Maze Runner

Assignment 2
Semester 1, 2022
CSSE1001/CSSE7030

Due date: 6th May 2022 16:00 GMT+10

1 Introduction

Maze Runner is a single-player game in which a player must make their way through a series of mazes. The player must complete each maze before running out of health points (HP), or becoming too hungry or thirsty to continue. To assist in their journey, items can be collected throughout the maze, and applied when needed.

In Assignment 2, you will create an extensible object-oriented implementation of *Maze Runner* which employs a text-based interface.

```
########
       #
###### #
#LL MP #
#L######
#LLL   #
###### #
---------------
Inventory
Coin: 3
Potion: 1
Honey: 1
Apple: 1
---------------
HP: 85
hunger: 3
thirst: 3

Enter a move: a
########
       #
###### #
#LL P  #
#L######
#LLL   #
###### #
---------------
Inventory
Coin: 3
Potion: 2
Honey: 1
Apple: 1
---------------
HP: 84
hunger: 3
thirst: 3

Enter a move:
```

Figure 1: A snippet from a functional game of Maze Runner. Each character in the maze occupies a tile; # are walls, P is the player, whitespaces are empty tiles, and other characters represent items.

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

2 Getting Started

Download `a2.zip` from Blackboard — this archive contains the necessary files to start this assignment.

Once extracted, the `a2.zip` archive will provide the following files/directories:

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

`a2_support.py`
: Provided classes implementing functionality to render the text interface.

`constants.py`
: Constants you *may* find helpful when writing your assignment.

`games`
: A folder containing several text files of playable games.

`game_examples`
: A folder containing example output from running the completed assignment.

3 Gameplay

Gameplay occurs according to the following steps when the program is run:

1. The user is prompted for a game file with the prompt 'Enter game file: '. The filename entered by the user must be a valid file from the file location of `a2.py`. For example, `games/game1.txt`. You do not need to handle the case of an invalid file path being entered.

2. The first map in the level file is displayed to the user. The user is prompted for their move with the prompt 'Enter a move: '.

3. At the move prompt the user can enter either 'w', 'a', 's', or 'd' to move up, left, down, or right respectively, or they may enter 'i *`item_name`*' to apply an item with the given *`item_name`*. From here, one of 3 cases may occur:

   (a) If the user has entered either 'w', 'a', 's', or 'd', and the player is allowed to move to the tile at the resulting position (i.e. the tile is non-blocking), then the player is moved to that tile. If an item exists on this tile, the item is removed from the map and added to the player's inventory. Once the player has collected all coin items from the map, the door is unlocked and the player can complete the level by walking through it.

   (b) If the user has entered 'i *`item_name`*', and the player's inventory contains at least one instance of the item with the given item name, then the item is applied to the player and one instance of the item is removed from the inventory. If no stock was available for the requested item, the user should be informed of this with the `ITEM_UNAVAILABLE_MESSAGE` (see `constants.py`).

   (c) If the player has entered anything else, they are reprompted; see Fig. 2.

4. After each valid move (i.e. moving into a non-blocking tile) the player's HP decreases by 1. After every 5 valid moves the player's hunger and thirst increase by 1. If the player's HP reaches 0, or either hunger or thirst reach their respective maximums, then the player loses, and is informed of their result with the text 'You lose :('. The program then terminates.

5. If the player moves through a door, one of two things will happen:

(a) If another map exists in this game, the player is moved to the beginning of the next map, the player's stats remain as they were (e.g. no change to health, hunger, or thirst based on this move), and the new map is displayed to the user. The user is prompted for their move, and the gameplay continues from step 3 again.

(b) If all maps have been completed, the user is informed that they have won the level with the text 'Congratulations! You have finished all levels and won the game!'. The game then terminates.

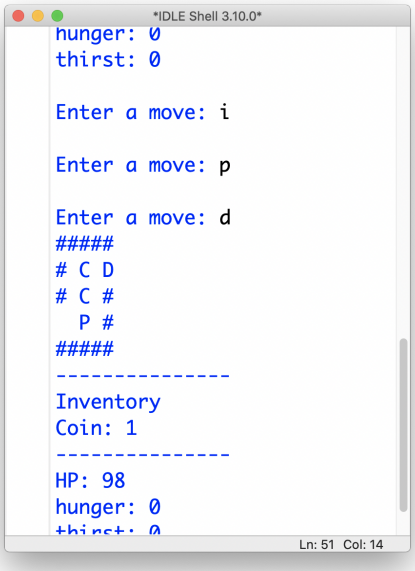

Figure 2: Example behaviour when user input is invalid.

