2048

Assignment 3
Semester 2, 2022
CSSE1001/CSSE7030

Due date: October 26 at 20:00.

Version 0.2: Changes from Version 0.1 marked in blue.

Introduction

2048 is a single-player game where the player tries to construct a 2048 tile on a 4x4 grid (see Figure 1). At any time each grid position is either empty or labelled with a power of two. Two random tiles are placed in random empty positions to initialize the game. Random tiles are usually 2's but 4's are occasionally possible.

Each turn, the player moves all tiles up, down, left, or right, by pressing `w`, `s`, `a`, or `d` respectively. Tiles move in the chosen direction until they hit a wall or another tile. If a tile hits another tile *with the same value* the two tiles merge (in the direction of the move) into a single tile with twice the value. After this move, and before the next, a random tile is placed on a random empty board position.

If the player constructs a 2048 tile they win and the game ends. If no more moves can be made that would change the game state, the player loses and the game ends.

To get a better idea of how the game works, try playing it here: `https://play2048.co/`.

Implement a GUI-based *2048* application using `tkinter`. You can find the `tkinter` documentation on effbot[1] and New Mexico Tech[2]. Your implementation should follow the Apple MVC structure[3].

Rules, Tips, and Hints

Functionality marking for this assignment is automated and works by analysing your GUI. Consequently, solutions that do not display, or display incorrectly, may not receive marks. You may obtain partial marks for partial displays; e.g. you can receive partial marks for a correct title label, even if your tiles do not display, and you can receive partial marks for correct initial tiles, even if you do not get player input to modify the game state. It is recommended to work on the model, view, and controller classes simultaneously.

This assignment has two tasks for undergraduate students and an additional task for postgraduate students. *You can pass the assignment and achieve a playable game by implementing task one*

[1] `https://web.archive.org/web/20171112065310/http://effbot.org/tkinterbook`
[2] `https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/index.html`
[3] `https://developer.apple.com/library/archive/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html`

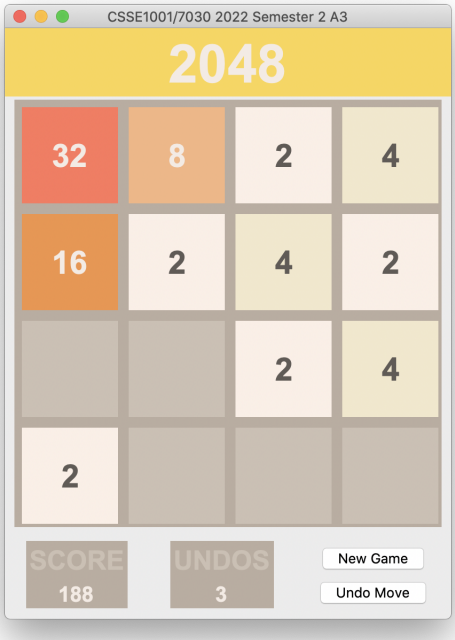

Figure 1: Example fully functional CSSE1001 2048 game.

only. Successful completion of task two is challenging and only necessary to obtain a top grade. The number of marks associated with each task is not an indication of difficulty. Task one may take less effort than task two, yet is worth significantly more marks.

Minor differences in the look (e.g. colours, fonts, etc.) of the GUI are acceptable *provided* they do not cause the Gradescope tests to fail. 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 your solution contains code that prevents it from being run, you will receive a mark of 0.

You *must only* make use of libraries listed in Appendix A. You *must not* import anything that is not on this list; doing so will result in a *deduction of up to 100% of your mark*.

You *may* use 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.

Task 1: Basic Gameplay

By the end of Task 1 you should have implemented a functional GUI-based version of *2048*, which looks like Figure 2 and Figure 3. A heading label with yellow background and white text '2048' should appear at the top of the window at all times. Below the label, the 4x4 grid of tiles should be displayed on a canvas. These tiles are represented by coloured rectangles annotated with their values. They *must* use the tile and text colours for each value found in `a3_support.py`. Certain events should cause behaviours as per Table 1.

