Assignment 2

Introduction to Software Engineering (CSSE 1001)

Due: 30 May 2025, 15:00 GMT+10

AUTHOR
Course Staff (Communal Document)

PUBLISHED
May 27, 2025

Changelog

- 29/04/25:
  - Initial Release.
- 05/05/25:
  - Fixed a (decent) number of cosmetic typos
  - Task 1 now reads "Cards can be considered permanent, but are not by default"
  - Table 1 now correctly reads "Health" instead of "Heal"
  - Task 6, removed erroneous `**kwargs` from `__init__`
- 07/05/25:
  - Gradescope Released! Access through `Assessment -> Assignment 2` on BlackBoard.
  - Fixed a cosmetic missing line in `gameplay/loss.txt`
  - Added `levels/in_progress.txt` for your testing. Redownload `a2.zip` for this and the above fix.
  - Updated `Minion` to explicitly describe the `__str__` behavior that should be inherited.
  - Fixed a display issue, and increased clarity for `HearthModel` example.
- 12/05/2025:
  - **PLEASE READ**: Added a note to the `repr` method of the `Hearthstone` class. We have been made aware that Gradescope is expecting this `repr` not to contain quotations around the argument (i.e. `Hearthstone(file.txt)` instead of `Hearthstone("file.txt")`). While this is not correct, due to the number of submissions made to Gradescope at this point, and the fact it is a minor difference that does not impact the rest of the assignment, we have elected not to change what Gradescope is expecting. Please write this method to satisfy Gradescope.
- 27/05/25:
  - Minor update to order of `play card` in Table 2 to match what Gradescope and examples are expecting.

Introduction

Hearthstone is a strategy card game developed by Blizzard Entertainment. In this assignment, you will be implementing a text based game (very) loosely based on this title. Note that where the behaviour of the original game differs from this specification, this specification takes precedence, and you should implement the assignment as per it. In our version of Hearthstone, the player is a Hero tasked with defeating an enemy computer controlled Hero. Heroes battle by casting spells represented by cards which they draw from their personal card decks. These spells can summon minions to fight for their summoner, or directly attack (or defend) a Hero or minion.

Getting Started

Download `a2.zip` from Blackboard. This archive contains the necessary files to start your assignment. Once extracted, the `a2.zip` archive will provide the following files:

- `a2.py`: *This is the only file you will submit* and is where you write your code. *Do not* make changes to any other files.
- `support.py`: *Do not modify or submit this file*, it contains pre-defined constants to use in your assignment. In addition to these, you are encouraged to create your own constants in `a2.py` where possible.
- `display.py`: *Do not modify or submit this file*, it contains view classes to display your assignment in a visually appealing manner.
- `gameplay/`: This folder contains a number of example outputs generated by playing the game using a fully-functional completed solution to this assignment. The purpose of the files in this folder is to help you understand how the game works, and how output should be formatted.
- `levels/`: This folder contains a small collection of files used to initialize games of HearthStone. In addition to these, you are encouraged to create your own files to help test your implementation where possible.

Gameplay

This section provides a high-level overview of gameplay, and is intended to provide you with an idea of the behaviour of a fully completed assignment. Where interactions are not explicitly mentioned in this section, please see the Implementation section.

> **Important**
>
> Do not simply implement the behaviour described here using your own structure; you *must* implement the individual functions, classes, and methods described in the Implementation section. Inputs and outputs must match *exactly* what is expected. Refer to the

> Implementation section, the given examples, and the provided Gradescope tests for clarification on required prompts and outputs.

The game focusses on conflict between Two *heroes*, one controlled by the player, and one controlled by the computer. Heroes are a type of *entity*. All entities have a health and shield value. These values can be depleted by receiving *damage*. If an entity has a positive shield value, then its health cannot be depleted in this way. An entity is *alive* while its health value is above zero; it is *defeated* when its health drops to zero.

Heroes also possess an *energy* level. Energy is a currency used to perform actions. A hero's energy cannot exceed that hero's *energy capacity*. Which is a value that gradually increases as the game progresses. Actions in Hearthstone are represented by *cards*. Heroes possess a *deck* of cards from which they draw into a *hand*. Cards can then be *played* from the hero's hand in order to perform actions. In addition to the regular entity rules, a hero is also defeated if its deck contains zero cards. Heroes take turns playing cards, and at the start of a turn a hero draws from the *top* of their deck until their hand is *full* (that is contains the maximum number of cards). A hero can hold a maximum of 5 cards in their hand. Each hero can play any number of cards on their turn.

In addition to a name and description, all cards have a *cost* and an *effect*. A cards cost is the energy that a hero must spend in order to play it. If a hero does not have enough energy to spend, it cannot play a given card. Instead of playing a card normally, a hero can choose to discard a given card. This moves the card from their hand to the *bottom* of their deck. A cards effect is what happens when that card is played. The types of effect are given in Table 1.

Table 1: Types of effect. A card may have zero, one or multiple effect types.

Effect	Impact on target entity
Damage	For each point of damage, the opponent's shield is reduced by 1. If the opponent's shield is reduced to 0 and there is still damage remaining, the opponent's health is reduced by the remaining amount.
Shield	Increase entity's shield by the given value
Health	Increase entity's health by the given value

Usually, a cards effect requires selecting a *target* entity. The cards effect is then inflicted on the targeted entity, and the card is then removed from the game. However, some cards are *permanent*, and have special rules when played. When a permanent card is played, it is placed in a *minion slot* in front of the hero that played it. Each hero has set number of minion slots and they are filled from left to right. If a permanent card is to be placed but all minion slots

and they are filled from left to right. If a permanent card is to be placed but all minion slots are full, the card in that hero's left most minion slot is removed from play, and all that hero's minions are moved one slot left before the card is placed. A hero has 5 minion slots. Permanent cards in minion slots are Entities, and can be targeted by card effects. At the end of a hero's turn, each permanent card in their minion slot (in order from left to right) independently selects a target and applies their effect to them. When a permanent card in a minion slot is defeated it is immediately removed from the game, and any permanent cards in slots to its right are moved left.

The game ends when one of the following occurs:

- The player *wins*, when their hero is not defeated, and their opponents hero is defeated.
- The player *loses*, when their hero is defeated.

After loading a game of Hearthstone from a given file, the player is welcomed and the following game loop occurs:

1. The player is repeatedly prompted to enter a command until they enter a valid command. Valid commands are given in Table 2
2. Action is taken according to the command entered by the player. The appropriate actions are given in Table 2
3. The display is updated, informing the player of the results of their action.
4. If the game is over, gameplay advances immediately to Step 5. Otherwise, gameplay returns to Step 1.
5. The display is updated, informing the player of the game outcome (win or loss).
6. The program exits gracefully.