3.1 Coordinate system

The coordinate system used for the maze is depicted in Figure 3 for an example 3×3 grid. Positions are represented as a tuple of row and column number. The top left corner is at position $(0, 0)$, and the bottom right corner of this example map is at position $(2, 2)$. Note that this is just an example; you must not assume that you will always have a 3×3 map.

(0, 0)	(0, 1)	(0, 2)
(1, 0)	(1, 1)	(1, 2)
(2, 0)	(2, 1)	(2, 2)

Figure 3: Coordinate system for example 3×3 grid.

4 Class overview and relationships

The design of the implentation follows the Apple Model-View-Controller (MVC) design pattern; see here for further details. You are *required* to implement a number of classes, which provide an extensible Apple MVC design. The class diagram in Figure 4 provides an overview of *all* of these classes, and the basic relationships between them. The details of these classes and their methods are described in depth in Section 5.

- Hollow-arrowheads indicate *inheritance* (i.e. the "is-a" relationship).
- Dotted arrows indicates *composition* (i.e. the "has-a" relationship). An arrow marked with 1-1 denotes

that each instance of the class at the base of the arrow contains exactly one instance of the class at the head of the arrow. An arrow marked with 1-n denotes that each instance of the class at the base of the arrow may contain many instances of the class at the head of the arrow. E.g. a `Model` instance may contain many different `Level` instances, but only one `Player` instance.

- Green classes are *abstract* classes. 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.
- The two classes surrounded by an orange box are provided to you in `a2_support.py`.

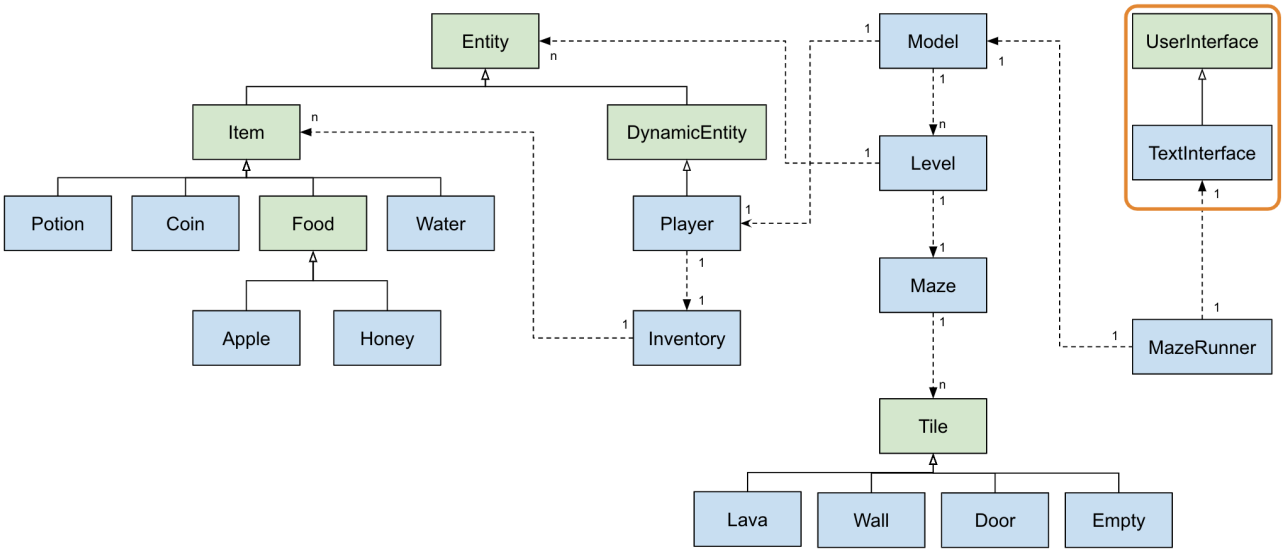

Figure 4: Basic class relationship diagram for the classes which need to be implemented for this assignment.

Note that all view classes have been provided to you. You are required to implement a number of modelling classes, and a single controller class which unites these modelling classes with the existing `TextInterface` view class.

