView` frame or label is left clicked. **Note:** The three callbacks are type-hinted as Optional to allow you to pass in None if you have not yet implemented these callbacks (i.e. when developing, just pass None to begin with, and hook up the functionality once you've completed the rest of the tasks; see Section 5).

- `update(self, amount: int, selected: bool = False) -> None`: Updates the text on the label, and the colour of this `ItemView` appropriately. See Figure 5 for more details on how these are presented. ***Note:*** you should configure the existing widgets in this method. You **must not** destroy and recreate the internal widgets.

Note: Handling callbacks is an advanced task. These callbacks will be created within the controller class, as this is the only place where you have access to the required modelling information. Start this task by trying to render the `ItemView`s correctly, without the callbacks. Then integrate these views into the game, before working on the select command. Once items can be selected from the inventory, work on planting, growing and removing plants. It is recommended that you leave the buying and selling functionality until the end of the assignment.

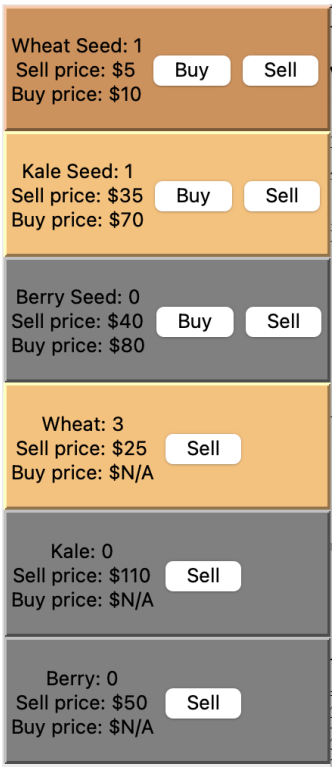

Figure 5: Example of all 6 `ItemView` instances in the overall game. Note: some operating systems will retain the default background colour behind buttons. While you will not be marked down for this, if you would like to set the colour behind the buttons, you can do so via the highlightbackground kwarg.

6.4 FarmGame

`FarmGame` is the controller class for the overall game. The controller is responsible for creating and maintaining instances of the model and view classes, event handling, and facilitating communication between the model and view classes. Figure 1 provides an example of how the *FarmGame* should look. Certain events should cause behaviour as per Table 1. The methods that you must implement in this class are:

- `__init__(self, master: tk.Tk, map_file: str) -> None`: Sets up the `FarmGame`. This includes the following steps:
  - Set the title of the window.
  - Create the title banner (you must use `get_image`).
  - Create the `FarmModel` instance.
  - Create the instances of your view classes, and ensure they display in the format shown in Figure 1.
  - Create a button to enable users to increment the day, which should have the text 'Next day' and be displayed below the other view classes. When this button is pressed, the model should advance to the next day, and then the view classes should be redrawn to reflect the changes in the model.
  - Bind the `handle_keypress` method to the `'<KeyPress>'` event.
  - Call the `redraw` method to ensure the view draws according to the current model state.
- `redraw(self) -> None`: Redraws the entire game based on the current model state.

- `handle_keypress(self, event: tk.Event) -> None`: An event handler to be called when a keypress event occurs. Should trigger the relevant behaviour as per Table 1, and cause the view to update to reflect the changes. If a key is pressed that does not correspond to an event, it should be ignored.
- `select_item(self, item_name: str) -> None`: The callback to be given to each ItemView for item selection. This method should set the selected item to be `item_name` and then redraw the view.
- `buy_item(self, item_name: str) -> None`: The callback to be given to each ItemView for buying items. This method should cause the player to attempt to buy the item with the given `item_name`, at the price specified in `BUY_PRICES`, and then redraw the view.
- `sell_item(self, item_name: str) -> None`: The callback to be given to each ItemView for selling items. This method should cause the player to attempt to sell the item with the given `item_name`, at the price specified in `SELL_PRICES`, and then redraw the view.

Event	**Behaviour**
Key Press: 'w'	Player attempts to move up one square.
Key Press: 'a'	Player attempts to move left one square.
Key Press: 's'	Player attempts to move down one square.
Key Press: 'd'	Player attempts to move right one square.
Key Press: 'p'	If an item is selected and that item is a seed, attempt to plant that seed at the player's current position. If that position does not contain soil, a plant already exists in that spot, or a seed is not currently selected, do nothing.
Key Press: 'h'	Attempt to harvest the plant from the player's current position. If no plant exists at the player's current location, or the plant is not ready for harvest, do nothing. If the harvest is successful, add the harvested item/s to the player's inventory, and if the plant should be removed on harvest, remove the plant from the farm.
Key Press: 'r'	Attempt to remove the plant from the player's current position. Note that this does not harvest the plant, and does not require the plant to be ready for harvest. If no plant exists at the player's currently location, do nothing.
Key Press: 't'	Attempt to till the soil from the player's current position. If that position does not contain untilled soil, do nothing.
Key Press: 'u'	Attempt to untill the soil from the player's current position. If that position does not contain tilled soil, do nothing. If the position contains a plant, do not untill the soil.
Left click on an item in the inventory	If the player has a non-zero amount of that item, set the selected item to the name of that item.
Button press (buy button on item in inventory)	Attempt to buy the item.
Button press (sell button on item in inventory)	If player has a non-zero quantity of that item, attempt to sell one of the item.

