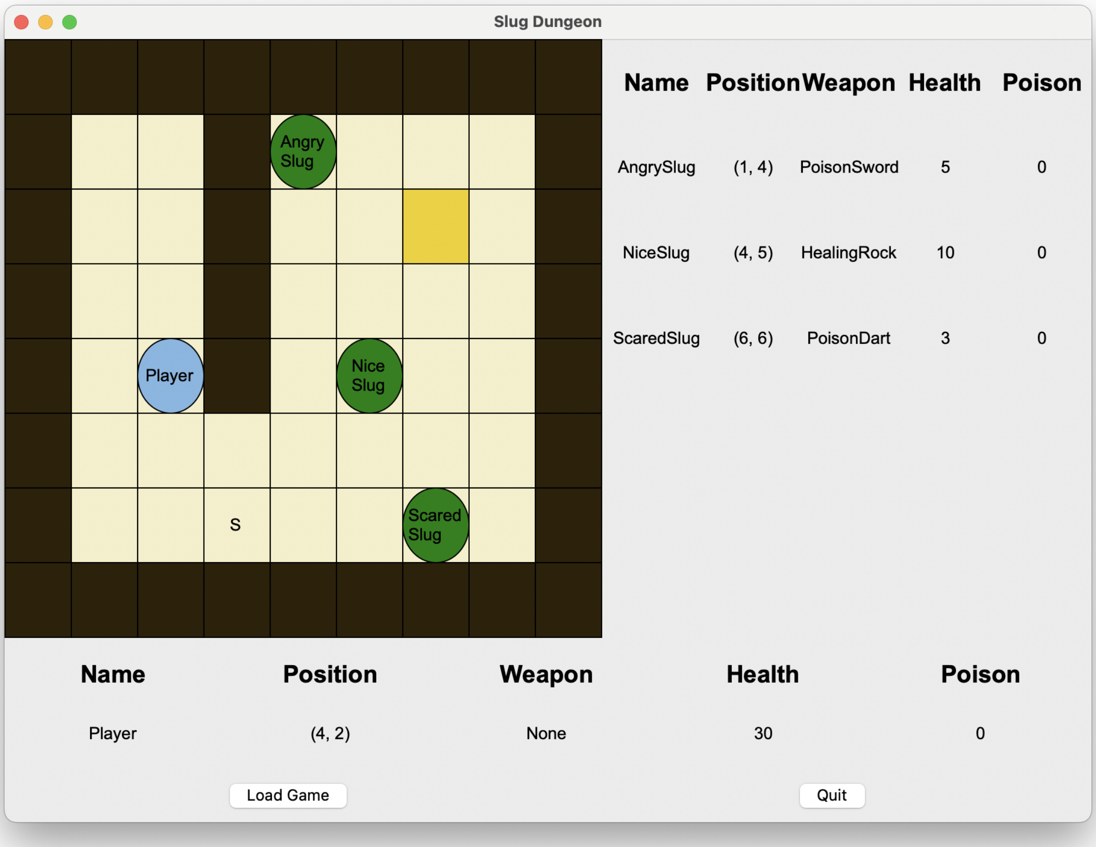
SlugDungeon

Assignment 2
Semester 2, 2024
CSSE1001

Due date: 18 October 2024, 15:00 GMT+10

1 Introduction

In this assignment, you will implement a game in which a player must escape from a dungeon of slugs. Unlike assignment 1, in this assignment you will be using object-oriented programming and following the Apple Model-View-Controller (MVC) design pattern shown in lectures. In addition to creating code for modelling the game, you will be implementing a graphical user interface (GUI). An example of a final completed game is shown in Figure 1.



Name	Position	Weapon	Health	Poison
AngrySlug	(1, 4)	PoisonSword	5	0
NiceSlug	(4, 5)	HealingRock	10	0
ScaredSlug	(6, 6)	PoisonDart	3	0

Name	Position	Weapon	Health	Poison
Player	(4, 2)	None	30	0

Figure 1: Example screenshot from a completed implementation. Note that your display may look slightly different depending on your operating system.

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:

`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 classes and constants to assist you in some parts of your assignment. In addition to these, you are encouraged to create your own constants and helper functions in `a2.py` where possible.

`levels/` This folder contains a small collection of files used to initialize games of *SlugDungeon*. In addition to these, you are encouraged to create your own files to help test your implementation where possible.

3 Gameplay

This section provides a brief overview of gameplay for Assignment 2. Where interactions are not explicitly mentioned in this section, please see Section 4.

3.1 Overview and Terminology

SlugDungeon takes place on a 2D grid called a *board*, where each $< row, col >$ position on the board is represented by a *tile*. In the dungeon, there are also *entities*, which move around the board and potentially attack other entities.
Each game consists of one *player* entity, and a number of *slug* entities. Entities have a health and poison stat, and may optionally have an equipped *weapon*. An entity *dies* when their health is reduced to 0. At the end of each turn, if an entity is *poisoned* (has a poison stat greater than 0) the entity takes damage equal to its poison amount before its poison stat is reduced by 1. Each *turn*, the player makes one move (either moving one square up, down, left, or right or remaining where they are). All entities attack every turn, but slugs can only move every second turn.
Weapons have an *effect* and a range (depending on their type). Weapon's effects can alter the health and poison stats of the entities at positions within their range.