5 Implementation

5.1 Model classes

This section outlines the model classes you must implement, along with their required methods. The classes are laid out in the order in which it is recommended you attempt them (with some exceptions that are discussed in their relevant sections). It is ***strongly recommended*** that you ensure a class is working via testing the implementation yourself before moving on to another class (particularly if that class is a superclass of the next class you will attempt).

Tile *(abstract class)*

Represents the floor for a (row, column) position.

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

Returns True iff the tile is blocking. A tile is blocking if a player would not be allowed to move onto the position it occupies. By default, tile's are not blocking.

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

Returns the damage done to a player if they step on this tile. For instance, if a tile has a damage of 3, the player's HP would be reduced by 3 if they step onto the tile. By default, tile's should do no damage to a player.

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

Returns the ID of the tile. For *non-abstract* subclasses, the ID should be a single character representing the type of Tile it is. See `constants.py` for the ID value of all tiles and entities.

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

Returns the string representation for this Tile.

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

Returns the text that would be required to create a new instance of this class.

Examples

```
>>> tile = Tile()
>>> tile.is_blocking()
False
>>> tile.damage()
0
>>> tile.get_id()
'AT'
>>> str(tile)
'AT'
>>> tile
Tile()
```

Wall *(class)*

Inherits from Tile
Wall is a type of Tile that *is blocking*.

Examples

```
>>> wall = Wall()
>>> wall.is_blocking()
True
>>> wall.get_id()
'#'
>>> str(wall)
'#'
>>> wall
Wall()
```

Empty *(class)*

Inherits from Tile
Empty is a type of Tile that does not contain anything special. A player can move freely over empty tiles without taking any damage. Note that the ID for an empty tile is a single space (not an empty string).

Examples

```
>>> empty = Empty()
>>> empty.is_blocking()
False
>>> empty.get_id()
' '
>>> str(empty)
' '
```

```
>>> empty
Empty()
```

Lava *(class)*

Inherits from Tile
Lava is a type of Tile that is not blocking, but does a damage of 5 to the player's HP when stepped on.

Note: This damage is in *addition* to any other damage sustained. For example, if the player's HP is also reduced by 1 for making a successful move, then the total reduction to the player's HP will be 6. However, the application of the damage for each move should not be handled within this class.

Examples

```
>>> lava = Lava()
>>> lava.is_blocking()
False
>>> lava.get_id()
'L'
>>> lava.damage()
5
>>> str(lava)
'L'
>>> lava
Lava()
```

Door *(class)*

Inherits from Tile
Door is a type of Tile that starts as locked (blocking). Once the player has collected all coins in a given maze, the door is 'unlocked' (becomes non-blocking and has its ID change to that of an empty tile), and the player can move through the square containing the unlocked door to complete the level. In order to facilitate this functionality in later classes, the Door class must provide a method through which to 'unlock' a door.

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

Unlocks the door.

Examples

```
>>> door = Door()
>>> door.is_blocking()
True
>>> door.get_id()
'D'
>>> str(door)
'D'
>>> door
Door()
>>> door.unlock()
>>> door.is_blocking()
False
>>> door.get_id()
' '
>>> str(door)
' '
>>> door
```

```
Door()
```

Entity *(abstract class)*

Provides base functionality for all entities in the game.

`__init__(self, position: tuple[int, int]) -> None` *(method)*

Sets up this entity at the given (row, column) position.

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

Returns this entities (row, column) position.

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

Returns the name of the class to which this entity belongs.

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

Returns the ID of this entity. For all *non-abstract* subclasses, this should be a single character representing the type of the entity.

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

Returns the string representation for this entity (the ID).

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

Returns the text that would be required to make a new instance of this class that looks identical (where possible) to `self`.

Note: you are only required to replicate information that can be set via arguments to the `__init__` method for the relevant class. For instance, in the `Player` subclass of `Entity` described later in this document, you are not required to provide the syntax for setting the new `Player` instance's hunger, thirst, health, and inventory. Only the position needs to be communicated in the `__repr__` output, because this is the only attribute that can be replicated via the `Player.__init__` method.

Examples

```
>>> entity = Entity((2, 3))
>>> entity.get_position()
(2, 3)
>>> entity.get_name()
'Entity'
>>> entity.get_id()
'E'
>>> str(entity)
'E'
>>> entity
Entity((2, 3))
```

DynamicEntity *(class)*

Inherits from Entity
DynamicEntity is an abstract class which provides base functionality for special types of Entities that are dynamic (e.g. can move from their original position).