When the player wins or loses the game inform them of the outcome via an `askyesno` messagebox. The messagebox must display the text of `WIN_MESSAGE` or `LOSS_MESSAGE` that are defined in `a3_support.py`. Both of these messages ask the player if they would like to play again. If 'yes', the messagebox should close and a new game should begin. If 'no', the entire program should end

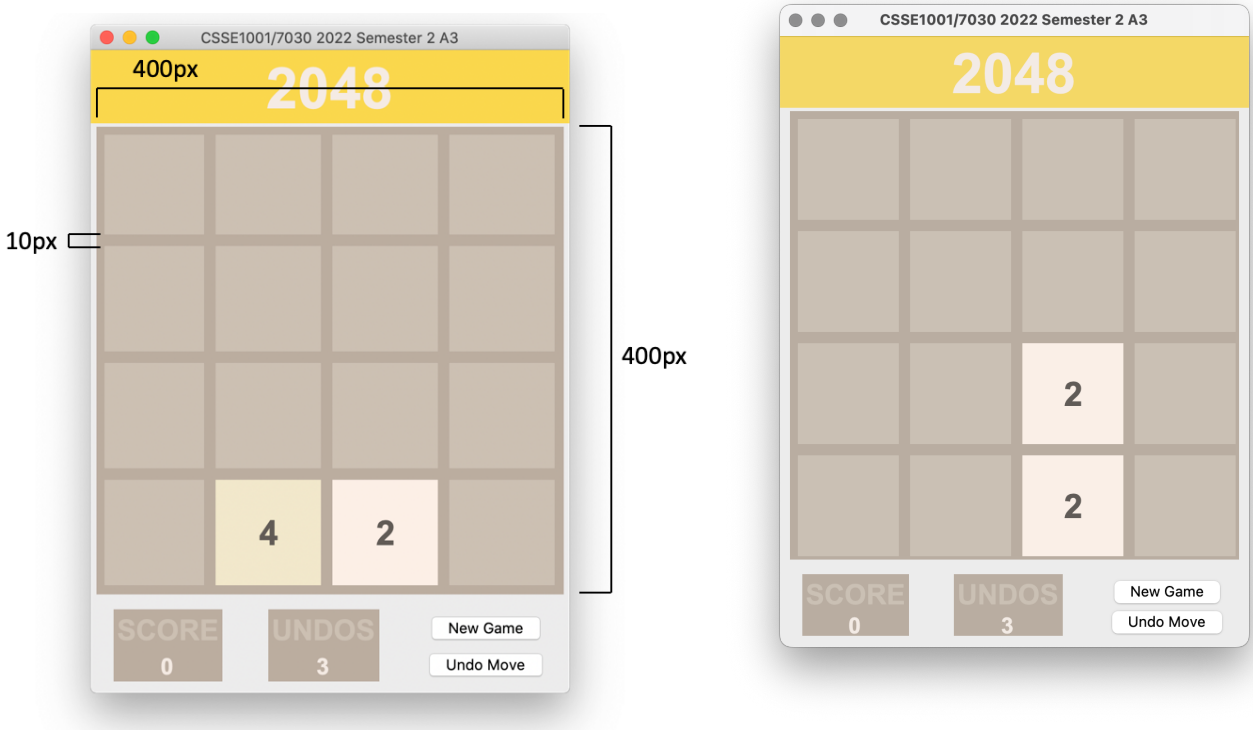

Figure 2: Ideal padding and spacing is indicated on the left. Due to some `tkinter` nuances some padding may be missing as on the right — this is acceptable (you will not lose marks).

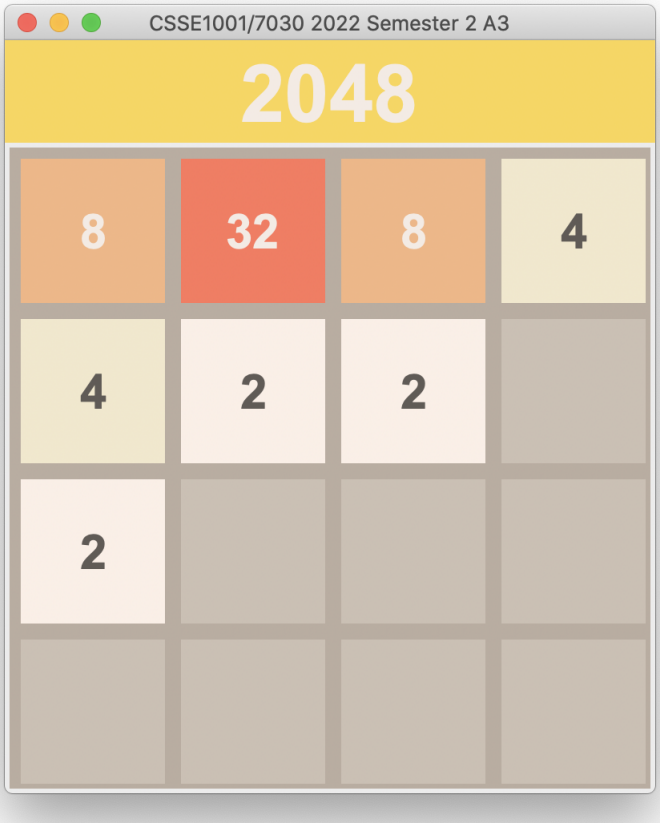

Figure 3: Example completed Part 1 partway through gameplay.

