Fancy Sokoban

Assignment 2
Semester 1, 2023
CSSE1001/CSSE7030

Due date: 22 Sept 2023 15:00 GMT+10

1 Introduction

Sokoban is a simple game in which a player tries to push crates onto goal locations in a grid. In assignment 2, you will create a slightly modified text-based version of Sokoban, in which the player may also need to collect potions in order to be able to push all crates to goal locations within an allocated number of moves. Your implementation will follow the Apple Model-View Controller (MVC) structure mentioned in lectures.

You are required to implement a collection of classes and methods as specified in Section 5 of this document. Your program's output must match the expected output *exactly*; minor differences in output (such as whitespace or casing) *will* cause tests to fail, resulting in *zero marks* for those tests. Any changes to this document will be listed in a changelog on Blackboard, and summarized on the Edstem megathread.

2 Getting Started

Download `a2.zip` from Blackboard — this archive contains the necessary files to start this assignment. Once extracted, the `a2.zip` archive will provide the following files / directories:

`a2.py`
: The game engine. This is *the only file you submit and modify. Do not* make changes to any other files.

`a2_support.py`
: Classes, functions, and constants to assist you in some parts of the assignment.

`maze_files`
: A collection of files to be used to initialize games.

`game_examples`
: A folder containing example output from running a full solution to the assignment.

3 Terminology & Gameplay

In Fancy Sokoban, a player appears in a *grid*, which is made up of *tiles*. Tiles may be a **wall**, **floor**, or **goal**. There are a number of movable *entities* that exist on the grid. The player (as a moveable object on the grid) is themself an **entity**. Other entities include *crates* and various types of **potions**. The objective of the game is for the player to push the crates onto the goal locations such that every goal tile is **filled** (covered by a crate). Every game has exactly as many crates as it has goals.

The player has a limited number of moves available, and starts with an initial strength. Crates also have a strength value, which represents the strength required to push them.

A player cannot move through a wall or through a crate. A crate cannot move through a wall or through any other entity (including other crates). Potions are collectable entities that can be applied to increase a player's strength and/or number of moves remaining, thereby allowing them to complete games they may otherwise be unable to complete.

At the beginning of the game, the initial game state is shown (including the grid with all entities, and player statistics). The *user* (person controlling the player) is then repeatedly prompted for a move. Valid moves and their corresponding behaviours are shown in Table 1. If a user enters anything other than these valid commands at the move prompt, the text `'Invalid move.'` should be displayed, then the game state should be displayed

and the user should be reprompted for another move.

Move name	Behaviour
'a'	The player attempts to move left one square.
'w'	The player attempts to move up one square.
's'	The player attempts to move down one square.
'd'	The player attempts to move right one square.
'q'	The player quits the game. The program should terminate gracefully.

Table 1: The behaviour of commands a user can input at the prompt for a move.

The game is over when the player has either:

- Won by placing all crates on goals (or equivalently, having all goals be filled)
- Lost by running out of moves
- Quit by entering 'q' at the move prompt.

You do not need to handle informing the user if the game has become unwinnable; you may assume they will have to quit to end the game in this case.

4 Class overview and relationships

You are *required* to implement a number of classes in this assignment. The class diagram in Figure 1 provides an overview of *all* of the classes you must implement in your assignment, and the basic relationships between them. The details of these classes and their methods are described in depth in Section 5.

You should develop the classes in the order in which they are described in Section 5, and test each one (including on Gradescope) before moving on to the next class. Functionality marks are awarded for each class (and each method) that *work correctly*. You can pass the assignment, and indeed do quite well, without implementing every class. Conversely, you will do very poorly on the assignment if you submit an attempt at every class, where no classes work according to the description. Some classes require significantly more time to implement than others. For example, `Tile` and its subclasses will likely be substantially shorter than `SokobanModel`. The marks allocated to each class are not necessarily an indication of their difficulty or the time required to complete them.