`set_position(self, new_position: tuple[int, int]) -> None` *(method)*

Updates the DynamicEntity's position to `new_position`, assuming it is a valid position.

Examples

```
>>> dynamic_entity = DynamicEntity((1, 1))
>>> dynamic_entity.get_position()
(1, 1)
>>> dynamic_entity.set_position((2, 3))
>>> dynamic_entity.get_position()
(2, 3)
>>> dynamic_entity.get_id()
'DE'
>>> str(dynamic_entity)
'DE'
>>> dynamic_entity
DynamicEntity((2, 3))
```

Player *(class)*

Inherits from DynamicEntity
Player is a DynamicEntity that is controlled by the user, and must move from its original position to the end of each maze. A player has health points (HP) which starts at 100, hunger and thirst which both start at 0, and an inventory (see Inventory class defined later in this section). Note that you will need to develop Player, Item and its subclasses, and Inventory in parallel, as they interact in non-trivial ways. You may choose to start by implementing all player functionality that does not involve the inventory, implementing items, implementing the inventory, and then returning to the Player class to complete the inventory functionality.

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

Returns the player's current hunger level.

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

Returns the player's current thirst level.

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

Returns the player's current HP.

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

Alters the player's hunger level by the given `amount`. You must cap the player's hunger to be between 0 and 10 (inclusive).

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

Alters the player's thirst level by the given `amount`. You must cap the player's thirst to be between 0 and 10 (inclusive).

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

Alters the player's health level by the given `amount`. You must cap the player's health to be between 0 and 100 (inclusive).

`get_inventory(self) -> Inventory` *(method)*

Returns the player's Inventory instance.

`add_item(self, item: Item) -> None` *(method)*

Adds the given `item` to the player's Inventory instance.

Examples

```
>>> player = Player((2, 3))
>>> player.get_hunger()
```

```
0
>>> player.get_thirst()
0
>>> player.get_health()
100
>>> player.change_hunger(4)
>>> player.change_thirst(3)
>>> player.change_health(-34)
>>> player.get_hunger()
4
>>> player.get_thirst()
3
>>> player.get_health()
66
>>> player.get_inventory().get_items()
{}
>>> player.add_item(Honey((2, 3)))
>>> player.add_item(Honey((4, 4)))
>>> player.add_item(Water((1, 1)))
>>> player.get_inventory().get_items()
{'Honey': [Honey((2, 3)), Honey((4, 4))], 'Water': [Water((1, 1))]}
>>> player
Player((2, 3))
```

Item *(abstract class)*

Inherits from Entity
Subclass of Entity which provides base functionality for all items in the game.

`apply(self, player: Player) -> None` *(abstract method)*

Applies the items effect, if any, to the given player.

Examples

```
>>> player = Player((2, 3))
>>> item = Item((4, 5))
>>> item.apply(player)
Traceback (most recent call last):
...
NotImplementedError
>>> item.get_position()
(4, 5)
>>> item.get_name()
'Item'
>>> item.get_id()
'I'
>>> str(item)
'I'
>>> item
Item((4, 5))
```

Potion *(class)*

Inherits from Item
A potion is an item that increases the player's HP by 20 when applied.

Examples

```
>>> player = Player((1, 1))
>>> potion = Potion((1, 1))
>>> player.change_health(-50)
>>> player.get_health()
50
>>> potion.apply(player)
>>> player.get_health()
70
>>> potion
Potion((1, 1))
```

Coin *(class)*

Inherits from Item
A coin is an item that has no effect when applied to a player.

Note: The purpose of a coin is to be collected and stored in a players inventory. However, the act of adding the coin to the players inventory is not done within this class.

Examples

```
>>> player = Player((4, 4))
>>> coin = Coin((4, 4))
>>> print(player.get_health(), player.get_thirst(), player.get_hunger())
100 0 0
>>> player.get_inventory().get_items()
{}
>>> coin.apply(player)
>>> print(player.get_health(), player.get_thirst(), player.get_hunger())
100 0 0
>>> player.get_inventory().get_items()
{}
```

Water *(class)*

Inherits from Item
Water is an item that will decrease the player's thirst by 5 when applied.