3.2 Player Interaction

The game is played as a series of turns. The user may instruct the player to move or stay where they are via keypresses (see Table 1 for the behaviours associated with each key). If the key pressed is not associated with a valid movement (e.g. it is not one of 'a', 's', 'w', 'd', or 'space', or it would move the player to a blocking tile or to another entity's position) it is ignored and not counted as a turn. Otherwise, if the keypress does result in a valid move (or intentionally leaving the player in place), the following steps should occur:

1. The player moves (or remains where it is) according to Table 1.
2. If the player's new position contains a weapon, the player picks up the weapon and discards any weapon it may have already been holding.
3. The player attacks from its new position.
4. All slugs that die as a result of the player's attack are removed from the game. Each dead slug drops its weapon onto the tile it occupied when it died, replacing any weapon that already existed on that tile.

5. The player's poison is applied and then reduced.
6. Each slug's poison is applied and then reduced.
7. If the slugs are able to move on this turn, all slugs should make their desired move.
8. All slugs attack from their (potentially new) position.
9. If the player has won or lost the game, a messagebox is displayed informing the user of the outcome and asking whether the user would like to play again. If the user chooses to play again, the game file should reload and the user should be able to play again. If they select not to play again, the program should terminate gracefully. Otherwise, the program should continue waiting for more keypresses to begin a new turn.

Key Pressed	**Intended behaviour**
a	Attempt to move player **left** by 1 square.
s	Attempt to move player **down** by 1 square.
d	Attempt to move player **right** by 1 square.
w	Attempt to move player **up** by 1 square.
space	Have player remain where they are (allow them to attack again from their current position).
Anything else	Ignore.

Table 1: Key press behaviours

3.3 Movement

Entities can only move one square at a time (directly up, down, left, or right). Slugs are only allowed to move once for every second turn (where a turn is one player movement). An entity can only move to a target position if they are allowed to move this turn, the target position does not contain a blocking tile or another entity, and the target position is one square away from the entity's original position.

3.4 Attacking

Entities attempt to make an *attack* every turn after they have moved. When the player attacks, it applies its weapon's effect to all slugs within its weapon's range. When a slug attacks, it applies its weapon's effect to the player if the player is within the slug's weapon's range. If an entity is not holding a weapon its attack should have no effect.

3.5 Game End

The game ends when one of the following occurs:

- The user *wins* the game because all slugs are dead and the player is on a goal tile.
- The user *loses* the game because the player dies.

4 Implementation

***NOTE:* 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 a **deduction of up to 100% of your mark.**

Required Classes, Methods, and Functions

You are required to follow the Apple Model-View-Controller design pattern when implementing this assignment. In order to achieve this, you are *required* to implement the classes, methods, and functions described in this section.

The class diagram in Figure 2 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 Sections 4.1, 4.2 and 4.3. Within Figure 2:

- Orange classes are those provided to you in the support file, or imported from TkInter.
- 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 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.
- 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.

The rest of this section describes the required implementation in detail. You should complete the model section, and ensure it operates correctly (via your own testing and the Gradescope tests) before moving on to the view and controller sections. You will not be able to earn marks for the controller section until you have passed all Gradescope tests for the model section.
NOTE: It is possible to receive a passing grade on this assessment by completing only section 4.1, provided the solution works well (according to the Gradescope tests) and is appropriately styled. See section 5.2 for more detail on style requirements.

4.1 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 class (locally and on Gradescope) before moving on to the next class. Functionality marks are awarded for each class (and each method) that *works 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).