- Orange classes are classes that are provided to you in the support file.
- Green classes are *abstract* classes. However, you are not required to enforce the abstract nature of the green classes in their implementation. The purpose of this distinction is to indicate to you that *you* should only ever instantiate the blue and orange classes in your program (though you should instantiate the green classes to test them before beginning work on their subclasses).
- Hollow-arrowheads indicate *inheritance* (i.e. the "is-a" relationship).
- Dotted arrows indicates *composition* (i.e. the "has-a" relationship). An arrow marked with 1-1 denotes that each instance of the class at the base of the arrow contains exactly one instance of the class at the head of the arrow. An arrow marked with 1-n denotes that each instance of the class at the base of the arrow may contain many instances of the class at the head of the arrow.

5 Implementation

This section outlines the classes, methods, and functions that you are *required* to implement as part of your assignment. It is recommended that you implement the classes in the order in which they are described. Ensure each class behaves as per the examples (and where possible, the Gradescope tests) before moving on to the next class.

5.1 Tiles

Tiles are used to represent the game grid (i.e. the floor on which the entities exist). All instantiable tiles inherit from the abstract `Tile` class, and should inheret the default `Tile` behaviour except where specified in the descriptions of each specific type of tile.

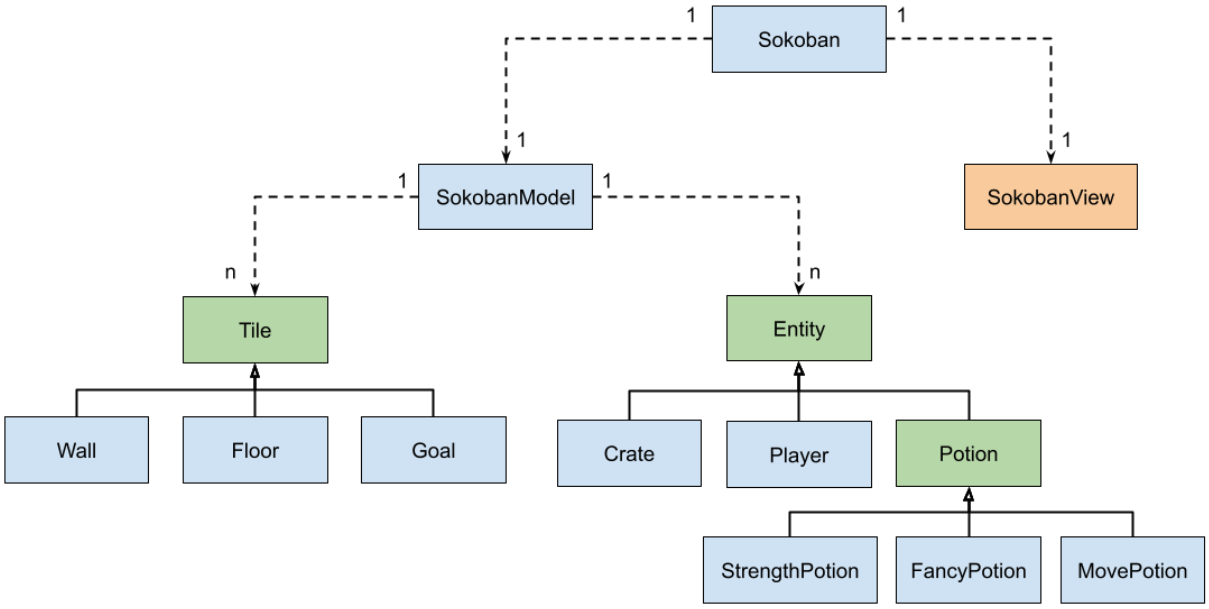

Figure 1: Basic class relationship diagram for the classes which need to be implemented for this assignment.

Tile *(abstract class)*

An abstract class from which all instantiable types of tiles inheret. Provides the default tile behaviour, which can be inhereted or overwritten by specific types of tiles. The `__init__` methods for all tiles do not take any arguments beyond `self`.