Examples

```
>>> player = Player((1, 1))
>>> player.change_thirst(8)
>>> player.get_thirst()
8
>>> water = Water((1, 1))
>>> water.apply(player)
>>> player.get_thirst()
3
>>> water.get_id()
'W'
>>> str(water)
'W'
>>> water
Water((1, 1))
```

Food *(abstract class)*

Inherits from Item
Food is an abstract class. Food subclasses implement an apply method that decreases the players hunger by a certain amount, depending on the type of food. The examples below describe the usage of the two food subclasses (Honey and Apple) described in the next two sections.

Examples

```
>>> player = Player((1, 1))
>>> player.change_hunger(7)
>>> apple = Apple((1, 1))
>>> honey = Honey((2, 3))
>>> player.get_hunger()
7
>>> apple.apply(player)
>>> player.get_hunger()
6
>>> honey.apply(player)
>>> player.get_hunger()
1
>>> apple.get_id()
'A'
>>> honey.get_id()
'H'
>>> honey
Honey((2, 3))
>>> apple
Apple((1, 1))
```

Apple *(class)*

Inherits from Food
Apple is a type of food that decreases the player's hunger by 1 when applied.

Honey *(class)*

Inherits from Food
Honey is a type of food that decreases the player's hunger by 5 when applied.

Inventory *(class)*

An Inventory contains and manages a collection of items.

`__init__(self, initial_items: Optional[list[Item,...]] = None) -> None` *(method)*

Sets up initial inventory. If no `initial_items` is provided, inventory starts with an empty dictionary for the items. Otherwise, the initial dictionary is set up from the `initial_items` list to be a dictionary mapping item names to a list of item instances with that name.

`add_item(self, item: Item) -> None` *(method)*

Adds the given item to this inventory's collection of items.

`get_items(self) -> dict[str, list[Item,...]]` *(method)*

Returns a dictionary mapping the names of all items in the inventory to lists containing each instance of the item with that name.

`remove_item(self, item_name: str) -> Optional[Item]` *(method)*

Removes and returns the first instance of the item with the given `item_name` from the inventory. If no item exists in the inventory with the given name, then this method returns None.

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

Returns a string containing information about quantities of items available in the inventory.

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

Returns a string that could be used to construct a new instance of `Inventory` containing the same items as `self` currently contains. Note that the order of the `initial_items` is not important for this method.

Examples

```
>>> inventory = Inventory([Water((1, 2)), Honey((2, 3)), Water((3, 4))])
>>> inventory
Inventory(initial_items=[Water((1, 2)), Water((3, 4)), Honey((2, 3))])
>>> inventory.get_items()
{'Water': [Water((1, 2)), Water((3, 4))], 'Honey': [Honey((2, 3))]}
>>> inventory.add_item(Honey((3, 4)))
>>> inventory.add_item(Coin((1, 1)))
>>> inventory.get_items()
{'Water': [Water((1, 2)), Water((3, 4))], 'Honey': [Honey((2, 3)), Honey((3, 4))],
 'Coin': [Coin((1, 1))]}
>>> inventory.remove_item('Honey')
Honey((2, 3))
>>> inventory.get_items()
{'Water': [Water((1, 2)), Water((3, 4))], 'Honey': [Honey((3, 4))], 'Coin': [Coin((1, 1))]}
>>> inventory.remove_item('Coin')
Coin((1, 1))
>>> non_existant_coin = inventory.remove_item('Coin')
>>> print(non_existant_coin)
None
>>> inventory.get_items()
{'Water': [Water((1, 2)), Water((3, 4))], 'Honey': [Honey((3, 4))]}
>>> print(inventory)
Water: 2
Honey: 1
>>> inventory
Inventory(initial_items=[Water((1, 2)), Water((3, 4)), Honey((3, 4))])
```

Maze *(class)*

A Maze instance represents the space in which a level takes place. The maze does not know what entities are placed on it or where the entities are; it only knows what tiles it contains and what dimensions it has.

`__init__(self, dimensions: tuple[int, int]) -> None` *(method)*

Sets up an empty maze of given dimensions (a tuple of the number of rows and number of columns).

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

Returns the (#rows, #columns) in the maze.

`add_row(self, row: str) -> None` *(method)*

Adds a row of tiles to the maze. Each character in `row` is a Tile ID which can be used to construct the appropriate Tile instance to place in that position of the row of tiles. A precondition of this method is that the addition of a row must not violate the maze dimensions. Note: if a `row` string contains an empty space or an invalid or unknown Tile ID, an `Empty` tile should be placed in that position.

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

Returns the Tile instances in this maze. Each element is a row (list of Tile instances in order).