Table 1: Events and their corresponding behaviours.

6.5 `play_game(root: tk.Tk, map_file: str) -> None` function

The `play_game` function should be fairly short. You should:

1. Construct the controller instance using given `map_file` and the root `tk.Tk` parameter.
2. Ensure the root window stays opening listening for events (using `mainloop`).

6.6 main function

The `main` function should:

1. Construct the root `tk.Tk` instance.
2. Call the `play_game` function passing in the newly created root `tk.Tk` instance, and the path to any map file you like (e.g. `'maps/map1.txt'`).

7 Postgraduate Task: File Menu

Students of CSSE7030 are required to implement a file menu into the game, as described in this section. The file menu should contain a 'Quit' button, which gracefully ends the program (similar to clicking the X in the top left corner). It should also contain a 'Map selection' option, that prompts the user for a map file via a file dialogue, and then starts a new game with that map. At the beginning of a new game, the player is set to have no money, full energy, and be at day 1. The player should be at position (0, 0). Figure 6 shows how this file menu would look on Mac OS. On different operating systems, it may display in different locations (e.g. on Windows, the menu would display within the window). Figure 7 shows how the file dialogue might look on Mac OS, and Figure 8 shows how the game would look directly after loading in `'maps/map2.txt'` from the supplied maps via this feature.

Note: you may assume we will only test your program with valid map files. However, you may not assume that the map file will be the same dimensions as the initially loaded map. You may add a public `clear_cache` method to your `FarmView` class to help with this task.

Figure 6: File menu on Mac (may look different on different OS).

8 Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. apply basic object-oriented concepts such as classes, instances and methods,

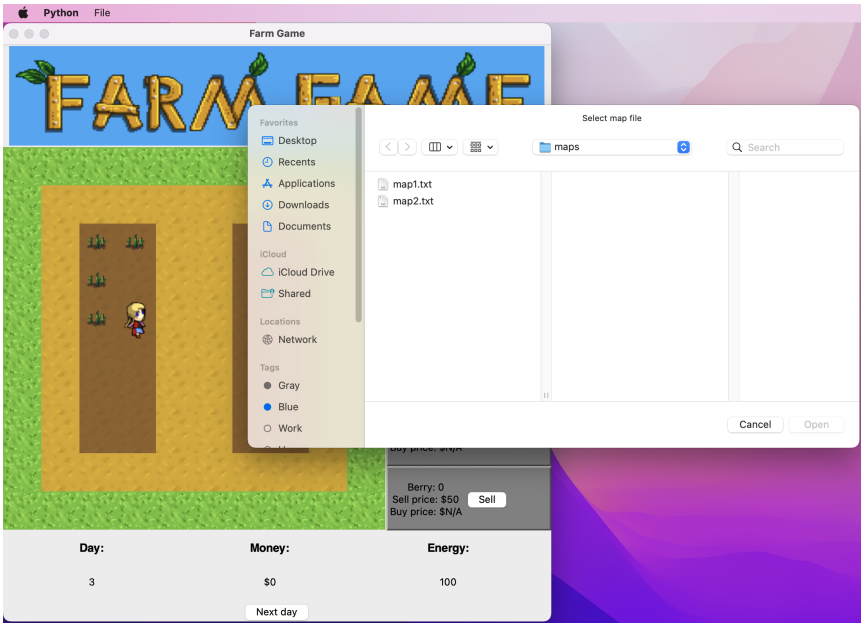

Figure 7: Prompt for new map file after navigating to maps directory (may look different on different OS).