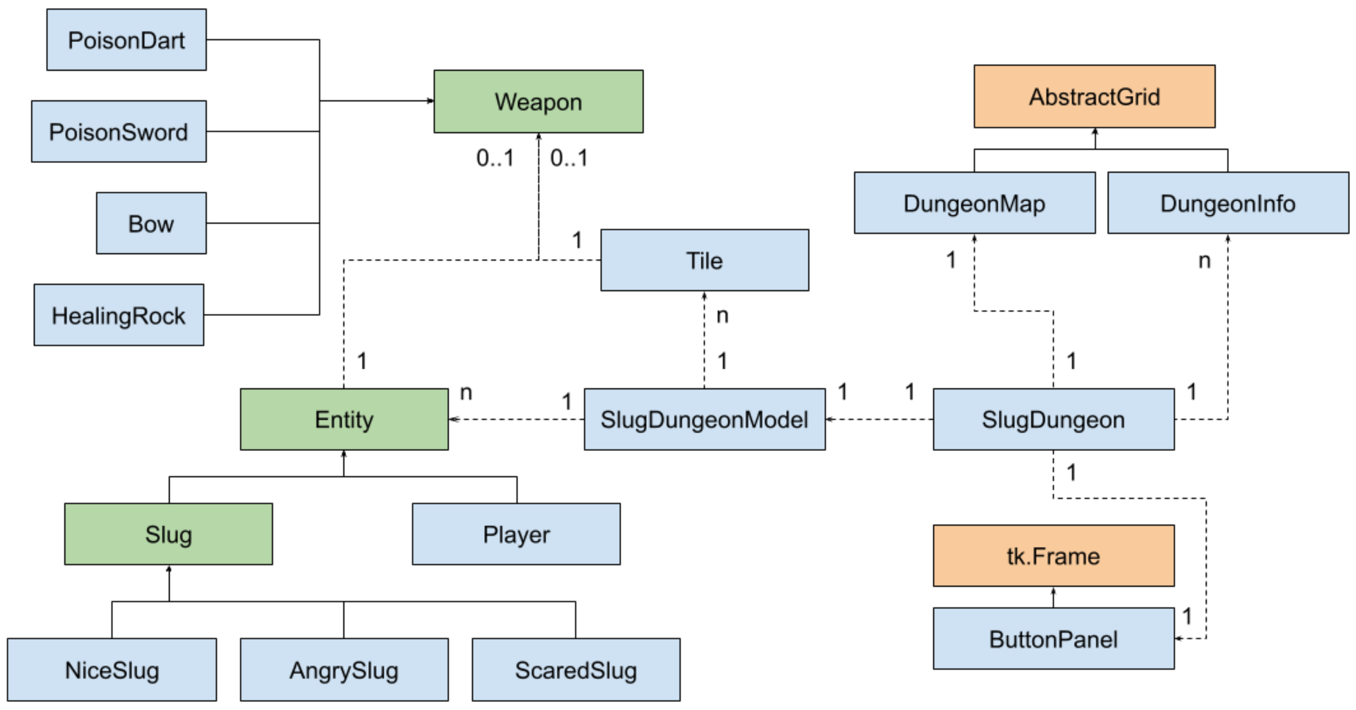


Figure 2: Basic class relationship diagram for the classes in assignment 2.

4.1.1 `Weapon()`

`Weapon` is an abstract class from which all instantiated types of weapon inherit. This class provides default weapon behaviour, which can be inherited or overridden by specific types of weapons. All weapons should have a symbol, name, effect, and range.

- When a weapon is used to attack, it has an *effect* on its target, which may alter the targets health or poison stat. Weapon classes are not responsible for applying their effects to entities, but must provide details about their intended effects. Within this class, a weapon's effect is represented by a dictionary mapping strings to integers, where the string represents the type of effect and the integer represents the strength of the effect. The valid effect types are `"damage"` (which reduces the target's health), `"healing"` (which increases the target's health), and `"poison"` (which increases the targets poison stat).
- A weapon's *range* is an integer representing how far the effects of the weapon reach in all 4 directions. A range of 1 would mean that a weapon's effects would reach entities located one position up, down, left, and right of the weapon.

Abstract weapons have no effects and a range of 0. The `__init__` method does not take any arguments beyond `self`. `Weapon` must implement the following methods:

- `get_name(self) -> str`

  Returns the name of this weapon.

- `get_symbol(self) -> str`

  Returns the symbol of this weapon.

- `get_effect(self) -> dict[str, int]`

  Returns a dictionary representing the weapon's effects.

- `get_targets(self, position: Position) -> list[Position]`

Returns a list of all positions within range for this weapon, given that the weapon is currently at `position`. Note that this method does not need to check whether the target positions exist within a dungeon.

- `__str__(self) -> str`

  Returns the name of this weapon.

- `__repr__(self) -> str`

  Returns a string which could be copied and pasted into a REPL to construct a new instance identical to `self`.

Examples:

```
>>> weapon = Weapon()
>>> weapon.get_name()
'AbstractWeapon'
>>> weapon.get_symbol()
'W'
>>> weapon.get_effect()
{}
>>> weapon.get_targets((1, 1))
[]
>>> str(weapon)
'AbstractWeapon'
>>> weapon
Weapon()
```

4.1.2 `PoisonDart(Weapon)`

`PoisonDart` inherits from `Weapon`. A poison dart has a range of 2 and applies 2 poison to its targets.
Examples:

```
>>> poison_dart = PoisonDart()
>>> poison_dart.get_name()
'PoisonDart'
>>> poison_dart.get_symbol()
'D'
>>> poison_dart.get_effect()
{'poison': 2}
>>> poison_dart.get_targets((1, 1))
[(1, 2), (1, 0), (2, 1), (0, 1), (1, 3), (1, -1), (3, 1), (-1, 1)]
>>> poison_dart.get_targets((8, 8))
[(8, 9), (8, 7), (9, 8), (7, 8), (8, 10), (8, 6), (10, 8), (6, 8)]
>>> str(poison_dart)
'PoisonDart'
>>> poison_dart
PoisonDart()
```

4.1.3 `PoisonSword(Weapon)`

`PoisonSword` inherits from `Weapon`. A poison sword has a range of 1 and both damages its targets by 2 and increases their poison stat by 1.
Examples:

```
>>> poison_sword = PoisonSword()
>>> poison_sword.get_name()
'PoisonSword'
>>> poison_sword.get_symbol()
'S'
>>> poison_sword.get_effect()
{'damage': 2, 'poison': 1}
>>> poison_sword.get_targets((1, 1))
[(1, 2), (1, 0), (2, 1), (0, 1)]
>>> poison_sword.get_targets((0, 0))
[(0, 1), (0, -1), (1, 0), (-1, 0)]
>>> str(poison_sword)
'PoisonSword'
>>> poison_sword
PoisonSword()
```

4.1.4 `HealingRock(Weapon)`

`HealingRock` inherits from `Weapon`. A healing rock has a range of 2 and increases its targets health by 2.
Examples:

```
>>> healing_rock = HealingRock()
>>> healing_rock.get_name()
'HealingRock'
>>> healing_rock.get_symbol()
'H'
>>> healing_rock.get_effect()
{'healing': 2}
>>> healing_rock.get_targets((1, 1))
[(1, 2), (1, 0), (2, 1), (0, 1), (1, 3), (1, -1), (3, 1), (-1, 1)]
>>> str(healing_rock)
'HealingRock'
>>> healing_rock
HealingRock()
```

4.1.5 `Tile()`

`Tile` is a class representing individual tiles (positions) in the dungeon. A tile may or may not be blocking and may or may not have a weapon sitting on it. Entities cannot move onto blocking tiles, but weapon effects can pass through blocking tiles. Each tile has a symbol which represents what type of tile it is. `Tile` must implement the following methods:

- `__init__(self, symbol: str, is_blocking: bool) -> None`

  Constructs a new tile instance with the given `symbol` and `is_blocking` status. Newly instantiated tiles do not have a weapon on them.