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

Unlocks any doors that exist in the maze.

`get_tile(self, position: tuple[int, int]) -> Tile` *(method)*

Returns the Tile instance at the given position.

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

Returns the string representation of this maze. Each line in the output is a row in the maze (each Tile instance is represented by its ID).

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

Returns a string that could be copied and pasted to construct a new Maze instance with the same dimensions as this Maze instance.

Examples

```
>>> maze = Maze((5, 5))
>>> maze.get_dimensions()
(5, 5)
>>> maze.get_tiles()
[]
>>> str(maze)
''
>>> maze.add_row('#####')
>>> maze.add_row('# C D')
>>> maze.add_row('# C #')
>>> maze.add_row('P C #')
>>> maze.add_row('#####')
>>> from pprint import pprint
>>> pprint(maze.get_tiles()) # unknown tile IDs treated as empty
[[Wall(), Wall(), Wall(), Wall(), Wall()],
[Wall(), Empty(), Empty(), Empty(), Door()],
[Wall(), Empty(), Empty(), Empty(), Wall()],
[Empty(), Empty(), Empty(), Empty(), Wall()],
[Wall(), Wall(), Wall(), Wall(), Wall()]]
>>> str(maze)
'#####\n#  D\n#  #\n   #\n#####'
>>> print(maze)
#####
#  D
#  #
   #
#####
>>> maze
Maze((5, 5))
>>> maze.get_tile((2, 3))
Empty()
>>> maze.unlock_door()
>>> print(maze)
#####
#
#  #
   #
#####
>>> pprint(maze.get_tiles())
[[Wall(), Wall(), Wall(), Wall(), Wall()],
[Wall(), Empty(), Empty(), Empty(), Door()],
[Wall(), Empty(), Empty(), Empty(), Wall()],
[Empty(), Empty(), Empty(), Empty(), Wall()],
[Wall(), Wall(), Wall(), Wall(), Wall()]]
```

Level *(class)*

A Level instance keeps track of both the maze *and* the non-player entities placed on the maze for a single level.

`__init__(self, dimensions: tuple[int, int]) -> None` *(method)*

Sets up a new level with empty maze using the given dimensions. The level is set up initially with no items or player.

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

Returns the Maze instance for this level.

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

Unlocks the doors in the maze if there are no coins remaining.

`add_row(self, row: str) -> None` *(method)*

Adds the tiles and entities from the row to this level. `row` is a string containing the Tile IDs and Entity IDs to place in this row.

`add_entity(self, position: tuple[int, int], entity_id: str) -> None` *(method)*

Adds a new entity to this level in the given `position`. Entity type is determined by the `entity_id`. The information you store from this method may differ depending on the specific type of the entity requested. If the entity is an item, you should store it with its position in such a way that, if an item existed at that position previously, this new item will replace it.

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

Returns the (#rows, #columns) in the level maze.

`get_items(self) -> dict[tuple[int, int], Item]` *(method)*

Returns a mapping from position to the Item at that position for all items currently in this level.

`remove_item(self, position: tuple[int, int]) -> None` *(method)*

Deletes the item from the given position. A precondition of this method is that there is an Item instance at the position.

`add_player_start(self, position: tuple[int, int]) -> None` *(method)*

Adds the start position for the player in this level.

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

Returns the starting position of the player for this level. If no player start has been defined yet, this method returns None.

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

Returns a string representation of this level.

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

Returns a string that could be copied and pasted to construct a new Level instance with the same dimensions as this Level instance.