Table 2: Valid commands and the actions that should be taken. Commands are case insensitive. If the command entered by the user does not match one of the commands in this table, then no action should be taken, the display should be updated informing the user they entered an invalid command, and the program should return immediately to step 1. Constants for informing the player the result of their command can be found in `support.py`.

Command	Action to take
`Help`	The display is updated, providing the player with a list of valid commands.
`Play X`, where `X` is an integer between 1 and the number of cards in the player's hand (inclusive)	The card at position `X` in the player's hand is selected. If the card is not permanent, the player is repeatedly prompted to select an entity until they select a valid entity.Then, if the player's current energy is below the cards cost, nothing happens. Otherwise, the player's current energy is reduced by the cards cost and the card is removed from the player's hand. Once the card is removed from the

	players hand, if it is permanent, it is placed in the player's next available minion slot with 1 health and 0 shield. Otherwise, the card's effects are then applied to the entity selected earlier.
`Discard X`, where `X` is an integer between 1 and the number of cards in the player's hand (inclusive)	The card at position `X` in the player's hand is removed from the player's hand, and added to the bottom of the player's deck.
`Load F` where `F` is the name of a file containing a game state.	If `F` is not an existing file, nothing happens. Otherwise, an attempt is made to load a new game state from `F`. If the attempt is successful, the game state is updated to that contained in `F`, otherwise nothing happens.
`End turn`	Each of the player's minions select a target and apply their effects to it. The enemy hero then (in order): Registers another turn on every Fireball card in their hand, draws from the top of their deck until their hand is full, Increases their energy capacity by 1, sets their energy level to their energy capacity. If the enemy hero is defeated, nothing more happens. Otherwise, the enemy hero then selects cards and plays them (The player will be informed what cards were played next time the display is updated). Each of the enemy's minions then select a target and apply their effects to it. Finally, the player (in order): Registers another turn on each Fireball card in their hand, draws from the top of their deck until their hand is full, increases their energy capacity by 1, and sets their energy level to their energy capacity. Afterwards, If the game has not ended, the game state is saved to `autosave.txt`.

Implementaion

> **Important**
>
> You are **not** permitted to add any additional import statements to `a2.py`. Doing so will result in a deduction of up to 100% of your mark. You must not modify or remove the import statements already provided to you in `a2.py`. Removing or modifying these existing import statements may result in your code not functioning, and may result in a deduction of up to 100% of your mark.

Required Classes and Methods

This section outlines the classes and methods you are **required** to implement in `a2.py`. Your program must operate *exactly* as specified. In particular, your program's output must match *exactly* with the expected output. Your program will be marked automatically so minor differences in output (such as whitespace or casing) *will* cause tests to fail resulting in *zero marks* for that test.

You will be following the Apple Model-View-Controller design pattern when implementing this assignment. You have been provided the view component, and are required to implement a number of classes in order to complete the Model and Controller components.

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 later in this section. Within Figure 1:

- Orange classes are those provided to you in the supporting files.
- Green classes are abstract classes (superclasses). 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 classes in your program (though you should instantiate the green classes to test them before beginning work on their subclasses).
- Blue classes are concrete classes (subclasses).
- Solid arrows indicate inheritance (i.e. the "is-a" relationship).
- Dotted arrows indicate 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.

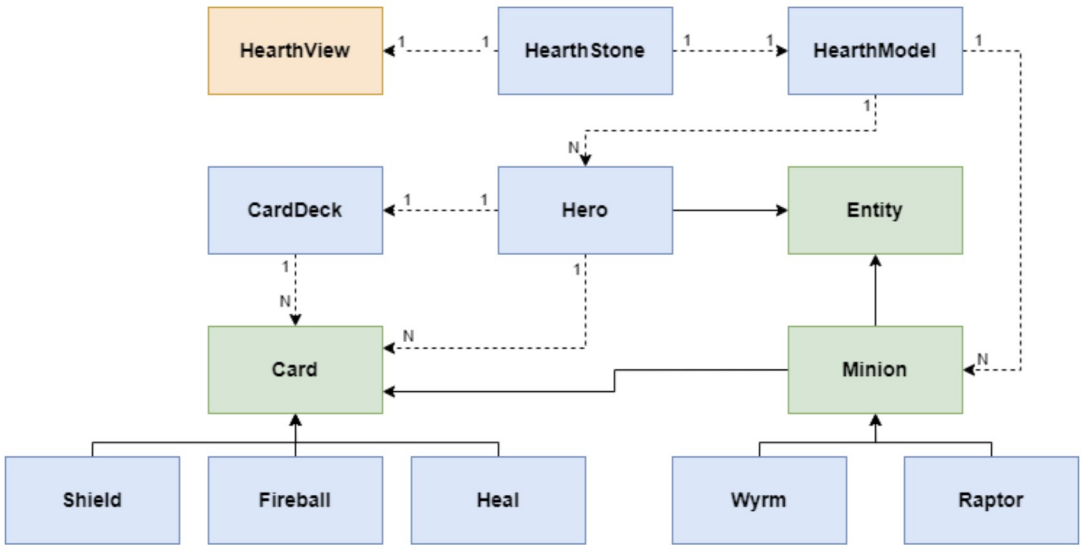

Figure 1: Basic UML diagram for the classes in assignment 2.

> **Note**
>
> You are awarded marks for the number of tests passed by your classes when they are tested *independently* of one another. Thus an incomplete assignment with *some* working classes may well be awarded more marks than a complete assignment with faulty classes.

Each class is accompanied with some examples for usage to help you *start* your own testing. You should also test your methods with other values to ensure they operate according to the descriptions.

The rest of this section describes what you must implement in `a2.py` in detail. You should complete the model section before attempting the controller section, ensuring that everything you implement is tested thoroughly, operating correctly, and passes all relevant Gradescope tests. You will not be able to earn marks for the controller section until you have passed all Gradescope tests for the model section.

Model

The following are the classes and methods you are required to implement as part of the model. You should develop the classes in the order in which they are described in this section 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 will likely do very poorly 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. The marks allocated to each class are not necessarily an indication of their difficulty or the time required to complete them. You are allowed (and encouraged) to write additional helper methods for any class to help break up long methods, but these helper methods MUST be private (i.e. they must be named with a leading underscore).

Task 1 `Card()`

`Card` is an abstract class from which all instantiated types of card inherit. This class provides default card behavior, which can be inherited or overridden by specific types of cards. All cards have a name, description, cost and effect. A card's effect is a dictionary mapping strings to integers, where the string represents the type of effect and the integer represents the strength of the effect. Cards can be considered permanent, but are not by default. Each type of card is represented by a symbol, typically a single character.