- `is_blocking(self) -> bool`

  Returns True if this tile is blocking and False otherwise.

- `get_weapon(self) -> Optional[Weapon]`

  Returns the weapon that's on this tile, or None if there is no weapon on this tile.

- `set_weapon(self, weapon: Weapon) -> None`

  Sets the weapon on this tile to `weapon`, replacing any weapon that may already be there.

- `remove_weapon(self) -> None`

  Removes the tiles's current weapon.

- `__str__(self) -> str`

  Returns the symbol associated with this tile.

- `__repr__(self) -> str`

  Returns a string which could be copied and pasted into a REPL to construct an instance of `Tile` with the same `symbol` and `is_blocking` status as `self`.

Examples:

```
>>> tile = Tile("#", True)
>>> tile.is_blocking()
True
>>> tile.get_weapon()
>>> healing_rock = HealingRock()
>>> tile.set_weapon(healing_rock)
>>> tile.get_weapon()
HealingRock()
>>> tile.get_weapon().get_effect()
{'healing': 2}
>>> tile.remove_weapon()
>>> tile.get_weapon()
>>> str(tile)
'#'
>>> tile
Tile('#', True)
>>> tile = Tile("hello", False)
>>> tile
Tile('hello', False)
>>> tile.is_blocking()
False
```

4.1.6 `create_tile(symbol: str) -> Tile`

`create_tile` is a ***function*** which constructs and returns an appropriate instance of `Tile` based on the `symbol` string. Table 2 describes how the tile should be created based on the value of `symbol`.

`symbol`	**Tile details**
`"#"`	A tile representing a wall, which is blocking and has the symbol `"#"`.
`" "`	A tile representing the floor, which is non-blocking and has the symbol `" "`.
`"G"`	A tile representing the a goal, which is non-blocking and has the symbol `"G"`.
`"D"`, `"S"`, or `"H"`	A floor tile (non-blocking with the symbol `" "`). The weapon on the tile should be an instance of the type of Weapon corresponding to the given symbol.
Any other symbol	A floor tile (non-blocking with the symbol `" "`).

Table 2: Symbol to Tile conversion.

Examples:

```
>>> wall = create_tile("#")
>>> wall
Tile('#', True)
>>> wall.is_blocking()
True
>>> wall.get_weapon()
>>> create_tile(" ")
Tile(' ', False)
>>> create_tile("hello")
Tile(' ', False)
>>> weapon_tile = create_tile("D")
>>> weapon_tile
Tile(' ', False)
>>> weapon_tile.get_weapon()
PoisonDart()
```

4.1.7 `Entity()`

`Entity` is an abstract class from which all instantiated types of entity inherit. This class provides default entity behaviour which can be inherited, overridden, or extended by specific types of entities. All entities should have a health and poison stat, as well as a maximum health. Entities may or may not hold a weapon.
`Entity` must implement the following methods:

- `__init__(self, max_health: int) -> None`

  Constructs a new entity with the given `max_health`. The entity starts with its health stat equal to `max_health`, its poison stat at 0, and no weapon.

- `get_symbol(self) -> str`

  Returns the symbol associated with this entity. The default for an abstract entity is `"E"`.

- `get_name(self) -> str`

  Returns the name associated with this entity. The default for an abstract entity is `"Entity"`.

- `get_health(self) -> int`

  Returns the entity's current health stat.

- `get_poison(self) -> int`

  Returns the entity's current poison stat.

- `get_weapon(self) -> Optional[Weapon]`

  Returns the weapon currently held by this entity, or None if the entity is not holding a weapon.

- `equip(self, weapon: Weapon) -> None`

  Sets the entity's weapon to `weapon`, replacing any weapon the entity is already holding.

- `get_weapon_targets(self, position: Position) -> list[Position]`

  Returns the positions the entity can attack with its weapon from the given position. If the entity doesn't have a weapon this method should return an empty list.

- `get_weapon_effect(self) -> dict[str, int]`

  Returns the effect of the entity's weapon. If the entity doesn't have a weapon this method should return an empty dictionary.

- `apply_effects(self, effects: dict[str, int]) -> None`

  Applies each of the effects described in the `effects` dictionary to the entity. This involves increasing the entity's health by any `"healing"` amount, reducing the entity's health by any `"damage"` amount, and increasing the entity's poison stat by any `"poison"` amount. Note that this method should not handle reducing the entity's health by the poison stat. The entity's health should be capped between 0 and its `max_health` (inclusive).

- `apply_poison(self) -> None`

  Reduces the entity's health by its poison amount (capping the health at 0), then reduces the entity's poison stat by 1 if it is above 0. If the player's poison stat is 0, this method should do nothing.

- `is_alive(self) -> bool`

  Returns True if the entity is still alive (its health is greater than 0) and False otherwise.

- `__str__(self) -> str`

  Returns the name of this entity.

- `__repr__(self) -> str`

  Returns a string which could be copied and pasted into a REPL to construct a new instance which would look identical to the *original state* of `self`.