Event	*Behaviour*
Key Press: `w`	All squares shift up as far as they can and merge if necessary.
Key Press: `a`	All squares shift left as far as they can and merge if necessary.
Key Press: `s`	All squares shift down as far as they can and merge if necessary.
Key Press: `d`	All squares shift right as far as they can and merge if necessary.

Table 1: Events and their corresponding behaviours. A move is only applicable if it would cause a change to the game state. All other keypresses should be ignored (no error should occur on other keypresses).

gracefully (e.g. the window should close and the `>>>` should reappear in the REPL).

To complete this task implement

1. a model class to maintain and modify the game state,
2. a single view class for the 4x4 grid, and
3. a controller class to maintain instances of and communication between, the model and view.

The following sub-sections outline the *required* structure for your code. You will benefit from writing these classes in parallel, but you should still test individual methods as you write them. You are permitted to add additional modelling classes and any additional helper methods if they improve your solution.

1.1 Model class

You *must* use the `generate_tile` function from `a3_support.py` to generate random tiles, when required. If you make your own calls to `random` your code may not generate the expected tiles and fail Gradescope tests. While you are not *required* to use the other support functions, they will likely be very useful in achieving the various `move` methods.

You *must* define a class called `Model` that implements the following messages.

- `__init__(self) -> None:`
  Constructs a new *2048* model instance. This includes setting up a new game (see `new_game` method below).
- `new_game(self) -> None:`
  Sets, or resets, the game state to an initial game state. Any information is set to its initial state, the tiles are all set to empty, and then two new starter tiles are randomly generated (see the `add_tile` method below).
- `get_tiles(self) -> list[list[Optional[int]]]:`
  Return the current tiles matrix. Each internal list represents a row of the grid, ordered from top to bottom. Each item in each row list is the integer value on the tile occupying that space, or None if no tile is occupying that space.
- `add_tile(self) -> None:`
  Randomly generate a new tile at an empty location (you *must* use `generate_tile` for this) and add it to the current tiles matrix.
- `move_left(self) -> None:`
  Moves all tiles to their left extreme, merging where necessary. This involves stacking all tiles to the left, merging to the left, and then restacking to the left to fill in any gaps created. If

you are keeping track of a score (see Task 2), this method should also add any points gained from the move to the total score.

- `move_right(self) -> None`: Moves all tiles to their right extreme, merging where necessary. This can be achieved by reversing the tiles matrix, moving left, and then reversing the matrix again. If you are keeping track of a score (see Task 2), this method should also result in gained points being added to the total score.
- `move_up(self) -> None`:
  Moves all tiles to their top extreme, merging where necessary. This can be achieved by transposing the tiles matrix, moving left, and then transposing the matrix again. If you are keeping track of a score (see Task 2), this method should also result in gained points being added to the total score.
- `move_down(self) -> None`:
  Moves all tiles to their bottom extreme, merging where necessary. This can be achieved by transposing the tiles matrix, moving right, and then transposing the matrix again. If you are keeping track of a score (see Task 2), this method should also result in gained points being added to the total score.
- `attempt_move(self, move: str) -> bool`:
  Makes the appropriate move according to the move string provided. Returns `True` if the move resulted in a change to the game state, else `False`. The move provided must be one of `wasd` (this is a pre-condition, not something that must be handled within this method).
- `has_won(self) -> bool`:
  Returns `True` if the game has been won, else `False`. The game has been won if a 2048 tile exists on the grid.
- `has_lost(self) -> bool`:
  Returns `True` if the game has been lost, else `False`. The game has been lost if there are no remaining empty places in the grid, but no move would result in a change to the game state.

1.2 GameGrid

The `GameGrid` is a view class which inherits from `tk.Canvas` and represents the 4x4 grid. On the `GameGrid` canvas, tiles should be drawn using a combination of the `create_rectangle` and `create_text` method on `self`. You should *not* create a new `tk.Canvas` instance as an attribute within this class; doing so will cause the Gradescope tests to report that your `GameGrid` canvas contains no items. You should implement the following methods in this class:

- `__init__(self, master: tk.Tk, **kwargs) -> None`:
  Sets up a new `GameGrid` in the `master` window. `**kwargs` is used to allow `GameGrid` to support any named arguments supported by `tk.Canvas`. The canvas should be 400 pixels wide and 400 pixels tall.
- `_get_bbox(self, position: tuple[int, int]) -> tuple[int, int, int, int]`:
  Return the bounding box for the (row, column) position, in the form

  `(x_min, y_min, x_max, y_max)`.