`is_blocking(self) -> bool` *(method)*

Returns `True` only when this tile is blocking. A tile is blocking if an entity would not be able to move onto that tile. By default, tiles are *non-blocking*.

`get_type(self) -> str` *(method)*

Returns a string representing the type of this tile. For the abstract `Tile` class, this method returns the string 'Abstract Tile'. For instantiable subclasses, this method should return the single letter constant corresponding to that *class*.

`__str__(self) -> str` *(method)*

Returns a string representing the type of this tile. In most cases, this will be the same string as would be returned by `get_type`.

`__repr__(self) -> str` *(method)*

Operates identically to the `__str__` method.

Examples

```
>>> tile = Tile()
>>> tile.is_blocking()
False
>>> tile.get_type()
'Abstract Tile'
>>> str(tile) # note that this is a demo of the __str__ method
'Abstract Tile'
>>> tile # note that this is a demo of the __repr__ method
Abstract Tile
```

Floor *(class)*

Inherits from Tile
`Floor` is a basic type of tile that represents an empty space on which entities can freely move. It is non-blocking and is represented by a single space character.
Examples

```
>>> floor.is_blocking()
False
>>> floor = Floor()
>>> floor.get_type()
' '
>>> str(floor)
' '
>>> floor # note that the below output contains a space character without quotation marks
```

Wall *(class)*

Inherits from Tile
`Wall` is a type of tile that represents a wall through which entities cannot pass. Wall tiles are blocking, and are represented by the character 'W'.
Examples

```
>>> wall = Wall()
>>> wall.is_blocking()
True
>>> wall.get_type()
'W'
>>> str(wall)
'W'
>>> wall
W
```

Goal *(class)*

Inherits from Tile
`Goal` is a type of tile that represents a goal location for a crate. `Goal` tiles are non-blocking, and the type is represented by 'G'. `Goal` tiles can either be filled (e.g. contain a crate) or unfilled (e.g. empty, with room for one crate). `Goal` tiles start unfilled, and become filled throughout gameplay as the player pushes crates onto them. If a goal tile is unfilled, the `__str__` and `__repr__` methods return 'G'. However, when a goal tile becomes filled, the `__str__` and `__repr__` methods should instead return 'X' to denote that this goal tile is filled. In addition to the regular `Tile` methods that the `Goal` must support, this class should also implement the following methods:

`is_filled(self) -> bool` *(method)*

Returns True only when the goal is filled.

`fill(self) -> None` *(method)*

Sets this goal to be filled.

Examples

```
>>> goal = Goal()
>>> goal.is_blocking()
False
>>> goal.get_type()
```

```
'G'
>>> str(goal)
'G'
>>> goal
G
>>> goal.is_filled()
False
>>> goal.fill()
>>> goal.is_filled()
True
>>> goal.get_type()
'G'
>>> str(goal)
'X'
>>> goal
X
```

5.2 Entities

Entities exist on top of the grid (i.e. on top of the tiles), and include the player, all crates, and all potions. Entities may or may not be movable.

Entity *(abstract class)*

Abstract base class from which all entities inherit. The `__init__` methods for this class does not take any arguments beyond `self`.

`get_type(self) -> str` *(method)*

Returns a string representing the type of this entity. For the abstract `Entity` class, this method returns the string 'Abstract Entity'. For instantiable subclasses, this method should return the single letter constant corresponding to that *class*.

`is_movable(self) -> bool` *(method)*

Returns True iff this entity is movable. By default, entities are movable.

`__str__(self) -> str` *(method)*

Returns a string representing the type of this entity. In most cases, this will be the same string as would be returned by `get_type`.

`__repr__(self) -> str` *(method)*

Operates identically to the `__str__` method.