Examples:

```
>>> entity = Entity(10)
>>> entity.get_symbol()
'E'
>>> entity.get_name()
'Entity'
>>> entity.get_health()
10
>>> entity.get_poison()
0
>>> entity.get_weapon()
>>> entity.get_weapon_targets((1, 1))
[]
>>> entity.get_weapon_effect()
{}
>>> entity.equip(PoisonSword())
>>> entity.get_weapon()
PoisonSword()
>>> entity.get_weapon_targets((0, 0))
[(0, 1), (0, -1), (1, 0), (-1, 0)]
>>> entity.get_weapon_effect()
{'damage': 2, 'poison': 1}
>>> entity.apply_effects({'poison': 4, 'damage': 3})
>>> entity.get_health()
7
>>> entity.get_poison()
4
>>> entity.apply_poison()
>>> entity.get_health()
3
>>> entity.get_poison()
3
>>> entity.apply_effects({'healing': 20})
>>> entity.get_health()
10
>>> str(entity)
'Entity'
>>> entity
Entity(10)
```

4.1.8 `Player(Entity)`

`Player` inherits from `Entity`. `Player` should provide almost identical behaviour to the abstract `Entity` class, but with a name of `"Player"` and a symbol of `"P"`.

Examples:

```
>>> player = Player(20)
>>> str(player)
'Player'
>>> player
Player(20)
>>> player.equip(PoisonDart())
>>> player.get_weapon_effect()
{'poison': 2}
>>> player.apply_effects({'damage': 25})
>>> player.get_health()
0
>>> player
Player(20)
```

4.1.9 `Slug(Entity)`

`Slug` is an abstract class which inherits from `Entity` and from which all instantiated types of slug inherit. Slugs can only move (change position) every second turn, starting on the first turn of the game. That is, slugs move on the first turn, the third turn, the fifth turn, and so on. Even though slugs only move every second turn, they attack every turn. Slugs provide functionality to decide how they should move. In addition to regular `Entity` behaviour, the `Slug` class must implement the following methods:

- `choose_move(self, candidates: list[Position], current_position: Position, player_position: Position) -> Position`

  Chooses and returns the position the slug should move to (from the positions in `candidates` or the `current_position`), assuming the slug can move. In the abstract `Slug` class this method should raise a `NotImplementedError` with the message `"Slug subclasses must implement` a `choose_move method."`. Subclasses of `Slug` must override this method with a valid implementation. Note that this method must not mutate its arguments.

- `can_move(self) -> bool`

  Returns True if the slug can move on this turn and False otherwise.

- `end_turn(self) -> None`

  Registers that the slug has completed another turn (toggles whether the slug can move).

Examples:

```
>>> slug = Slug(5)
>>> slug.equip(HealingRock())
>>> slug.get_weapon_effect()
{'healing': 2}
>>> slug.can_move()
True
>>> slug.end_turn()
>>> slug.can_move()
False
>>> slug.can_move()
False
>>> slug.end_turn()
>>> slug.can_move()
True
>>> slug.choose_move([(0, 0), (1, 1)], (1, 0), (2, 4))
...
NotImplementedError: Slug subclasses must implement a choose_move method.
>>> str(slug)
'Slug'
>>> slug
Slug(5)
```

4.1.10 `NiceSlug(Slug)`

`NiceSlug` inherits from `Slug`, and represents a nice but lazy slug which stays where it is whenever it can move. All `NiceSlug` instances should hold a `HealingRock` weapon and have a `max_health` of 10. `NiceSlug` should not take any arguments to `__init__` (beyond `self`). `NiceSlug`, as a `Slug` subclass, must override the `choose_move` function with appropriate behaviour.

Examples:

```
>>> nice_slug = NiceSlug()
>>> nice_slug
NiceSlug()
>>> str(nice_slug)
'NiceSlug'
>>> nice_slug.get_health()
10
>>> nice_slug.get_weapon()
HealingRock()
>>> nice_slug.get_weapon_effect()
{'healing': 2}
```

```
>>> nice_slug.choose_move([(0, 1), (1, 0), (1, 2), (2, 1)], (1, 1), (2, 4))
(1, 1)
```

4.1.11 `AngrySlug(Slug)`

`AngrySlug` inherits from `Slug`, and represents an angry slug that tries to move towards the player whenever it can move. All `AngrySlug` instances should hold a `PoisonSword` weapon and have a `max_health` of 5. `AngrySlug` should not take any arguments to `__init__` (beyond `self`). `AngrySlug`, as a `Slug` subclass, must override the `choose_move` function with appropriate behaviour.

`AngrySlug` instances always choose the position (out of the positions in `candidates` or the `current_position`) which is closest (by Euclidean distance) to the `player_position`. If multiple positions tie as closest to the player, the position with the lowest value (according to the `<` operator on tuples) should be chosen.

Examples:

```
>>> angry_slug = AngrySlug()
>>> angry_slug
AngrySlug()
>>> str(angry_slug)
'AngrySlug'
>>> angry_slug.get_health()
5
>>> angry_slug.get_weapon()
PoisonSword()
>>> angry_slug.get_weapon_effect()
{'damage': 2, 'poison': 1}
>>> angry_slug.choose_move([(0, 1), (1, 0), (1, 2), (2, 1)], (1, 1), (2, 4))
(1, 2)
```

4.1.12 `ScaredSlug(Slug)`

`ScaredSlug` inherits from `Slug`, and represents a fearful slug that tries to move away from the player whenever it can move. All `ScaredSlug` instances should hold a `PoisonDart` weapon and have a `max_health` of 3. `ScaredSlug` should not take any arguments to `__init__` (beyond `self`). `ScaredSlug`, as a `Slug` subclass, must override the `choose_move` function with appropriate behaviour.

`ScaredSlug` instances always choose the position (out of the positions in `candidates` or the `current_position`) which is furthest (by Euclidean distance) from the `player_position`. If multiple positions tie as furthest from player, the position with the highest value (according to the `>` operator on tuples) should be chosen.