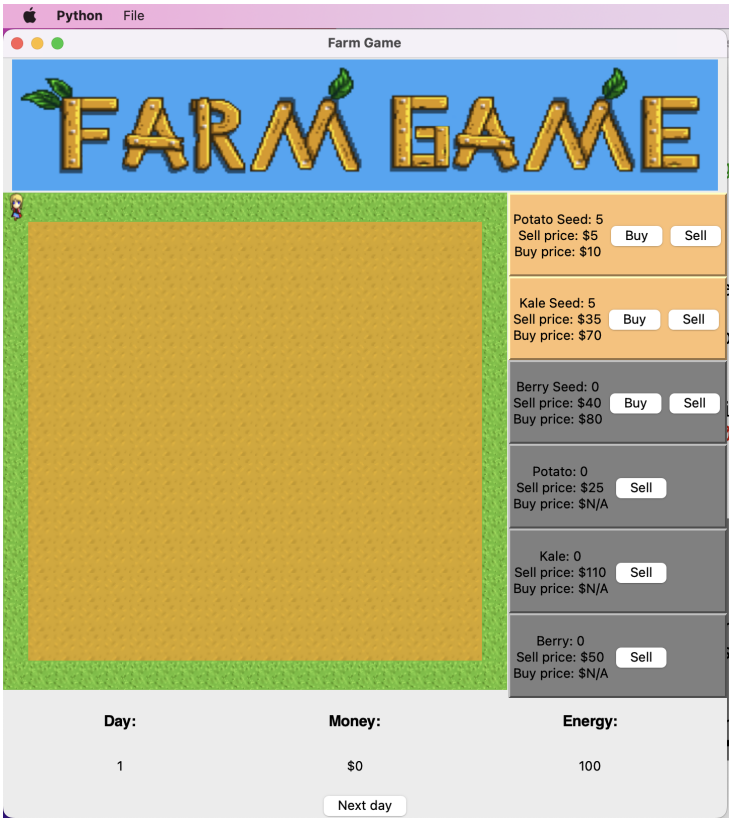

Figure 8: The game after loading in `'maps/map2.txt'` via the 'Map selection' feature on the file menu.

3. read and analyse code written by others,
4. analyse a problem and design an algorithmic solution to the problem,
5. read and analyse a design and be able to translate the design into a working program, and
6. apply techniques for testing and debugging, and
7. design and implement simple GUIs.

8.1 Functionality Marking

Your program's functionality will be marked out of a total of 70 marks. As in assignment 1 and 2, your assignment will be put through a series of tests and your functionality mark will be proportional to the number of weighted tests you pass. You will be given a *subset* of the functionality tests before the due date for the assignment.

Your assignment will be tested on the functionality of gameplay features. The automated tests will play the game and attempt to identify components of the game, how these components function during gameplay will then be tested. Well before submission, run the functionality tests to ensure components of your application can be identified. If the autograder is unable to identify components, you will not receive marks, **even if your assignment is functional**. The tests provided prior to submission will help you ensure that all components can be identified by the autograder.

You also need to perform your *own* testing of your program to make sure that it meets *all* specifications given in the assignment. Only relying on the provided tests is likely to result in your program failing in some cases and you losing some functionality marks.

Your program must run via Gradescope, which runs Python 3.11. Partial solutions will be marked, but if there are errors in your code that cause the interpreter to fail to execute your program, you will get zero for functionality marks. If there is a part of your code that causes the interpreter to fail, comment out the code so that the remainder can run. Your program must run using the Python 3.11 interpreter. If it runs in another environment (e.g. Python 3.9, or PyCharm) but not in the Python 3.11 interpreter, you will get zero for the functionality mark.

8.2 Style Marking

The style of your assignment will be assessed by a tutor. The style mark will be out of 30.

The key consideration in marking your code style is whether the code is easy to understand. There are several aspects of code style that contribute to how easy it is to understand code. In this assignment, your code style will be assessed against the following criteria.

- Readability
  - Program Structure: Layout of code makes it easier to read and follow its logic. This includes using whitespace to highlight blocks of logic.
  - Identifier Names: Variable, constant, function, class and method names clearly describe what they represent in the program's logic. Do **not** use *Hungarian Notation* for identifiers.
- Documentation

  - Inline Comments: All significant blocks of code should have a comment to explain how the logic works. For a small method or function, the logic should usually be clear from the code and docstring. For long or complex methods or functions, each logical block should have an in-line comment describing its logic.
  - Informative Docstrings: Every class, method and function should have a docstring that summarises its purpose. This includes describing parameters and return values so that others can understand how to use the method or function correctly.

- Code Design
  - Single Instance of Logic: Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a method or function.
  - Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. loops and conditional statements).

- Object-Oriented Program Structure
  - Model View Controller: The GUI's view and control logic is clearly separated from the model. Model information stored in the controller and passed to the view when required.
  - Abstraction: Public interfaces of classes are simple and reusable. Enabling modular and reusable components which abstract GUI details..
  - Encapsulation: Classes are designed as independent modules with state and behaviour. Methods only directly access the state of the object on which they were invoked. Methods never update the state of another object.
  - Inheritance: Subclasses extend the behaviour of their superclass without re-implementing behaviour, or breaking the superclass behaviour or design. Abstract classes have been used to effectively group shared behaviour amongst subclasses.

9 Assignment Submission

Your assignment must be submitted as `a3.py` via the assignment three submission link on Gradescope. You should not submit any other files (e.g. maps, images, etc.). You do not need to resubmit any supplied files.

Late submission of the assignment without an approved extension will incur a late penalty as per the course ECP. In the event of exceptional circumstances, you may submit a request for an extension.

All requests for extension must be submitted on the UQ Application for Extension of Progressive Assessment form: `https://my.uq.edu.au/node/218/2` **at least 48 hours prior** to the submission deadline.