Examples

```
>>> entity = Entity()
>>> entity.get_type()
'Abstract Entity'
>>> entity.is_movable()
False
>>> str(entity)
'Abstract Entity'
>>> entity
Abstract Entity
```

Crate *(class)*

Inherits from Entity

`Crate` is a movable entity, represented (in `get_type`) by the letter 'C'. Crates are constructed with a strength value, which represents the strength a player is required to have in order to move that crate. The string representation of a crate should be the string version of its strength value. You may assume that the strength values will always be between 0 and 9 inclusive.

Note: blocking players from moving crates that they are not strong enough to move should not be handled by the crate class. A crate only needs to be aware of its own strength requirement, and provide an interface through which the model class can access that information.

In addition to the regular `Entity` methods, the `Crate` class is required to implement the following methods:

`__init__(self, strength: int) -> None` *(method)*

Ensure any code from the `Entity` constructor is run, and set this crate's strength value to `strength`.

`get_strength(self) -> int` *(method)*

Returns this crate's strength value.

Examples

```
>>> crate = Crate(4)
>>> crate.get_type()
'C'
>>> crate.is_movable()
True
>>> str(crate)
'4'
>>> crate # Note that this is a string displaying without quotation marks
4
>>> crate.get_strength() # Note that this is an integer
4
```

Potion *(abstract class)*

Inherits from Entity
This is an abstract class which provides a simple interface which all instances of potions must implement. The `__init__` method for all potions do not take any arguments besides `self`. Since this class inherits from `Entity`, it (along with its subclasses) should also provide all methods available from the `Entity` class. Potions are not movable. An abstract potion is represented by 'Potion' and has no effect.

All potions must additionally implement the following method:

`effect(self) -> dict[str, int]` *(method)*

Returns a dictionary describing the effect this potion would have on a player. Note that potions are not responsible for *applying* their effects to a player; they only need to provide information about the effects they would cause. The abstract potion class should just return an empty dictionary, since it has no effect.

StrengthPotion *(class)*

Inherits from Entity
A `StrengthPotion` is represented by the string 'S' and provides the player with an additional 2 strength.

MovePotion *(class)*

Inherits from Entity
A `MovePotion` is represented by the string 'M' and provides the player with 5 more moves.

FancyPotion *(class)*

Inherits from Entity

A `FancyPotion` is represented by the string 'F' and provides the player with an additional 2 strength and 2 more moves.

Examples of Potions

```
>>> potion = Potion()
>>> print(potion.get_type(), potion.is_movable(), str(potion))
Potion False Potion
>>> potion.effect()
{}
>> strength = StrengthPotion()
>>> print(strength.get_type(), strength.is_movable(), strength.effect(), str(strength))
S False {'strength': 2} S
>>> move = MovePotion()
>>> print(move.get_type(), move.is_movable(), move.effect(), str(move))
M False {'moves': 5} M
>>> fancy = FancyPotion()
>>> print(fancy.get_type(), fancy.is_movable(), fancy.effect(), str(fancy))
F False {'strength': 2, 'moves': 2} F
```

Player *(class)*

Inherits from Entity

`Player` is a movable entity, represented by the letter 'P'. A player instance is constructed with a starting strength and an initial number of moves remaining. These two values can change throughout regular gameplay, or through the use of potions, via methods provided by the `Player` class. A player is only movable if they have a positive number of moves remaining. In addition to the regular `Entity` methods, the `Player` class is required to implement the following methods:

`__init__(self, start_strength: int, moves_remaining: int) -> None` *(method)*

Ensure any code from the `Entity` constructor is run, and set this player's strength to `start_strength` and their remaining moves to `moves_remaining`.

`get_strength(self) -> int` *(method)*

Returns the player's current strength value.

`add_strength(self, amount: int) -> None` *(method)*

Adds the given amount to the player's strength value.

`get_moves_remaining(self) -> int` *(method)*

Returns the player's current number of moves remaining.

`add_moves_remaining(self, amount: int) -> None` *(method)*

Adds the given amount to the player's number of remaining moves. Note that amount may be negative.

`apply_effect(self, potion_effect: dict[str, int]) -> None` *(method)*

Applies the effects described in `potion_effect` to this player.