Abstract cards have no effects, have 1 cost, and are represented by the symbol `C`.

`Card` must implement the following methods:

- `__init__(self, **kwargs)` Instantiate a new card. The `**kwargs` is required for multiple inheritance in a later task, you do not need to understand it yet and can ignore it for now.
- `__str__(self) -> str`
  Returns the name and description of this card
- `__repr__(self) -> str`
  Returns a string which could be copied and pasted into a REPL to construct a new instance identical to self.
- `get_symbol(self) -> str`
  Returns the symbol representing this card.
- `get_name(self) -> str`
  Returns the name of this card.
- `get_cost(self) -> int`
  Returns the cost of this card.
- `get_effect(self) -> dict[str, int]`
  Returns this card's effect.
- `is_permanent(self) -> bool` Returns if this card is permanent or not.

Example:

```
>>> card = Card()
>>> card
```

```
Card()
```

```
>>> str(card)
```

```
'Card: A card.'
```

```
>>> card.get_symbol()
```

```
'C'
```

```
>>> card.get_name()
```

```
'Card'
```

```
>>> card.get_cost()
```

```
1
```

```
>>> card.get_effect()
```

```
{}
```

```
>>> card.is_permanent()
```

```
False
```

Task 2 `Shield(Card)`

`Shield` is a card that applies 5 shield to a target entity. Shield cards cost 1, and are represented by the symbol `S`

Example:

```
>>> card = Shield()
>>> card
```

```
Shield()
```

```
>>> str(card)
```

```
'Shield: Cast a protective shield that can absorb 5 damage.'
```

```
>>> card.get_symbol()
```

```
'S'
```

```
>>> card.get_name()
```

```
'Shield'
```

```
>>> card.get_cost()
```

```
1
```

```
>>> card.get_effect()
```

```
{'shield': 5}
```

```
>>> card.is_permanent()
```

```
False
```

Task 3 *`Heal(Card)`*

`Heal` is a card that applies 2 heal to a target entity. Heal cards cost 2, and are represented by the symbol `H`

Example:

```
>>> card = Heal()
>>> card
```

```
Heal()
```

```
>>> str(card)
```

```
'Heal: Cast an aura on target. It recovers 2 health.'
```

```
>>> card.get_symbol()
```

```
'H'
```

```
>>> card.get_name()
```

```
'Heal'
```

```
>>> card.get_cost()
```

```
2
```

```
>>> card.get_effect()
```

```
{'health': 2}
```

```
>>> card.is_permanent()
```

```
False
```

Task 4 `Fireball(Card)`

`Fireball` is a card that applies 3 damage to a target entity. Fireball cards apply 1 point of additional damage for each turn they have spent in a hero's hand. Fireball cards cost 3. These cards are not represented by a single letter, but instead their symbol is the integer number of turns they have spent in hand.

To support this extended functionality, The `__init__` method of `Fireball` takes in an additional argument. In addition to the `Card` methods that must be supported, `Fireball` must implement the following methods:

- `__init__(self, turns_in_hand: int)`
  Initialise a new `Fireball` instance, with the given number of turns spent in hand.
- `increment_turn(self)`
  Register another turn spent in a hero's hand.

Example:

```
>>> card = Fireball(7)
>>> card
```

```
Fireball(7)
```

```
>>> str(card)
```

```
'Fireball: FIREBALL! Deals 3 + [turns in hand] damage. Currently dealing 10 damage.'
```

```
>>> card.get_symbol()
```

```
'7'
```

```
>>> card.get_name()
```

```
'Fireball'
```

```
>>> card.get_cost()
```

```
3
```

```
>>> card.get_effect()
```

```
{'damage': 10}
```

```
>>> card.is_permanent()
```

```
False
```

```
>>> card.increment_turn()
>>> card
```

```
Fireball(8)
```

```
>>> card.get_symbol()
```

```
'8'
```

```
>>> card.get_effect()
```

```
{'damage': 11}
```

Task 5 `CardDeck()`

`CardDeck` represents an ordered deck of cards. Cards are drawn from the top of a deck, and added to the bottom.

`CardDeck` must implement the following methods:

- `__init__(self, cards: list[Card])`
  Initialise a deck of cards containing the given cards. Cards are provided in order, with the

Initialise a deck of cards containing the given cards. Cards are provided in order, with the first card in the list being the topmost card of the deck, and the last card in the list being the bottom most card of the deck.

- `__str__(self) -> str`
  Returns a comma separated list of the symbols representing each card in the deck. Symbols should appear in order, from top to bottom.
- `__repr__(self) -> str`
  Returns a string which could be copied and pasted into a REPL to construct a new instance identical to self.
- `is_empty(self) -> bool`
  Returns if this `CardDeck` is empty or not.
- `remaining_count(self) -> int` Returns how many cards are currently in this `CardDeck`.
- `draw_cards(self, num: int) -> list[Card]` Draws the specified number of cards from the top of the deck. Cards should be returned in the order they are drawn. If there are not enough cards remaining in the deck, as many cards as possible should be drawn.
- `add_card(self, card: Card)`
  Adds the given card to the bottom of the deck.

Example:

```
>>> cards = [Card(), Card(), Shield(), Heal(), Fireball(6)]
>>> deck = CardDeck(cards)
>>> deck
```

```
CardDeck([Card(), Card(), Shield(), Heal(), Fireball(6)])
```

```
>>> str(deck)
```

```
'C,C,S,H,6'
```

```
>>> deck.remaining_count()
```

```
5
```

```
>>> deck.is_empty()
```

```
False
```

```
>>> deck.draw_cards(3)
```

```
[Card(), Card(), Shield()]
```

```
>>> deck.remaining_count()
```

```
2
```

```
>>> str(deck)
```

```
'H,6'
```

```
>>> deck.add_card(Fireball(5))
```

```
>>> deck.remaining_count()
```

```
3
```

```
>>> str(deck)
```

```
'H,6,5'
```

```
>>> deck.draw_cards(1001)
```

```
[Heal(), Fireball(6), Fireball(5)]
```

```
>>> str(deck)
```

```
''
```

```
>>> deck.is_empty()
```

```
True
```

Task 6 `Entity()`

`Entity` is an abstract class from which all instantiated types of entity inherit. This class

provides default entity behavior, which can be inherited or overridden by specific types of entities. Each entity has a health and shield value, and are alive if and only if their health value is above 0.