Examples:

```
>>> scared_slug = ScaredSlug()
>>> scared_slug
ScaredSlug()
>>> str(scared_slug)
'ScaredSlug'
>>> scared_slug.get_health()
3
>>> scared_slug.get_weapon()
```

```
PoisonDart()
>>> scared_slug.get_weapon_effect()
{'poison': 2}
>>> scared_slug.choose_move([(0, 1), (1, 0), (1, 2), (2, 1)], (1, 1), (2, 4))
(1, 0)
```

4.1.13 `SlugDungeonModel()`

`SlugDungeonModel` models the logical state of a game of *SlugDungeon*. `SlugDungeonModel` must implement the following methods:

- `__init__(self, tiles: list[list[Tile]], slugs: dict[Position, Slug], player: Player, player_position: Position) -> None`

  Initializes a new `SlugDungeonModel` from the provided information.

  - `tiles` is a list of lists of `Tile` *instances* where each internal list represents a row (from topmost to bottommost) of tiles (from leftmost to rightmost).
  - `slugs` is a dictionary mapping slug positions to the `Slug` *instances* at those positions. You should never change the order of this dictionary from the initial order. Operations that apply to all slugs should be applied in the order in which the slugs appear in this dictionary.
  - `player` is the `Player` instance and `player_position` is the player's starting position.

- `get_tiles(self) -> list[list[Tile]]`

  Returns the tiles for this game in the same format provided to `__init__`.

- `get_slugs(self) -> dict[Position, Slug]`

  Returns a dictionary mapping slug positions to the `Slug` *instances* at those positions.

- `get_player(self) -> Player`

  Returns the player instance.

- `get_player_position(self) -> Position`

  Returns the player's current position.

- `get_tile(self, position: Position) -> Tile`

  Returns the tile at the given `position`.

- `get_dimensions(self) -> tuple[int, int]`

  Returns the dimensions of the board as (#rows, #columns).

- `get_valid_slug_positions(self, slug: Slug) -> list[Position]`

  Returns a list of valid positions that `slug` can move to from its current position in the next move. If the slug cannot move on this turn (i.e. it was able to move last turn) this method should return an empty list. Otherwise, a slug can move one position up, down, left, or right of its current position if those positions exist on the board and do not contain a blocking tile or another entity. A pre-condition to this method is that `slug` must be one of the slugs in the game, and must still be alive.

- `perform_attack(self, entity: Entity, position: Position) -> None`

Allows the given `entity` to make an attack from the given `position`. If `entity` isn't holding a weapon this method should do nothing. Otherwise, the entity's weapon's effects should be applied to each target entity whose position is within the weapon's target positions. Note that the player only attacks slugs, and slugs only attack the player.

- `end_turn(self) -> None`

  Handles the steps that should occur after the player has moved. This involves:

  - Applying the player's poison (including decremementing their poison stat).
  - Applying each slug's poison (including decrementing their poison stat). If a slug is no longer alive after their poison is applied they should be removed from the game, and should no longer appear in the dictionary when calling `get_slugs`. When a slug is removed from the game, the tile at their final position should have its weapon set to the slug's weapon.
  - If the remaining slugs are able to move this turn, they should choose a position out of their valid positions (or their current position) and move there. Note that, while this method will be called directly after a player makes a move, the slugs are too slow to process that move before making their decision. As such, they should make their choice of move based on the player's *previous* position, not their current position. If this is the first move of the game, the player's previous position is considered to be their current position.
  - All (remaining) slugs should perform an attack from their position, and then register that they have completed another turn.

  Note that you may find it beneficial to work on this method in parallel with `handle_player_move`.

- `handle_player_move(self, position_delta: Position) -> None`

  Moves the player by `position_delta` (by adding `position_delta` to the player's position element-wise) if the move is valid. A move is valid if it results in a position that is within bounds for the board which does not contain a blocking tile or another entity. If the move is valid, the player's position should update before the following steps occur:

  - If the tile at the new position contains a weapon, the player should equip the weapon and the weapon should be removed from the tile.
  - The player should perform an attack from its new position.
  - The `end_turn` method should be called to run all other steps required for this turn.

- `has_won(self) -> bool`

  Returns True if the player has won the game (killed all slugs and reached a goal tile), and False otherwise.

- `has_lost(self) -> bool`

  Returns True if the player has lost the game (is no longer alive), and False otherwise.