  Here, `(x_min, y_min)` is the top left corner of the position with 10 pixels of padding added, and `(x_max, y_max)` is the bottom right corner of the cell with 10 pixels of padding subtracted.

- `_get_midpoint(self, position: tuple[int, int]) -> tuple[int, int]`:
  Return the graphics coordinates for the center of the cell at the given `(row, col)` position.
- `clear(self) -> None`:
  Clears all items.
- `redraw(self, tiles: list[list[Optional[int]]]) -> None`:
  Clears and redraws the entire grid based on the given tiles.

1.3 Game class

You *must* implement a class for the controller, called `Game`. This class should be instantiated in your main function to cause the game to be created and run. The controller class is responsible for maintaining the model and view classes, binding some event handlers, and facilitating communication between model and view classes. You must implement the following methods:

- `__init__(self, master: tk.Tk) -> None`:
  Constructs a new 2048 game. This method should create a `Model` instance, set the window title, create the title label and create instances of any view classes packed into `master`. It should also bind key press events to an appropriate handler, and cause the initial GUI to be drawn.
- `draw(self) -> None`:
  Redraws any view classes based on the current model state.
- `attempt_move(self, event: tk.Event) -> None`:
  Attempt a move if the `event` represents a key press on character 'a', 'w', 's', or 'd'. Once a move has been made, this method should redraw the view, display the appropriate messagebox if the game has been won, or create a new tile after 150ms if the game has not been won.
- `new_tile(self) -> None`: Adds a new tile to the model and redraws. If the game has been lost with the addition of the new tile, then the player should be prompted with the appropriate messagebox displaying the `LOSS_MESSAGE`.

1.4 play_game function

Fill in the `play_game` function with code to instantiate your app. This function just requires you to create an instance of your `Game` class that uses the given `root` window as the `master` argument.

Task 2: StatusBar

Once your Task 1 functionality works (read: passes the Gradescope tests) you may attempt Task 2, which requires the addition of a `StatusBar`. As part of this status bar feature, you will need to add a scoring system and the capacity to undo previous moves. You will need to extend the model, and controller, and add a `StatusBar` class. The rest of this section outlines the required additions.

2.5 Model modifications

You must add the following methods to your `Model` class:

- `get_score(self) -> int`: Returns the current score for the game. Each time a new tile is created by a merge, its new value should be added to the score. The total score to be added after a merge is calculated for you by the `combine_left` function in `a3_support.py`.
- `get_undos_remaining(self) -> int`: Get the number of undos the player has remaining. This should start at 3 at the beginning of a new game, and reduce each time an undo is used.
- `use_undo(self) -> None`: Attempts to undo the previous move, returning the current tiles to the previous tiles state before the last move that made changes to the tiles matrix. If the player does not have any undos remaining, or they are back at the initial state, this method should do nothing.

Note that you will likely need to update existing methods in order to get all of these features to work correctly.

2.6 StatusBar class

You must add a class `StatusBar` that inherits from `tk.Frame` and represents information about score and remaining undos, as well as a button to start a new game and a button to undo the previous move. You can see the layout of the `StatusBar` frame below the 4x4 grid in Figure 1. While you will likely need to construct `tk.Frame` instances in this class to achieve the layout, these frames (and the things inside them) *must* be in the `StatusBar` it*self*, not directly in `master`.

You must include the following methods in this class:

- `__init__(self, master: tk.Tk, **kwargs)`: Sets up `self` to be an instance of `tk.Frame` and sets up inner frames, labels and buttons in this status bar.
- `redraw_infos(self, score: int, undos: int) -> None`: Updates the score and undos labels to reflect the information given.
- `set_callbacks(self, new_game_command: callable, undo_command: callable) -> None`: Sets the commands for the new game and undo buttons to the given commands. The arguments here are *references* to functions to be called when the buttons are pressed.

2.7 Game modifications

You must modify your `Game` class to add the `StatusBar` in to display underneath the 4x4 grid. In order to do this you will likely need to modify the `__init__` and `draw` methods, and add the following methods:

- `undo_previous_move(self) -> None`: A handler for when the 'Undo' button is pressed in the status bar. This method should attempt to undo the last action, and then redraw the view classes with the updated model information.
- `start_new_game(self) -> None`: A handler for when the 'New Game' button is pressed in the status bar. This method should cause the model to set its state to that of a new game, and redraw the view classes to reflect these changes. *Note:* The new game should not replicate the initial state of the previous game. The new game state should be the result of calling the `new_game` method on the Model instance.