`Entity` must implement the following methods:

- `__init__(self, health: int, shield: int)`
  Initialise a new entity with the given health and shield value.
- `__repr__(self) -> str`
  Returns a string which could be copied and pasted into a REPL to construct a new instance identical to self.
- `__str__(self) -> str`
  Returns this hero's health and shield, comma separated.
- `get_health(self) -> int`
  Returns this entity's health.
- `get_shield(self) -> int`
  Returns this entity's shield.
- `apply_shield(self, shield: int)`
  Applies the given amount of shield as per Table 1.
- `apply_health(self, health: int)`
  Applies the given amount of health as per Table 1.
- `apply_damage(self, damage: int)`
  Applies the given amount of damage as per Table 1. This function should not allow the entity's health to drop below 0. Any damage exceeding the amount required to reduce the entity's health to 0 should be discarded.
- `is_alive(self) -> bool`
  Returns if this entity is alive or not.

Example:

```
>>> entity = Entity(5,3)
>>> entity
```

```
Entity(5, 3)
```

```
>>> str(entity)
```

```
'5,3'
```

```
>>> entity.get_health()
```

```
5
```

```
>>> entity.get_shield()
```

```
3
```

```
>>> entity.apply_shield(1)
>>> entity
```

```
Entity(5, 4)
```

```
>>> entity.apply_health(10)
>>> entity
```

```
Entity(15, 4)
```

```
>>> entity.apply_damage(10)
>>> entity
```

```
Entity(9, 0)
```

```
>>> entity.is_alive()
```

```
True
```

```
>>> entity.apply_damage(9999999999)
>>> entity.get_health()
```

```
0
```

```
>>> entity.is_alive()
```

```
False
```

Task 7 `Hero(Entity)`

A `Hero` is an entity with the agency to take actions in the game, possessing an energy level (and corresponding energy capacity), a deck of cards, and a hand of cards. When a hero is instantiated, its energy level is always at maximum capacity. A hero's maximum hand size is 5. Unlike a base entity, a hero is only alive when *both* its health *and* the number of cards in its deck are greater than 0.

In addition to the `Entity` methods that must be supported, `Hero` must implement the following methods:

- `__init__(self, health: int, shield: int, max_energy: int, deck: CardDeck, hand: list[Card])`
  Instantiate a new `Hero` with the given health, shield, energy_capacity, deck, and hand.
- `__str__(self) -> str`
  Returns a string containing the following: This hero's health, shield, and energy capacity, comma separated; followed by a semi-colon; followed by the string representation of this hero's deck; followed by another semicolon; followed finally by the symbols of each card in this hero's hand in order, comma separated.
- `get_energy(self) -> int`
  Returns this hero's current energy level.
- `spend_energy(self, energy: int) -> bool`
  Attempts to spend the specified amount of this hero's energy. If this hero does not have sufficient energy, then nothing happens. Returns whether the energy was spent or not.
- `get_max_energy(self) -> int`
  Returns this hero's energy capacity.
- `get_deck(self) -> CardDeck`
  Returns this hero's deck.
- `get_hand(self) -> list[Card]`
  Returns this hero's hand, in order.
- `new_turn(self)`
  Registers a new turn of all fireball cards in this hero's hand, draws from their deck into their hand, expands their energy capacity by 1, and refills their energy level, as per the Gameplay Section

Example:

```
>>> cards = [Card(), Card(), Shield(), Heal(), Fireball(6)]
>>> deck = CardDeck(cards)
>>> hand = [Heal(),Heal(),Fireball(2)]
>>> hero = Hero(4,5,3,deck,hand)
>>> hero
```

```
Hero(4, 5, 3, CardDeck([Card(), Card(), Shield(), Heal(), Fireball(6)]), [Heal(),
Heal(), Fireball(2)])
```

```
>>> str(hero)
```

```
'4,5,3;C,C,S,H,6;H,H,2'
```

```
>>> hero.get_energy()
```

```
3
```

```
>>> hero.get_max_energy()
```

```
3
```

```
>>> hero.spend_energy(123456789)
```

```
False
```

```
>>> hero.get_energy()
```

```
3
```

```
>>> hero.spend_energy(2)
```

```
True
```

```
>>> hero.get_energy()
```

```
1
```

```
>>> hero.get_max_energy()
```

```
3
```

```
>>> hero.get_deck()
```

```
CardDeck([Card(), Card(), Shield(), Heal(), Fireball(6)])
```

```
>>> hero.get_hand()
```

```
[Heal(), Heal(), Fireball(2)]
```

```
>>> hero.new_turn()
>>> hero.get_energy()
```

```
4
```

```
>>> hero.get_max_energy()
```

```
4
```

```
>>> hero.get_deck()
```

```
CardDeck([Shield(), Heal(), Fireball(6)])
```

```
>>> hero.get_hand()
```

```
[Heal(), Heal(), Fireball(3), Card(), Card()]
```

Task 8 `Minion(Card, Entity)`

`Minion` is an abstract class from which all instantiated types of minion inherit. This class provides default minion behavior, which can be inherited or overridden by specific types of minions. Minions are a special type of `Card` that also inherits from `Entity`. Its `__init__` method takes in the arguments of the `Entity` class. You may need to modify your `__init__` method in the `Card` class to support multiple inheritance. The string representation of a `Minion` should be that of the `Card` class.

All minions are permanent cards. Generic Minions have cost 2, no effect, and are represented by the symbol `M`.

A minion has the capacity to select its own target entity out of a given set. Generic minions

ignore all given entities, and returns itself.

In addition to the `Entity` and `Card` methods that must be supported, `Minion` must implement the following methods:

- `__init__(self, health, shield)`
  Instantiate a new `Minion` with the specified health and shield value.
- `__str__(self) -> str`
  Returns the name and description of this card
- `choose_target(self, ally_hero: Entity, enemy_hero: Entity, ally_minions: list[Entity], enemy_minions: list[Entity]) -> Entity`
  Select this minion's target out of the given entities. Note that here, the allied hero and minions will be those friendly to this minion, not necessarily to the player. This logic extends to the specified enemy hero and minions. minions should be provided in the order they appear in their respective minion slots, from left to right.