Examples:

```
>>> tiles = [
[create_tile("#"), create_tile("#"), create_tile("#"), create_tile("#")],
[create_tile("#"), create_tile(" "), create_tile(" "), create_tile("#")],
[create_tile("#"), create_tile(" "), create_tile(" "), create_tile("#")],
[create_tile("#"), create_tile("S"), create_tile("G"), create_tile("#")],
[create_tile("#"), create_tile("#"), create_tile("#"), create_tile("#")]
]
```

```
>>> slugs = {(1, 1): AngrySlug()}
>>> player = Player(20)
>>> model = SlugDungeonModel(tiles, slugs, player, (2, 1))
>>> model.get_tiles()
[[Tile('#', True), Tile('#', True), Tile('#', True), Tile('#', True)],
[Tile('#', True), Tile(' ', False), Tile(' ', False), Tile('#', True)],
[Tile('#', True), Tile(' ', False), Tile(' ', False), Tile('#', True)],
[Tile('#', True), Tile(' ', False), Tile('G', False), Tile('#', True)],
[Tile('#', True), Tile('#', True), Tile('#', True), Tile('#', True)]]
>>> model.get_slugs()
{(1, 1): AngrySlug()}
>>> model.get_player()
Player(20)
>>> model.get_player_position()
(2, 1)
>>> model.get_tile((0, 0))
Tile('#', True)
>>> model.get_tile((2, 1))
Tile(' ', False)
>>> model.get_dimensions()
(5, 4)
>>> angry_slug = model.get_slugs().get((1, 1))
>>> angry_slug.get_health()
5
>>> angry_slug.get_weapon_effect()
{'damage': 2, 'poison': 1}
>>> player.get_health()
20
>>> angry_slug.get_poison()
0
>>> player.get_poison()
0
>>> player.get_effect()
{}
>>> model.get_valid_slug_positions(angry_slug)
[(1, 1), (1, 2)]
>>> model.perform_attack(angry_slug, (1, 1))
>>> player.get_health()
18
>>> player.get_poison()
1
>>> model.perform_attack(player, model.get_player_position())
>>> angry_slug.get_health()
5
>>> angry_slug.get_poison()
0
>>> model.handle_player_move((1, 0))
>>> player.get_health()
15
>>> player.get_poison()
1
>>> player.get_weapon() # Player picked up a weapon when it moved
```

```
PoisonSword()
>>> player.get_weapon_effect()
{'damage': 2, 'poison': 1}
>>> player.get_weapon_targets(model.get_player_position())
[(3, 2), (3, 0), (4, 1), (2, 1)] # The slugs original position was not in the targets
>>> angry_slug.get_health() # So its health and poison remain unchanged in this attack
5
>>> angry_slug.get_poison()
0
>>> model.get_slugs() # Even though the slug moved to a target position after the attack
{(2, 1): AngrySlug()}
>>> model.get_player_position()
(3, 1)
>>> model.has_won()
False
>>> model.has_lost()
False
```

4.1.14 `load_level(filename: str) -> SlugDungeonModel`

Implement a ***function*** that reads the file at the given filename and initializes and returns the corresponding `SlugDungeonModel` instance. When creating the slugs dictionary to pass to the `SlugDungeonModel` initializer, slugs appearing in earlier rows (lines of the file) should appear first. Within one row, slugs appearing in earlier columns (earlier in the line) should appear first.

The first line of the file provides an integer representing the player's `max_health`, and the rest of the lines describe the rest of the game state. See the `levels/` folder included in `a2.zip` for examples of the file format. Note that these are not the only levels that will be used to test your code.

4.2 View

The following are the classes and methods you are *required* to implement to complete the view component of this assignment. As opposed to section 4.1, where you would work through the required classes and methods in order, GUI development tends to require that you work on various interacting classes in parallel. Rather than working on each class in the order listed, you may find it beneficial to work on one *feature* at a time and test it thoroughly before moving on. It is likely that you will also need to implement components from the controller class (`SlugDungeon`) in order to develop each feature. Each feature may require updates / extensions to the `SlugDungeon` and view classes. You should implement `play_game` first, as Gradescope calls `play_game` in order to test your code, so you cannot earn marks for the View or Controller sections until you have implemented this function (see section 4.3 for details).

4.2.1 `DungeonMap(AbstractGrid)`

`DungeonMap` inherits from the `AbstractGrid` class provided in `support.py`. `DungeonMap` is a view component that displays the dungeon (tiles, entities, and tile weapons). Tiles are represented by coloured rectangles, and entities are displayed by coloured ovals drawn on top of these tiles, annotated with the entity's name. Tile weapons are represented by annotating the weapon's symbol on top of the tile. Figure 3 shows an example of a completed `DungeonMap`. `DungeonMap`'s `__init__` method should take the same arguments as the `__init__` method for `AbstractGrid`. `DungeonMap` should implement the following method:

- `redraw(self, tiles: list[list[Tile]], player_position: Position, slugs: dict[Position, Slug]) -> None:`

Clears the dungeon map, then redraws it according to the provided information. Note that you must draw on the `DungeonMap` instance it*self* (not directly onto `master` or any other widget).

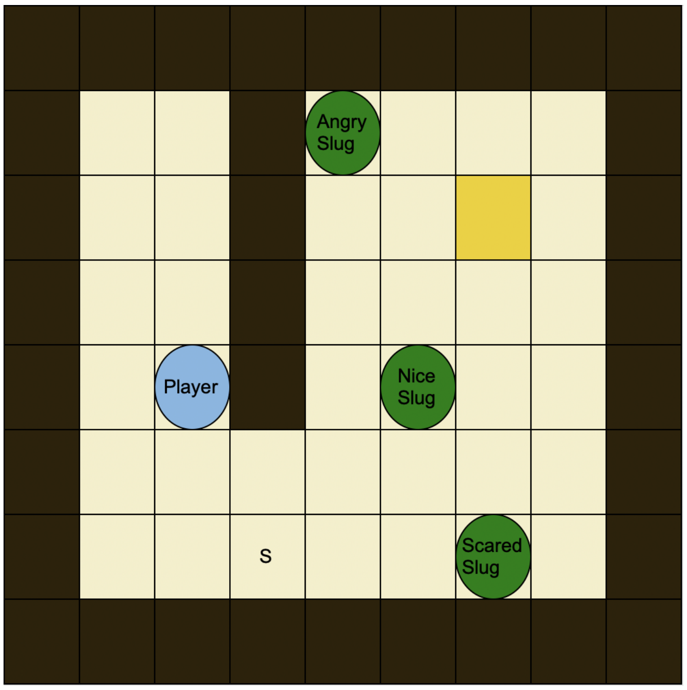


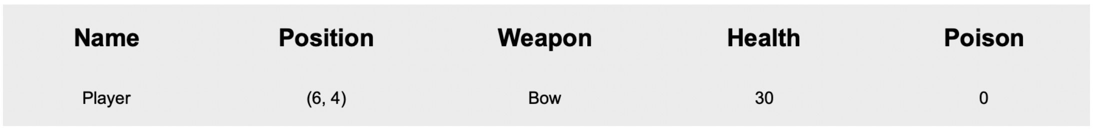
Figure 3: Example of a completed DungeonMap partway through a game.