Examples

```
>>> player = Player(1, 8)
>>> print(player.get_strength(), player.get_moves_remaining())
1 8
>>> player.add_strength(2)
>>> player.add_moves_remaining(-3)
>>> print(player.get_strength(), player.get_moves_remaining())
3 5
```

```
>>> potion = StrengthPotion()
>>> player.apply_effect(potion.effect())
>>> print(player.get_strength(), player.get_moves_remaining())
5 5
>>> potion = MovePotion()
>>> player.apply_effect(potion.effect())
>>> print(player.get_strength(), player.get_moves_remaining())
5 10
>>> potion = FancyPotion()
>>> player.apply_effect(potion.effect())
>>> print(player.get_strength(), player.get_moves_remaining())
7 12
```

5.3 convert_maze

The `read_file` function in `a2_support.py` will return a tuple containing a representation of the maze (including tiles and entitites), and the player stats (strengths and moves remaining). The maze representation is in the format `list[list[str]]`, where each string is a character representing either the tile or entity at that location. If an entity is present at a location, it is assumed that the tile underneath it is a floor tile.

You will need to convert this maze representation into more appropriate data structures containing *instances* of the classes you have just written. To do so, you must write the following *function* (as opposed to a *method* which exists within a class):

`convert_maze(game: list[list[str]]) -> tuple[Grid, Entities, Position]` *(function)*
This function converts the simple format of the maze representation into a more sophisticated representation. Namely, this function must construct the following structures:

1. A list of lists of `Tile` instances, representing the tiles on the grid.
2. A dictionary mapping (row, column) positions to `Entity` instances. This dictionary only contains positions on which entities exist, and **does not contain the player**, despite the fact that the player is an entity.
3. A tuple containing the (row, column) position of the player.

This function must then return a tuple containing these three structures (in order).

Examples

```
>>> raw_maze, player_stats = read_file('maze_files/maze1.txt')
>>> maze, entities, player_position = convert_maze(raw_maze)
>>> maze
[[W, W, W, W, W, W, W, W], [W, , , , W, , , W], [W, , , , W, , , W],
[W, , , , W, G, , W], [W, , , , , , , W], [W, , , , , , , W],
[W, W, W, W, W, W, W, W]]
>>> entities
{(3, 2): 1}
>>> player_position
(1, 1)
```

5.4 SokobanModel

The `SokobanModel` class is responsible for maintaining the game state, and applying game logic. The `SokobanModel` class must implement the following methods. Note, however, that some of these methods will become *very long*. You will likely benefit from adding extra helper methods to this class to help break up some of these long methods.

`__init__(self, maze_file: str) -> None` *(method)*

This method should read the given maze file (see `a2_support.py`), call the `convert_maze` function to get representations for the maze, non-player entities, and player position, and construct a player instance with the

player stats described in the maze file.

You may assume we will not test your code with invalid maze files. You may also assume that the maze file will not contain any goals that are already filled.

`get_maze(self) -> Grid` *(method)*

Returns the maze representation (list of lists of Tile instances).

`get_entities(self) -> Entities` *(method)*

Returns the dicitonary mapping positions to non-player entities.

`get_player_position(self) -> tuple[int, int]` *(method)*

Returns the player's current position.

`get_player_moves_remaining(self) -> int:` *(method)*

Returns the number of moves the player has remaining.

`get_player_strength(self) -> int:` *(method)*

Returns the player's current strength.

`attempt_move(self, direction: str) -> bool:` *(method)*

This method should handle trying to move the player in the given direction and any flow on effects from that move. The method should return `True` if a move occurred successfully, and `False` otherwise. The steps you must handle in this method are as follows:

1. If the direction is not a valid move direction, or the new position would be out of bounds or blocked by a blocking tile, then return `False`.

2. If the move would cause the player to move to a position containing a crate, attempt to move the crate in the given 'direction'. If the crate cannot be moved then return `False` and do not move the player. If the crate is moved onto an unfilled goal tile, remove the crate from the maze and update that goal tile to be filled. A crate cannot be moved if either of the following cases occur:

   - The player's strength is less than the strength required to push the crate.