Example:

```
>>> minion = Minion(1,0)
>>> minion
```

```
Minion(1, 0)
```

```
>>> str(minion)
```

```
'Minion: Summon a minion.'
```

```
>>> minion.get_symbol()
```

```
'M'
```

```
>>> minion.get_name()
```

```
'Minion'
```

```
>>> minion.get_cost()
```

```
2
```

```
>>> minion.get_effect()
```

```
{}
```

```
>>> minion.is_permanent()
```

```
True
```

```
>>> minion.get_health()
```

```
1
```

```
>>> minion.get_shield()
```

```
0
```

```
>>> minion.apply_shield(1)
>>> minion
```

```
Minion(1, 1)
```

```
>>> minion.apply_health(10)
>>> minion
```

```
Minion(11, 1)
```

```
>>> minion.apply_damage(10)
>>> minion
```

```
Minion(2, 0)
```

```
>>> minion.is_alive()
```

```
True
```

```
>>> hero_friend = Hero(1, 2, 3, CardDeck([Fireball(8)]), [])
```

```
>>> hero_friend = Hero(1,2,3,CardDeck([Fireball(8)]), [])
>>> hero_foe = Hero(3,2,1,CardDeck([Card()]), [Shield()])
>>> minions_friend = [Minion(1,0),Minion(2,0),Minion(1,2)]
>>> minions_foe = [Minion(1,1),Minion(2,2)]
minion.choose_target(hero_friend, hero_foe, minions_friend, minions_foe)
```

```
Minion(2, 0)
```

Task 9 `Wyrm(Minion)`

A `Wyrm` is a minion that has 2 cost, is represented by the symbol `W`, and whose effect is to apply 1 heal and 1 shield.

When selecting a target entity, a `Wyrm` will choose the allied entity with the lowest health.

If multiple entities have the lowest health, if one of the tied entities is the allied hero, the allied hero should be selected. Otherwise, the leftmost tied minion should be selected.

Example:

```
>>> minion = Wyrm(2,2)
>>> minion
```

```
Wyrm(2, 2)
```

```
>>> str(minion)
```

```
'Wyrm: Summon a Mana Wyrm to buff your minions.'
```

```
>>> minion.get_symbol()
```

```
'W'
```

```
>>> minion.get_name()
```

```
'Wyrm'
```

```
>>> minion.get_cost()
```

```
2
```

```
>>> minion.get_effect()
```

```
{'health': 1, 'shield': 1}
```

```
>>> minion.is_permanent()
```

```
True
```

```
>>> hero_friend = Hero(1,2,3,CardDeck([Fireball(8)]), [])
>>> hero_foe = Hero(3,2,1,CardDeck([Card()]), [Shield()])
>>> minions_friend = [Minion(1,0),Minion(2,0),Minion(1,2)]
>>> minions_foe = [Minion(1,1),Minion(2,2)]
minion.choose_target(hero_friend, hero_foe, minions_friend, minions_foe)
```

```
Hero(1, 2, 3, CardDeck([Fireball(8)]), [])
```

Task 10 `Raptor(Minion)`

A `Raptor` is a minion that has 2 cost, is represented by the symbol `R`, and whose effect is to apply damage equal to its health.

When selecting a target entity, a `Raptor` will choose the enemy minion with the highest health. If there is no such minion, it will select the enemy hero.

If multiple minions have the highest health, the leftmost tied minion should be selected.

Example:

```
>>> minion = Raptor(2,2)
>>> minion
```

```
Raptor(2, 2)
```

```
>>> str(minion)
```

```
'Raptor: Summon a Bloodfen Raptor to fight for you.'
```

```
>>> minion.get_symbol()
```

```
'R'
```

```
>>> minion.get_name()
```

```
'Raptor'
```

```
>>> minion.get_cost()
```

```
2
```

```
>>> minion.get_effect()
```

```
{'damage': 2}
```

```
>>> minion.apply_health(2)
```

```
>>> minion.get_effect()
```

```
{'damage': 4}
```

```
>>> minion.is_permanent()
```

```
True
```

```
>>> hero_friend = Hero(1,2,3,CardDeck([Fireball(8)]), [])
>>> hero_foe = Hero(3,2,1,CardDeck([Card()]), [Shield()])
>>> minions_friend = [Minion(1,0),Minion(2,0),Minion(1,2)]
>>> minions_foe = [Minion(1,1),Minion(2,2)]
minion.choose_target(hero_friend, hero_foe, minions_friend, minions_foe)
```

```
Minion(2, 2)
```

```
minion.choose_target(hero_friend, hero_foe, minions_friend, [])
```

```
Hero(3, 2, 1, CardDeck([Card()]), [Shield()])
```

Task 11 `HearthModel()`

`HearthModel` models the logical state of a game of Hearthstone.

Here, active minions are those minions currently existing within a minion slot. Both the player and their opponent have a maximum of 5 minion slots. Minion slots are filled out from left to right. If a minion is to be placed and all respective minion slots are full, the minion in the leftmost minion slot is removed from the game, and all minions in remaining slots are moved one slot left before the new minion is placed.

Within this model, the enemy hero follows the following logic: When the enemy hero takes a turn, it attempts to play each card in its hand in order. Whenever it successfully plays a card, it begins trying cards from the beginning of its hand again. If the enemy plays a card that includes a damage effect, it always targets the player's hero. Otherwise it targets itself.

`HearthModel` must implement the following methods:

- `__init__(self, player: Hero, active_player_minions: list[Minion], enemy: Hero, active_enemy_minions: list[Minion])`
  Instantiates a new `HearthModel` using the given player, enemy, and active minions. Each respective list of minions is given in the order they appear in their corresponding minion slots, from left to right.
- `__str__(self) -> str`
  Return the following in order, separated by the pipe character (`|`): The string representation of the player's hero; a semicolon separated list of the players active minions (symbol, health, and shield, comma separated); the string representation of the enemy hero; and a semicolon separated list of the active enemy minions (symbol, health, and shield, comma separated).
- `__repr__(self) -> str`
  Returns a string which could be copied and pasted into a REPL to construct a new instance identical to self.
- `get_player(self) -> Hero`
  Return this model's player hero instance.
- `get_enemy(self) -> Hero`
  Return this model's enemy hero instance.
- `get_player_minions(self) -> list[Minion]`
  Return the player's active minions. Minions should appear in order from leftmost minion slot to rightmost minion slot.
- `get_enemy_minions(self) -> list[Minion]`
  Return the enemy's active minions. Minions should appear in order from leftmost minion slot to rightmost minion slot.

- `has_won(self) -> bool`
  Return true if and only if the player has won the game as per the Gameplay section.
- `has_lost(self) -> bool`
  Return true if and only if the player has lost the game as per the Gameplay section
- `play_card(self, card: Card, target: Entity) -> bool`
  Attempts to play the specified card on the player's behalf. Table 2 Specifies the actions that must occur when an attempt is made to play a card. Returns whether the card was successfully played or not. The `target` argument will be ignored if the specified card is permanent. If a minion is defeated, it should be removed from the game, and any remaining minions within the respective minion slots should be moved one slot left if able.
- `discard_card(self, card: Card)`
  Discards the given card from the players hand. The discarded card should be added to the bottom of the player's deck as per Table 2.
- `end_turn(self) -> list[str]`
  Follows the instructions for the `end turn` command in Table 2, excluding the last instruction (saving the game to `autosave.txt`). Returns the names of the cards played by the enemy hero (in order). If a minion is defeated at any point, it should be removed from the game, and any remaining minions within the respective minion slots should be moved one slot left if able. If the enemy hero is not alive after it has drawn cards, it should not take a turn, and the player should not subsequently update its own status.