Examples

```
>>> level = Level((5, 5))
>>> level
Level((5, 5))
```

```
>>> level.get_maze()
Maze((5, 5))
>>> level.get_maze().get_tiles()
[]
>>> level.get_items()
{}
>>> level.get_player_start()
>>> level.get_dimensions()
(5, 5)
>>> level.add_row('#####')
>>> level.add_row('# C D')
>>> level.add_row('# C #')
>>> level.add_row('P C #')
>>> level.add_row('#####')
>>> print(level.get_maze())
#####
#   D
#   #
    #
#####
>>> level.get_items()
{(1, 2): Coin((1, 2)), (2, 2): Coin((2, 2)), (3, 2): Coin((3, 2))}
>>> level.get_player_start()
(3, 0)
>>> level.add_entity((2, 3), 'M')
>>> level.get_items()
{(1, 2): Coin((1, 2)), (2, 2): Coin((2, 2)), (3, 2): Coin((3, 2)), (2, 3): Potion((2, 3))}
>>> level.attempt_unlock_door()
>>> print(level.get_maze())
#####
#   D
#   #
    #
#####
>>> level.remove_item((1, 2))
>>> level.remove_item((2, 2))
>>> level.remove_item((3, 2))
>>> level.get_items()
{(2, 3): Potion((2, 3))}
>>> level.attempt_unlock_door()
>>> print(level.get_maze())
#####
#
#   #
    #
#####
>>> from pprint import pprint
>>> pprint(level.get_maze().get_tiles())
[[Wall(), Wall(), Wall(), Wall(), Wall()],
[Wall(), Empty(), Empty(), Empty(), Door()],
[Wall(), Empty(), Empty(), Empty(), Wall()],
[Empty(), Empty(), Empty(), Empty(), Wall()],
[Wall(), Wall(), Wall(), Wall(), Wall()]]
>>> print(level)
Maze: #####
#
#   #
    #
#####
```

```
Items: {(2, 3): Potion((2, 3))}
Player start: (3, 0)
```

Model *(class)*

This is the model class that the controller uses to understand and mutate the game state. The model keeps track of a Player, and multiple Level instances. The Model class must provide the interface through which the controller can request information about the game state, and request changes to the game state.

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

Sets up the model from the given `game_file`, which is a path to a file containing game information (e.g. `games/game1.txt`). Once you have written the `Level` class, you can uncomment the provided `load_game` function and use it to aid in your implemention of the this method.

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

Returns True if the game has been won, otherwise returns False. A game has been won if all the levels have been successfully completed.

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

Returns True if the game has been lost, otherwise returns False (HP too low or hunger or thirst too high).

`get_level(self) -> Level` *(method)*

Returns the current level.

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

Changes the level to the next level in the game. If no more levels remain, the player has won the game.

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

Returns True if the player just moved to the next level on the previous turn, otherwise returns False.

`move_player(self, delta: tuple[int, int]) -> None` *(method)*

Tries to move the player by the requested (row, column) change (`delta`). This method should also level up if the player finished the maze by making the move. If the player did not level up and the tile that the player is moving into is non-blocking, this method should:

- Update the players hunger, thirst based on the number of moves made.
- Update players health based on the successful movement and the damage caused by the tile the player has moved onto.
- Update the players position.
- Attempt to collect any item that is on the players new position.

`attempt_collect_item(self, position: tuple[int, int]) -> None` *(method)*

Collects the item at the given position if one exists. Unlocks the door if all coins have been collected.

`get_player(self) -> Player` *(method)*

Returns the player in the game.

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

Returns the player's current stats as (HP, hunger, thirst).

`get_player_inventory(self) -> Inventory` *(method)*

Returns the players inventory.

get_current_maze(self) -> Maze *(method)*

Returns the Maze for the current level.

get_current_items(self) -> dict[tuple[int, int], Item] *(method)*

Returns a dictionary mapping tuple positions to the item that currently exists at that position on the maze. Only positions at which an item exists should be included in the result.

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

Returns the text required to construct a new instance of `Model` with the same game file used to construct `self`.

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

Does the same thing as `__str__`.

Examples

```
>>> model = Model('games/game1.txt')
>>> print(model.get_level())
Maze: #####
#   D
#   #
    #
#####
Items: {(1, 2): Coin((1, 2)), (2, 2): Coin((2, 2)), (3, 2): Coin((3, 2))}
Player start: (3, 0)
>>> model.has_won()
False
>>> model.has_lost()
False
>>> model.did_level_up()
False
>>> model.move_player((0, 1))
>>> model.move_player((0, 1))
>>> model.move_player((-1, 0))
>>> print(model.get_level())
Maze: #####
#   D
#   #
    #
#####
Items: {(1, 2): Coin((1, 2))}
Player start: (3, 0)
>>> print(model.get_player().get_position())
(2, 2)
>>> model.get_player_inventory().get_items()
{'Coin': [Coin((3, 2)), Coin((2, 2))]}
>>> model.get_current_items()
{(1, 2): Coin((1, 2))}
>>> model.get_player_stats()
(97, 0, 0)
>>> print(model.get_current_maze())
#####
#   D
#   #
    #
#####
>>> model.attempt_collect_item((1, 2))
>>> model.get_current_items()
```

```
{}
>>> print(model.get_current_maze())
#####
#
#   #
    #
#####
>>> model.level_up()
>>> model.did_level_up()
True
>>> print(model.get_current_maze())
########
       #
###### #
#      #
# ######
#      #
######D#
>>> str(model)
"Model('games/game1.txt')"
>>> model
Model('games/game1.txt')
```

5.2 Controller class

MazeRunner *(class)*

MazeRunner is the controller class, which should maintain instances of the model and view, collect user input and facilitate communication between the model and view. The methods you *must* implement are outlined below, but you are strongly encouraged to implement your own helper methods where possible.