   - The position on which the crate would move is not in bounds, or contains a blocking tile or any entity.

3. If the move would cause the player to move to a position containing a potion, remove that potion from the maze and apply its effects to the player.

4. If the move is valid according to all earlier steps, update the player's position and decrease their moves remaining by 1. Then return True to indicate that the move was a success.

`has_won(self) -> bool:` *(method)*

Returns `True` only when the game has been won. The game has been won if all goals are filled, or equivalently (since the number of goals is always equal to the number of crates), there are no more crates on the grid.

Examples

```
>>> model = SokobanModel('maze_files/maze1.txt')
>>> model.get_maze()
[[W, W, W, W, W, W, W, W], [W, , , , W, , , W], [W, , , , W, , , W],
[W, , , , W, G, , W], [W, , , , , , , W], [W, , , , , , , W],
[W, W, W, W, W, W, W, W]]
>>> model.get_entities()
{(3, 2): 1}
>>> model.get_player_position()
(1, 1)
```

```
>>> model.get_player_moves_remaining()
12
>>> model.get_player_strength()
1
>>> model.has_won()
False
>>> model.attempt_move('s')
True
>>> model.get_player_moves_remaining()
11
>>> model.get_player_position()
(2, 1)
>>> model.attempt_move('a')
False
>>> model.attempt_move('s')
True
>>> model.attempt_move('d')
True
>>> model.get_player_position()
(3, 2)
>>> model.get_entities()
{(3, 3): 1}
```

5.5 Controller

The `Sokoban` class represents the controller class in the MVC structure. It is responsible for instantiating the model class (that you just wrote) and the view class (which is provided in `a2_support.py`). The controller class handles events (such as user input) and facilitates communication between the model and view classes. The Sokoban class must implement the following methods.

`__init__(self, maze_file: str) -> None:` *(method)*

This method should construct an instance of the `SokobanModel` class using the provided `maze_file`, as well as an instance of the `SokobanView` class.

`display(self) -> None:` *(method)*

This method should call the `display_game` and `display_stats` methods on the instance of the `SokobanView` class. The arguments given should be based on the state of the game as defined by the `SokobanModel` instance.

`play_game(self) -> None:` *(method)*

This method runs the main game loop, and should implement the following behaviour:

- While the game is still going (i.e. the function has not returned), repeat the following procedure:
    1. If the game has been won, display the game state and the message 'You won!', and return.
    2. If the game has been lost, display the message 'You lost!', and return.
    3. Display the current game state.
    4. Prompt the user for a move with the prompt 'Enter move: '.
    5. If the move is 'q', return, otherwise tell the model to attempt the given move.
    6. If the move was invalid, display the message 'Invalid move\n'.

5.6 CSSE7030 Task: Undo command

Students enrolled in CSSE7030 must also implement support for an undo command. Students enrolled in CSSE1001 do not need to attempt this task to earn full marks, and cannot earn extra marks for attempting this task.

When the user enters the command 'u' at the prompt for a move, rather than considering it an invalid move your program should reverse the last *valid* move made and continue on with gameplay. In order to achieve this task, you will need to add the following methods:

- `Goal.unfill(self) -> None`: create a method which sets a goal to be unfilled.
- `SokobanModel.undo(self) -> None`: create a method in the model which reverses the effects of the last *valid* move.

You will also need to make changes to the following methods:

- `Controller.play_game`: add support for handling the input 'u' appropriately. In the flow of logic described in section 5.5 for this method, support for 'u' input would be at the beginning of step 5.

You may assume your program will only be tested with one undo at a time. That is, you may assume the user will not enter 'u' and then immediately enter 'u' again without entering a valid move in between. If the user enters 'u' after an invalid move, you should reverse the *last valid move*; that is, the last move that had any effect. You may **not** assume that the user will only enter 'u' once per game.