Example:

```
>>> deck1 = CardDeck([Shield(),Heal(),Fireball(3),Heal(),Raptor(1,0),Wyrm(1,0),Shield
>>> hand1 = [Raptor(2,2), Heal(), Shield(),Fireball(8)]
>>> player = Hero(5,0,2,deck1,hand1)
>>> deck2 = CardDeck([Heal(),Shield(),Heal(),Heal(),Raptor(1,2),Wyrm(1,3),Shield(),He
>>> hand2 = [Wyrm(1,0),Fireball(0),Raptor(1,0),Shield()]
>>> enemy = Hero(10,0,3,deck2,hand2)
>>> player_minions = [Raptor(1,0),Wyrm(1,1)]
>>> enemy_minions = [Wyrm(1,2)]
>>> model = HearthModel(player,player_minions,enemy,enemy_minions)
>>> model
```

```
HearthModel(Hero(5, 0, 2, CardDeck([Shield(), Heal(), Fireball(3), Heal(), Raptor(1,
0), Wyrm(1, 0), Shield(), Heal(), Heal(), Raptor(1, 0)]), [Raptor(2, 2), Heal(),
Shield(), Fireball(8)]), [Raptor(1, 0), Wyrm(1, 1)], Hero(10, 0, 3, CardDeck([Heal(),
Shield(), Heal(), Heal(), Raptor(1, 2), Wyrm(1, 3), Shield(), Heal(), Heal(),
Raptor(2, 2)]), [Wyrm(1, 0), Fireball(0), Raptor(1, 0), Shield()]), [Wyrm(1, 2)])
```

```
>>> str(model)
```

```
'5,0,2;S,H,3,H,R,W,S,H,H,R;R,H,S,8|R,1,0;W,1,1|10,0,3;H,S,H,H,R,W,S,H,H,R;W,0,R,S|W,1,2'
```

```
>>> model.get_player()
```

```
Hero(5, 0, 2, CardDeck([Shield(), Heal(), Fireball(3), Heal(), Raptor(1, 0), Wyrm(1, 0), Shield(), Heal(), Heal(), Raptor(1, 0)]), [Raptor(2, 2), Heal(), Shield(), Fireball(8)])
```

```
>>> model.get_enemy()
```

```
Hero(10, 0, 3, CardDeck([Heal(), Shield(), Heal(), Heal(), Raptor(1, 2), Wyrm(1, 3), Shield(), Heal(), Heal(), Raptor(2, 2)]), [Wyrm(1, 0), Fireball(0), Raptor(1, 0), Shield()])
```

```
>>> model.get_player_minions()
```

```
[Raptor(1, 0), Wyrm(1, 1)]
```

```
>>> model.get_enemy_minions()
```

```
[Wyrm(1, 2)]
```

```
>>> model.has_won()
```

```
False
```

```
>>> model.has_lost()
```

```
False
```

```
>>> card = model.get_player().get_hand()[3]
>>> card
```

```
Fireball(8)
```

```
>>> model.play_card(card, enemy)
```

```
False
```

```
>>> model
```

```
HearthModel(Hero(5, 0, 2, CardDeck([Shield(), Heal(), Fireball(3), Heal(), Raptor(1,
0), Wyrm(1, 0), Shield(), Heal(), Heal(), Raptor(1, 0)]), [Raptor(2, 2), Heal(),
Shield(), Fireball(8)]), [Raptor(1, 0), Wyrm(1, 1)], Hero(10, 0, 3, CardDeck([Heal(),
Shield(), Heal(), Heal(), Raptor(1, 2), Wyrm(1, 3), Shield(), Heal(), Heal(),
Raptor(2, 2)]), [Wyrm(1, 0), Fireball(0), Raptor(1, 0), Shield()]), [Wyrm(1, 2)])
```

```
>>> card = model.get_player().get_hand()[2]
>>> card
```

```
Shield()
```

```
>>> model.play_card(card, player)
```

```
True
```

```
>>> model.get_player().get_energy() # Note that repr below depicts energy capacity (u
```

```
1
```

```
>>> model
```

```
HearthModel(Hero(5, 5, 2, CardDeck([Shield(), Heal(), Fireball(3), Heal(), Raptor(1,
0), Wyrm(1, 0), Shield(), Heal(), Heal(), Raptor(1, 0)]), [Raptor(2, 2), Heal(),
Fireball(8)]), [Raptor(1, 0), Wyrm(1, 1)], Hero(10, 0, 3, CardDeck([Heal(), Shield(),
Heal(), Heal(), Raptor(1, 2), Wyrm(1, 3), Shield(), Heal(), Heal(), Raptor(2, 2)]),
[Wyrm(1, 0), Fireball(0), Raptor(1, 0), Shield()]), [Wyrm(1, 2)])
```

```
>>> card = model.get_player().get_hand()[0]
>>> card
```

```
Raptor(2, 2)
```

```
>>> model.discard_card(card)
>>> model
```

```
HearthModel(Hero(5, 5, 2, CardDeck([Shield(), Heal(), Fireball(3), Heal(), Raptor(1,
0), Wyrm(1, 0), Shield(), Heal(), Heal(), Raptor(1, 0), Raptor(2, 2)]), [Heal(),
Fireball(8)]), [Raptor(1, 0), Wyrm(1, 1)], Hero(10, 0, 3, CardDeck([Heal(), Shield(),
Heal(), Heal(), Raptor(1, 2), Wyrm(1, 3), Shield(), Heal(), Heal(), Raptor(2, 2)]),
[Wyrm(1, 0), Fireball(0), Raptor(1, 0), Shield()]), [Wyrm(1, 2)])
```

```
>>> model.end_turn()
```

```
['Wyrm', 'Raptor']
```

```
>>> model
```

```
HearthModel(Hero(5, 5, 3, CardDeck([Heal(), Raptor(1, 0), Wyrm(1, 0), Shield(),
Heal(), Heal(), Raptor(1, 0), Raptor(2, 2)]), [Heal(), Fireball(9), Shield(), Heal(),
Fireball(3)]), [Raptor(2, 0), Wyrm(1, 1)], Hero(10, 0, 4, CardDeck([Shield(), Heal(),
Heal(), Raptor(1, 2), Wyrm(1, 3), Shield(), Heal(), Heal(), Raptor(2, 2)]),
[Fireball(1), Shield(), Heal()]), [Wyrm(2, 2), Wyrm(2, 1), Raptor(1, 0)])
```

```
>>> model.end_turn()
```

```
['Fireball', 'Shield', 'Shield']
```

```
>>> model
```

```
HearthModel(Hero(5, 0, 4, CardDeck([Heal(), Raptor(1, 0), Wyrm(1, 0), Shield(),
Heal(), Heal(), Raptor(1, 0), Raptor(2, 2)]), [Heal(), Fireball(10), Shield(),
Heal(), Fireball(4)]), [Wyrm(2, 2)], Hero(10, 10, 5, CardDeck([Heal(), Raptor(1, 2),
Wyrm(1, 3), Shield(), Heal(), Heal(), Raptor(2, 2)]), [Heal(), Heal()]), [Wyrm(3, 1),
Wyrm(2, 1), Raptor(2, 1)])
```

```
>>> model.end_turn()
```

```
['Heal', 'Heal', 'Heal']
```

```
>>> model
```

```
HearthModel(Hero(5, 0, 5, CardDeck([Heal(), Raptor(1, 0), Wyrm(1, 0), Shield(),
Heal(), Heal(), Raptor(1, 0), Raptor(2, 2)]), [Heal(), Fireball(11), Shield(),
Heal(), Fireball(5)]), [Wyrm(3, 0)], Hero(16, 10, 6, CardDeck([Shield(), Heal(),
Heal(), Raptor(2, 2)]), [Raptor(1, 2), Wyrm(1, 3)]), [Wyrm(3, 1), Wyrm(3, 2),
Raptor(3, 2)])
```

```
>>> model.end_turn()
```

```
['Raptor', 'Wyrm', 'Shield', 'Heal']
```

```
>>> model
```

```
HearthModel(Hero(5, 0, 6, CardDeck([Heal(), Raptor(1, 0), Wyrm(1, 0), Shield(),
Heal(), Heal(), Raptor(1, 0), Raptor(2, 2)]), [Heal(), Fireball(12), Shield(),
Heal(), Fireball(6)]), [], Hero(18, 15, 7, CardDeck([Raptor(2, 2)]), [Heal()]),
[Wyrm(3, 1), Wyrm(3, 2), Raptor(3, 2), Raptor(3, 4), Wyrm(2, 4)])
```

```
>>> model.end_turn() # Enemy draws last of their cards
```

```
[]
```

```
>>> model.has_won()
```

```
True
```

```
>>> model
```

```
HearthModel(Hero(5, 0, 6, CardDeck([Heal(), Raptor(1, 0), Wyrm(1, 0), Shield(),
Heal(), Heal(), Raptor(1, 0), Raptor(2, 2)]), [Heal(), Fireball(12), Shield(),
Heal(), Fireball(6)]), [], Hero(18, 15, 8, CardDeck([]), [Heal(), Raptor(2, 2)]),
[Wyrm(3, 1), Wyrm(3, 2), Raptor(3, 2), Raptor(3, 4), Wyrm(2, 4)])
```