These handlers for the button presses must be defined in the controller, because they require interaction with both the model and the view. In order to connect them to the buttons in your `StatusBar` class, you will need to call the `StatusBar.set_callbacks` method you defined earlier.

Task 3: Postgraduate task

If you are enrolled in CSSE7030, once you have completed section 1 and section 2, you should attempt section 3, which requires the addition of a file menu. The file menu must have the option to quit, create a new game, save a game, and load existing games.

3.8 File Menu

Add a file menu with the options described in Table 2. Note that on Windows this will appear in the window, whereas on Mac this will appear at the top of your screen. The exact layout is not overly important (e.g. your menu may have a dashed line at the top, or other minor differences in display), as long as the options are present with the correct text, and correct functionality.

For saving and loading files, you must design an appropriate file format to store information about games. You may use any format you like, as long as you do not import anything that has not been explicitly permitted in Appendix A, and your save and load functionality work together. Reasonable actions by the player (e.g. trying to load a non-game file) should be handled appropriately (e.g. with a pop-up to inform the player that something went wrong).

Option	*Behaviour*
Save game	Prompt the player for the location to save their file (using the `asksaveasfile` function from the `filedialog` submodule in `tkinter`) and save all necessary information to replicate the current state of the game.
Load game	Prompt the player for the location of the file (using the `askopenfile` function from the `filedialog` submodule in `tkinter`) to load a game from and load the game described in that file.
New game	Create a new game.
Quit	Prompt the player via messagebox to ask whether they are sure they would like to quit. If no, do nothing. If yes, quit the game (window should close and program should terminate).

Table 2: File menu options.

4 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,
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.

4.1 Marking Breakdown

This assignment is worth 17% of your final grade. The marks that make up this 17% differ depending on whether you are enrolled in CSSE1001 (undergraduate students) or CSSE7030 (postgraduate students). Undergraduate students will be marked out of 100 possible marks, while postgraduate students will be marked out of 125 possible marks. Your total mark will be made up of functionality marks and style marks. Functionality marks are worth 75 marks for undergraduate and 100 marks for postgraduate students. The overall percentage earned towards your final grade will be calculated according to:

$$\text{percentage earned} = 17 \times \frac{\text{style mark} + \text{functionality mark}}{\begin{cases} 125, & \text{if postgraduate} \\ 100, & \text{otherwise} \end{cases}}$$

4.2 Functionality Marking

Table 3 outlines the breakdown of functionality marks for the assignment. 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. It is *your responsibility* to ensure that you submit your assignment early enough to identify issues in your assignment that are caught by the tests that may not be caught by your local testing process. If the autograder is unable to identify components, you will not receive marks, *even if your assignment appears 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 in Gradescope, which uses Python 3.10. 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 3.10 interpreter. If it runs in another environment (e.g. Python 3.9, or PyCharm) but not in the Python 3.10 interpreter, you will get zero for the functionality mark.

4.3 Style Marking

The style of your assignment will be assessed by a tutor. The style mark will be out of 25. 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.

Feature	*Undergradute*	*Postgraduate*
Task 1	*45*	*45*
Window title and banner	5	5
Initial game displays	15	15
Simple move behaviour	8	8
Tiles are correctly generated	5	5
Merging occurs correctly	7	7
Game end events	5	5
Task 2	*30*	*30*
StatusBar displays and contains correctly formatted widgets	10	10
Score updates correctly	5	5
Undo Move (button works and label updates correctly)	10	10
New Game (button works and views update correctly)	5	5
Task 3	*0*	*25*
File menu exists with correct options	0	5
Quit feature on file menu functions correctly	0	5
New game feature on file menu functions correctly	0	5
Save game and load game function correctly together	0	10
Total	75	100

Table 3: Functionality marks breakdown.

- 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.

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. support code, etc.). Late submission of the assignment will *not* be accepted. 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 submissiondeadline.

Appendix: Permitted Libraries

You are permitted to import `tkinter` and any of its submodules in order to form a working solution. You may also import `typing` if you wish, though you do not need to in order to create a working solution. You may import `a3_support.py`, which is already imported by default in the provided `a3.py` stub file.

Use of any other libraries (including in-built libraries) *is not permitted* and will be penalized by a deduction of up to 100% of the Assignment 3 grade.