Key Cave Adventure Game

Assignment 2
Version 1.0: 07/09/20
CSSE1001/7030
Due, 28th September
15 marks

Gameplay

Key Cave Adventure Game is a single-player dungeon-crawler game where the player adventurously explores a dungeon. The objective is for the player to find the key and get out of the dungeon through the door. This game is full of adventure. The game play utilises simple key commands. The positions of Entities are always represented as (row, col).

Commands

Input	Action
`'-DIRECTION-'`	Direction to move in
`'I -DIRECTION-'`	`'I W'` would investigate the Entity in the given direction
`'H'`	Help text (see examples at end)
`'Q'`	Quit

Table 1. List of valid actions.

Input	Direction
`'W'`	Up
`'S'`	Down
`'A'`	Left
`'D'`	Right

Table 2. List of valid Directions.

Character	Description

'#'	WALL
'O'	PLAYER
'K'	KEY
'D'	DOOR
' '	SPACE

Table 3. List of characters used in the game

The Player can move within the dungeon by using any of the valid direction inputs outlined in Table 2. The Player must collect the KEY by walking into it and then the DOOR can be unlocked.

Game over

The Player ***wins*** the game by collecting the KEY and heading to the DOOR. This message should display: **You have won the game with your strength and honour!**
If the Player runs out of *moves* before they collect the KEY and unlock the DOOR then the Player loses the game. This message should display: **You have lost all your strength and honour.**

Getting Started

Files

a2.py - file where you write your submission
game**n**.txt - the **n**th dungeon layout. There will be multiple provided. “game1.txt” is the best one to start from.

To start, download *a2_files.zip* from Blackboard and extract the contents. This folder contains all the necessary files to start this assignment. You will be required to implement your assignment in *a2.py*. Do not modify any files from *a2.zip* except *a2.py*. The only file that you are required to submit is *a2.py*. Some support code has been provided to assist with implementing the tasks.

Classes

Many modern software programs are implemented with classes and this assignment will give you some practice in working with them. The class structure that you will be working with is shown diagrammatically in Figure 1.

In Figure 1 an arrow with a solid head indicates that one class stores an instance of another class. (This is analogous, say, to a list storing instances of strings within it, say, sample_list=[‘dog’, ‘cat’]). In Figure 1, for example, GameApp points to Display meaning that GameApp contains an instance of Display.
An arrow with an open head indicates that one class is a subclass of another class. I.e. one class inherits from another. E.g. Item is a subclass of Entity.

You can add classes to reduce code duplication. This can include adding your own private helper methods. However, you must include the classes outlined below.

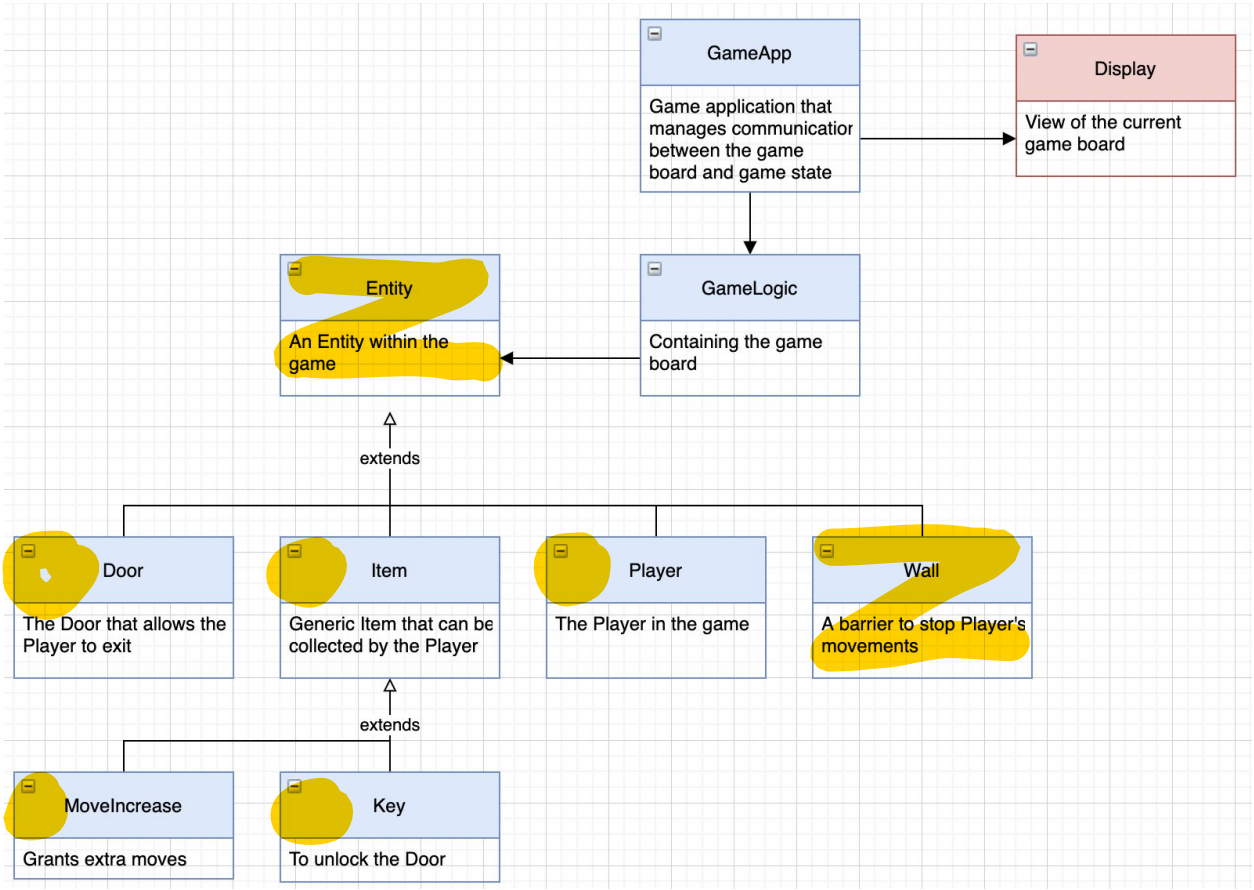

Figure 1. Class structure for the software program

Entity

A generic Entity within the game.