Controller

The controller is a single class, `Hearthstone()`, which you must implement according to this section. You will also implement the `play_game` function to construct the controller within this

section. You will also implement the `play_game` function to construct the controller within this section. Unlike with the model, there is no recommended task completion order in this section. It will likely be helpful to work on certain tasks in parallel to implement features. Please refer to the `gameplay/` folder provided with this assignment for examples of intended controller behavior. Note that for all examples, the file initially loaded is `levels/deck1.txt`. Note that examples may appear to have alignment issues if your text editor does not enforce uniform character widths, this has no functional impact on your assignment (you are not implementing the view class).

Task 12 `Hearthstone()`

`Hearthstone()` is the controller class for the overall game. The controller is responsible for creating and maintaining instances of the model and view classes, handling player input, and facilitating communication between the model and view classes.

When loading a game state from a file, it is expected that the first line of the file contains the string representation of a `HearthModel`. Any content after the first line of a file should be ignored when loading a game state from it.

> **Note**
>
> One of the most important skills in programming is the ability to understand and use someone elses code. You will be expected to understand and correctly use the `HearthView` class provided in `display.py`.

`Hearthstone()` should implement the following methods:

- `__init__(self, file: str)`
  Instantiates the controller. Creates view and model instances. The model should be instantiated with the game state specified by the data within the file with the given name. Minions that are not currently in a minion slot should be instantiated with 1 health and 0 shield. You should not handle the case where the file with the specified name does not exist, nor should you handle the case where the file does not contain a valid game state. That is to say, you should not check for an invalid file/game state, and you should not handle any errors that may occur because of one. You should make use of the provided `HearthView` class.
- `__str__(self) -> str`
  Returns a human readable string stating that this is a game of Hearthstone using the current file path.
- `__repr__(self) -> str`
  Returns a string which could be copied and pasted into a REPL to construct a new instance identical to self. **PLEASE READ:** We have been made aware that Gradescope is expecting the `repr` not to contain quotations around the argument

expecting the `repr` not to contain quotations around the argument (i.e. `Hearthstone(file.txt)` instead of `Hearthstone("file.txt")`). While this is not correct, due to the number of submissions made to Gradescope at this point, and the fact it is a minor difference that does not impact the rest of the assignment, we have elected not to change what Gradescope is expecting. Please write your `repr` method to satisfy Gradescope.

- `update_display(self, messages: list[str])`
  Update the display by printing out the current game state. The display should contain (from top to bottom): A banner containing the game name; A depiction of the enemy hero, with (from left to right) their health (HP), shield, number of remaining cards, and energy level; The enemy's minion slots (with minion depiction including health (HP) and shield); The player's minion slots (with minion depiction including health (HP) and shield); The player's current hand; A depiction of the player hero, with (from left to right) their health (HP), shield, number of remaining cards, and energy level; A list of messages to the player. Minions and heros are labelled with the character/number that should be entered to target them. Messages are arranges such that earlier messages in the provided list appear above later ones. You should make use of the provided `HearthView` class to accomplish this.
- `get_command(self) -> str`
  Repeatedly prompts the user until they enter a valid command. Returns the first valid command entered by the user. The possible valid commands are given in Table 2. Whenever the user enters an invalid command, the display should be updated with the `INVALID_COMMAND` message from `support.py` before the user is prompted again. The player's command will be case insensitive, but the returned command should be lower case. Note also that card positions will be entered one-indexed (that is, starting at 1, not 0).
- `get_target_entity(self) -> str`
  Repeatedly prompts the user until they enter a valid entity identifier. A valid entity identifier is one of the following: `PLAYER_SELECT` or `ENEMY_SELECT` from `support.py`, to select the player or enemy hero respectively; an integer between 1 and 5 inclusive, to select the minion in the enemy's minion slot at the respective position; or an integer between 6 and 10 inclusive, to select the minion in the player's minion slot at the position given by the subtracting 5 from the integer. If a minion does not currently exist at a specified position, then the identifier is treated as invalid. If a hero is selected, the identifier should be returned directly. If an enemy's minion is selected, the (zero-indexed) *index* of the minion in the enemy's minion slots should be returned prepended by `ENEMY_SELECT` from `support.py`. If an player's minion is selected, the (zero-indexed) *index* of the minion in the player's minion slots should be returned prepended by `PLAYER_SELECT` from `support.py`. Input should be *case-insensitive*, but the returned identifier should be *upper-case*. Whenever the user enters an invalid identifier, the display

should be updated with the `INVALID_ENTITY` message from `support.py` before the user is prompted again.