`__init__(self, game_file: str, view: UserInterface) -> None` *(method)*

Creates a new MazeRunner game with the given `view` and a new Model instantiated using the given `game_file`.

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

Executes the entire game until a win or loss occurs. When the `a2.py` file is run, a MazeRunner instance is created and this method is run. As such, this method should cause all of the program output. For examples of how this method should operate, see the `gameExamples/` folder, which contains example outputs of full MazeRunner games.

6 Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. apply basic object-oriented concepts such as classes, instances and methods,
3. read and analyse code written by others,
4. analyse a problem and design an algorithmic solution to the problem,
5. read and analyse a design and be able to translate the design into a working program, and
6. apply techniques for testing and debugging.

6.1 Marking Breakdown

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

6.2 Functionality Marking

Your program's functionality will be marked out of a total of 50 marks. As in assignment 1, your assignment will be put through a series of tests and your functionality mark will be proportional to the number of tests you pass. You will be given a *subset* of the functionality tests before the due date for the assignment. You may receive partial marks within each class for partially working methods, or for implementing only a few classes.

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 the Python interpreter (the IDLE environment). Partial solutions will be marked, but if there are errors in your code that cause the interpreter to fail to execute your program, you will get zero for functionality marks. If there is a part of your code that causes the interpreter to fail, comment out the code so that the remainder can run. Your program must run using the Python 3.10 interpreter. If it runs in another environment (e.g. Python 3.9 or PyCharm) but not in the Python 3.10 interpreter, you will get zero for the functionality mark.

6.3 Style Marking

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 also be out of 50.

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

- Readability
  - Program Structure: Layout of code makes it easier to read and follow its logic. This includes using whitespace to highlight blocks of logic.
  - Descriptive Identifier Names: Variable, constant, function, class and method names clearly describe what they represent in the program's logic. Do **not** use what is called the *Hungarian Notation* for identifiers. In short, this means do not include the identifier's type in its name (e.g. `item_list`), rather make the name meaningful. (e.g. Use `items`, where plural informs the reader it is a collection of items and it can easily be changed to be some other collection and not a list.) The main reason for this restriction is that most people who follow the *Hungarian Notation* convention, use it poorly (including Microsoft).
  - Named Constants: All non-trivial fixed values (literal constants) in the code are represented by descriptive named (symbolic) constants.
- Documentation
  - Comment Clarity: Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0.`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.
  - Informative Docstrings: Every class, method and function should have a docstring that summarises its purpose. This includes describing parameters and return values so that others can understand

how to use the method or function correctly.

- Description of Logic: All significant blocks of code should have a comment to explain how the logic works. For a small method or function, the logic should usually be clear from the code and docstring. For long or complex methods or functions, each logical block should have an in-line comment describing its logic.

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

- Object-Oriented Program Structure
  - Classes & Instances: Objects are used as entities to which messages are sent, demonstrating understanding of the differences between classes and instances.
  - Encapsulation: Classes are designed as independent modules with state and behaviour. Methods only directly access the state of the object on which they were invoked. Methods never update the state of another object.
  - Inheritance & Polymorphism: Subclasses are designed as specialised versions of their superclasses. Subclasses extend the behaviour of their superclass without re-implementing behaviour, or breaking the superclass behaviour or design. Subclasses redefine behaviour of appropriate methods to extend the superclasses' type. Subclasses do not break their superclass' interface.
- Algorithmic Logic
  - Single Instance of Logic: Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a method or function.
  - Variable Scope: Variables should be declared locally in the method or function in which they are needed. Attributes should be declared clearly within the `__init__` method. Class variables are avoided, except where they simplify program logic. Global variables should not be used.
  - Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. loops and conditional statements).

6.4 Documentation Requirements

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

6.5 Assignment Submission

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

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

6.6 Plagiarism

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