6 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.

6.1 Marking Breakdown

Your total grade for this assessment piece will be a combination of your functionality and style marks. For this assignment, functionality and style have equal weighting, meaning you should be devoting at least as much time towards proper styling of your code as you do trying to make it functional.

6.2 Functionality Marking

Your program's functionality will be marked out of a total of 6 marks. As in assignment 1, your assignment will be put through a series of tests and your functionality mark will be proportional to the number of tests you pass. You will be given a *subset* of the functionality tests before the due date for the assignment. You may receive partial marks within each class for partially working methods, or for implementing only a few classes. Note that you do not need to implement the model and controller classes in order to earn a passing grade for this assignment, provided that your attempts at earlier sections of the assignment are functional, well-designed, and well-styled.

You 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. Note: Functionality tests are automated, so outputs need to match *exactly* what is expected.

Your program must run in the Python interpreter (the IDLE environment). 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 3.10 or PyCharm) but not in the Python 3.11 interpreter, you will get zero for the functionality mark.

If your program cannot not run on Gradescope, you will receive **no marks** for functionality. It is your responsibility to upload to Gradescope in time to debug any issues that may cause Gradescope to be unable to

run your submission. Tutors will not fix any aspect of your code (including file names).

6.3 Style Marking

The style of your assignment will be assessed by a tutor. The style mark will be out of 4 marks. The key consideration in marking your code style is whether the code is easy to understand and demonstrates understanding of object-oriented programming concepts. 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.
  - Descriptive Identifier Names: Variable, constant, function, class and method names clearly describe what they represent in the program's logic. Do **not** use what is called the *Hungarian Notation* for identifiers. In short, this means do not include the identifier's type in its name (e.g. `item_list`), rather make the name meaningful. (e.g. Use `items`, where plural informs the reader it is a collection of items and it can easily be changed to be some other collection and not a list.) The main reason for this restriction is that most people who follow the *Hungarian Notation* convention, use it poorly (including Microsoft).
  - Named Constants: All non-trivial fixed values (literal constants) in the code are represented by descriptive named (symbolic) constants.
- Documentation
  - Comment Clarity: Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0.`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.
  - 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.
  - Description of Logic: 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.

Structure will be assessed as to how well your code design conforms to good object-oriented programming practices.

- Object-Oriented Program Structure
  - Classes & Instances: Objects are used as entities to which messages are sent, demonstrating understanding of the differences between classes and instances.
  - 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 & Polymorphism: Subclasses are designed as specialised versions of their superclasses. Subclasses extend the behaviour of their superclass without re-implementing behaviour, or breaking the superclass behaviour or design. Subclasses redefine behaviour of appropriate methods to extend the superclasses' type. Subclasses do not break their superclass' interface.
  - Model View Controller: Your program adheres to the Apple MVC structure.
- Algorithmic Logic
  - 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.
  - Variable Scope: Variables should be declared locally in the method or function in which they are

needed. Attributes should be declared clearly within the `__init__` method. Class variables are avoided, except where they simplify program logic. Global variables should not be used.

- Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. loops and conditional statements).

6.4 Documentation Requirements

There are a significant number of classes and contained methods you have to implement for this assignment. For each one, *you must provide documentation* in the form of a docstring. The only exception is for overridden methods on subclasses, as python docstrings are inherited.

6.5 Assignment Submission

This assignment follows the same assignment submission policy as assignment 1. Please refer to the assignment 1 task sheet. You must submit your assignment as a single Python file called `a2.py` (use this name – all lower case), and *nothing else*. Your submission will be automatically run to determine the functionality mark. If you submit a file with a *different name*, the tests will *fail* and you will get *zero* for functionality. Do *not* submit the `a2_support.py` file, or any other files. Do *not* submit any sort of archive file (e.g. zip, rar, 7z, etc.).

6.6 Plagiarism

This assignment follows the same plagiarism policy as assignment 1. Please refer to the assignment 1 task sheet.