- `save_game(self)`
  Writes the string representation of this controllers `HearthModel` instance to `autosave.txt`. If `autosave.txt` does not exist, it should be created. If `autosave.txt` already has content in it, it should be overwritten.
- `load_game(self, file: str)`
  Replaces the current model instance with a new one loaded from the data within the file with the given name. Minions that are not currently in a minion slot should be instantiated with 1 health and 0 shield. You should **not** handle the case where the file with the specified name does not exist, nor should you handle the case where the file does not contain a valid game state. That is to say, you should not check for an invalid file/game state, and you should not handle any errors that may occur because of one.
- `play(self)`
  Conducts a game of Hearthstone from start to finish, following the Gameplay section

Task 13 `play_game(file: str)`

The play game function should be fairly short and do *exactly* two things:

1. Construct a controller instance with the given game file. You *cannot* assume no errors will occur when loading the file. If a `ValueError` or `FileNotFound` error is raised while creating the controller, you should attempt to recreate the controller using `autosave.txt`. You should not handle any errors raised when constructing this backup controller.
2. Play a single game of Hearthstone from beginning to end (using `.play()`)

Task 14 `main()`

The purpose of the main function is to allow you to test your own code. Like the play game function, the main function should be short (A single line) and do exactly one thing: call `play_game` with a file name of your choice.

Assessment and Marking Criteria

This assignment assesses following 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.

There are a total of 100 marks for this assessment item.

Functionality

Your program's functionality will be marked out of a total of 50 marks.

The breakdown of marks for each implementation section is as follows:

- Model: 35 Marks
- Controller: 15 Marks

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 section for partially working functions, or for implementing only a few functions.

> **Warning**
>
> 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 string outputs need to match *exactly* what is expected.

Your program must run in Gradescope, which uses Python 3.12. 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 3.12 interpreter. If it runs in another environment (e.g. Python 3.8 or PyCharm) but not in the Python 3.12 interpreter, you will get zero for the functionality mark.

Code Style

The style of your assignment will be assessed by a tutor. Style will be marked according to the style rubric provided with the assignment. The style mark will be out of 50.

> **Note**

> Style accounts for half the marks available on this assignment

You are expected to follow the PEP-8 style guidelines discussed in lectures. 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 easy to read and follow its logic. This includes using whitespace to highlight blocks of logic, and ensuring all lines are below 80 characters.
  - Descriptive Identifier Names: Variable, constant, and function names clearly describe what they represent in the program's logic. Do not use Hungarian Notation for identifiers. In short, this means do not include the identifier's type in its name, rather make the name meaningful (e.g. employee identifier).
  - Named Constants: Any non-trivial fixed value (literal constant) in the code is represented by a descriptive named constant (identifier).
- 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 function.
  - Variable Scope: Variables should be declared locally in the function in which they are needed. 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).
- 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.
  - Abstraction: Public interfaces of classes are simple and reusable. Enabling modular and reusable components which abstract GUI details.
  - 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 Model-View-Controller design pattern. 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.

- 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 function should have a docstring that summarises its purpose. This includes describing parameters and return values (including type information) so that others can understand how to use the function correctly.
  - Description of Logic: All significant blocks of code should have a comment to explain how the logic works. For a small function, this would usually be the docstring. For long or complex functions, there may be different blocks of code in the function. Each of these should have an in-line comment describing the logic.

Assignment Submission

You must submit your assignment electronically via Gradescope. You **must** use your UQ email address which is based on your student number (e.g. `s4123456@student.uq.edu.au`) as your Gradescope submission account.

When you login to Gradescope you may be presented with a list of courses. Select `CSSE1001`. You will see a list of assignments. Choose `Assignment 2`. You will be prompted to choose a file to upload. Do *not* select the option to submit using a git repository if it is presented to you. The prompt may say that you can upload any files, including zip files. 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 *any* sort of archive file (e.g. zip, rar, 7z, etc.).

Upload an initial version of your assignment *at least* one week before the due date. Do this even if it is just the initial code provided with the assignment. If you are unable access Gradescope, make a post on Edstem *immediately*. Excuses, such as you were not able to login or were unable to upload a file will not be accepted as reasons for granting an extension.

When you upload your assignment it will run a **subset** of the functionality autograder tests on your submission. It will show you the results of these tests. It is your responsibility to ensure that your uploaded assignment file runs and that it passes the tests you expect it to pass.

Late submission of the assignment will result in a deduction of 100% of the total possible mark. A one-hour grace period will be applied to the due time, after which time (16:00) your submission will be considered officially late and will receive a mark of 0. Do not wait until the

last minute to submit your assignment, as the time to upload it may make it late. **Multiple submissions are allowed and encouraged**, so ensure that you have submitted an almost complete version of the assignment *well* before the submission deadline of 15:00. Your latest submission will be marked. Do not submit after the deadline, as this will result in a late penalty of 100% of the maximum possible mark being applied to your submission.

In the event of exceptional personal or medical circumstances that prevent you from handing in the assignment on time, you may submit a request for an extension. See the course profile for details of how to apply for an extension.

Requests for extensions must be made **before** the submission deadline. The application and supporting documentation (e.g. medical certificate) must be submitted via myUQ. You must retain the original documentation for a minimum period of six months to provide as verification, should you be requested to do so.

Plagiarism

This assignment must be your own individual work. By submitting the assignment, you are claiming it is entirely your own work. You may discuss general ideas about the solution approach with other students. Describing details of how you implement a function or sharing part of your code with another student is considered to be **collusion** and will be counted as plagiarism. You **must not** copy fragments of code that you find on the Internet to use in your assignment. You **must not** use any artificial intelligence programs to assist you in writing your assignment.

Please read the section in the course profile about plagiarism. You are encouraged to complete *both* parts A and B of the academic integrity modules *before* starting this assignment. Submitted assignments will be electronically checked for potential cases of plagiarism.