4.2.2 `DungeonInfo(AbstractGrid)`

`DungeonInfo` inherits from the `AbstractGrid` class provided in `support.py`. `DungeonInfo` is a view component that displays information about a set of entities. The first row of any `DungeonInfo` instance should display headings for Name, Position, Weapon, Health, and Poison (respectively). Subsequent rows display the corresponding information about entities. Figure 4 shows an example of a completed `DungeonInfo`. `DungeonInfo`'s `__init__` method should take the same arguments as the `__init__` method for `AbstractGrid`. `DungeonInfo` should implement the following method:

- `redraw(self, entities: dict[Position, Entity]) -> None`

  Clears and redraws the info with the given entities. Note that you must draw on the `DungeonInfo` instance it*self* (not directly onto `master` or any other widget).



Name	**Position**	**Weapon**	**Health**	**Poison**
Player	(6, 4)	Bow	30	0

Figure 4: A completed DungeonInfo displaying stats for the player partway through a game.

4.2.3 `ButtonPanel(tk.Frame)`

`ButtonPanel` inherits from `tk.Frame` and contains two buttons; one for loading a new game, and one for quiting the game. Figure 5 shows how this widget should look. `ButtonPanel` does not need to be redrawn once it has been created, and should only need to implement an initializer method according to the following definition:

- `__init__(self, root: tk.Tk, on_load: Callable, on_quit: Callable) -> None`

Constructs a new `ButtonPanel` instance, where the leftmost button has the text `"Load Game"` and uses `on_load` as its command, and the rightmost button has the text `"Quit"` and uses `on_quit` as its command.

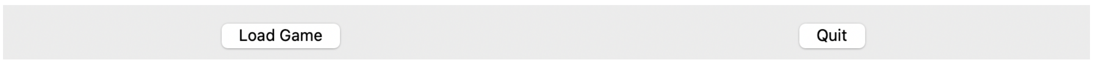


Figure 5: An example of the `ButtonPanel` display.

4.3 Controller

The controller is a single class, `SlugDungeon`. As with the view section, you may find it beneficial to work on one *feature* at a time, instead of working through the required classes and functions in order. You should work on these features in tandem with features from the View section.

4.3.1 `SlugDungeon()`

`SlugDungeon` is the controller class for the overall game. The controller is responsible for creating and maintaining instances of the model and view classes, handling events, and facilitating communication between the model and view classes. `SlugDungeon` should implement the following methods:

- `__init__(self, root: tk.Tk, filename: str) -> None`

  Instantiates the model (based on the data in the file with the given name) and view classes, and redraws the display to show the initial game state. This method should also handle binding any relevant events to handlers. You may assume that no IO errors will occur when loading from `game_file`. The view instances you must make include:

  – A `DungeonMap` instance, which should be 500 pixels wide and 500 pixels tall.
  – A `DungeonInfo` instance with 7 rows to display the slug information. This should be 400 pixels wide and 500 pixels tall. You can assume that the game will not contain more than 6 slugs.
  – A `DungeonInfo` instance to display the player information. This should be 900 pixels wide and 100 pixels tall.
  – A `ButtonPanel` instance, which should fill all the horizontal space available, and have a vertical internal padding of 10 pixels above and below.

- `redraw(self) -> None`

  Redraws the view based on the current state of the model.

- `handle_key_press(self, event: tk.Event) -> None`

  Handles a single keypress event from the user. If the key pressed is not a valid movement (e.g. 'a', 's', 'w', 'd', or 'space') this method should do nothing. Otherwise, this method should attempt to handle a player move and then redraw the view. If the game has been won or lost after the move is made, the user should be informed via an appropriate messagebox, which should also prompt for whether they would like to play again (see `messagebox.askyesno`). If the user opts to play again, the game should return to its original state and the player should be able to play again. Otherwise, the program should terminate gracefully. Note that in order to get the view classes to update before the messagebox appears, you may need to use the `update_idletasks` method on the `tk.Tk` instance.

- `load_level(self) -> None`

Prompts the user to select a file to load (see `filedialog.askopenfilename`), and then replaces the game state with the game from the chosen file. The view should update to display this new game state. Your code will not be tested with invalid files.

4.4 `play_game(root: tk.Tk, file_path: str) -> None`

The `play_game` function should be fairly short and do *exactly* two things:

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

Note that the tests will call this function to test your code, rather than main.

4.5 `main() -> None`

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 fairly short and do *exactly* two things:

1. Construct the root `tk.Tk` instance.
2. Call the `play_game` function passing in the newly created root `tk.Tk` instance, and the path to any map file you like (e.g. `'levels/level1.txt'`).

5 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.

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

5.1 Functionality

Your program's functionality will be marked out of a total of 50 marks. The model is worth 25 marks, and the view and controller together are worth 25 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.

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.

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

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.

5.2 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 that style accounts for half the marks availible on this assignment.

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.
  - 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.

5.3 Assignment Submission

You must submit your assignment electronically via Gradescope (`https://gradescope.com/`). 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. 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 to access Gradescope, post on Edstem or attend a help center or practical session *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 submissions of the assignment will **not** be marked. 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. Submitting after the deadline incurs late penalties. Ensure that you submit the correct version of your assignment.

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 my.UQ. You must retain the original documentation for a minimum period of six months to provide as verification, should you be requested to do so.

5.4 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 **may not** copy fragments of code that you find on the Internet to use in 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.