Each Entity has an id, and can either be collided with (two entities can be in the same position) or not (two entities cannot be in the same position.) The collidable attribute should be set to True for an Entity upon creation. Entity should be constructed with Entity(). Entity should have the following methods defined:

`get_id(self) -> str`: Returns a string that represents the Entity's ID.

`set_collide(self, collidable: bool)`: Set the collision state for the Entity to be True.

`can_collide(self) -> bool`: Returns True if the Entity can be collided with (another Entity can share the position that this one is in) and False otherwise.

`__str__(self) -> str`: Returns the string representation of the Entity. See example output below.

`__repr__(self) -> str`: Same as str(self).

Examples

```
>>> entity = Entity()
>>> entity.get_id()
'Entity'
>>> entity.can_collide()
True
>>> str(entity)
"Entity('Entity')"
>>> repr(entity)
"Entity('Entity')"
>>> entity = Entity()
>>> entity.can_collide()
True
>>> entity.set_collide(False)
>>> entity.can_collide()
False
(Note that Entity should never be set to False, this is just an example)
```

Wall

A Wall is a special type of an Entity within the game.

The Wall Entity cannot be collided with. Wall should be constructed with Wall(). Wall should have the following methods defined:

`get_id(self) -> str`: Returns a string that represents the Wall's ID.

`set_collide(self, collidable: bool)`: Set the collision state for the Wall to be False.

`can_collide(self) -> bool`: Returns True if the Wall can be collided with and False otherwise.

`__str__(self) -> str`: Returns the string representation of the Wall. See example output below.

`__repr__(self) -> str`: Same as str(self).

Examples

```
>>> wall = Wall()
>>> wall.get_id()
'#'
>>> wall.can_collide()
False
>>> str(wall)
"Wall('#')"
>>> repr(wall)
"Wall('#')"
>>> wall = Wall()
>>> entity.can_collide()
False
>>> entity.set_collide(True)
>>> entity.can_collide()
True
(Note that Wall should never be set to True, this is just an example)
```

Item

An Item is a special type of an Entity within the game. This is an abstract class.

By default the Item Entity can be collided with. Item should be constructed with Item(). Item should have the following methods defined:

`get_id(self) -> str`: Returns a string that represents the Item's ID.

`set_collide(self, collidable: bool)`: Set the collision state for the Item to be True.

`can_collide(self) -> bool`: Returns True if the Item can be collided with and False otherwise.

`on_hit(self, game: GameLogic)`: This function should raise the NotImplementedError.

Examples

```
>>> game = GameLogic()
>>> item = Item()
>>> item.can_collide()
True
>>> item.on_hit(game)
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    item.on_hit(game)
  File "/Users/.../a2.py", line 75, in on_hit
    class Item(Entity):
NotImplementedError
>>> str(item)
"Item('Entity')"
>>> repr(item)
"Item('Entity')"
```

Key

A Key is a special type of Item within the game.

The Key Item can be collided with. Key should be constructed with Key(). Key should have the following methods defined:

`get_id(self) -> str`: Returns a string that represents the Key's ID.

`set_collide(self, collidable: bool)`: Set the collision state for the Key to be True.

`can_collide(self) -> bool`: Returns True if the Key can be collided with and False otherwise.

`on_hit(self, game: GameLogic) -> None`: When the player takes the Key the Key should be added to the Player's inventory. The Key should then be removed from the dungeon once it's in the Player's inventory.
(hint: Move onto the Player and GameLogic class and come back to this when they are partly implemented.)

`__str__(self) -> str`: Returns the string representation of the Key. See example output below.

`__repr__(self) -> str`: Same as str(self).

Examples

```
>>> key = Key()
>>> key.get_id()
'K'
>>> key.can_collide()
True
>>> str(key)
"Key('K')"
>>> repr(key)
"Key('K')"
(below are example of the on_hit() method)
>>> game = GameLogic()
>>> game.get_game_information()
{(1, 3): Key('K'), (3, 2): Door('D'), (0, 0): Wall('#'), (0, 1):
Wall('#'), (0, 2): Wall('#'), (0, 3): Wall('#'), (0, 4): Wall('#'),
(1, 0): Wall('#'), (1, 2): Wall('#'), (1, 4): Wall('#'), (2, 0):
Wall('#'), (2, 4): Wall('#'), (3, 0): Wall('#'), (3, 4): Wall('#'),
(4, 0): Wall('#'), (4, 1): Wall('#'), (4, 2): Wall('#'), (4, 3):
Wall('#'), (4, 4): Wall('#')}
>>> player = game.get_player()
>>> player.get_inventory()
[]
>>> player.set_position((1,3))
>>> key.on_hit(game)
>>> player.get_inventory()
[Key('K')]
>>> game.get_game_information()
{(3, 2): Door('D'), (0, 0): Wall('#'), (0, 1): Wall('#'), (0, 2):
Wall('#'), (0, 3): Wall('#'), (0, 4): Wall('#'), (1, 0): Wall('#'),
(1, 2): Wall('#'), (1, 4): Wall('#'), (2, 0): Wall('#'), (2, 4):
Wall('#'), (3, 0): Wall('#'), (3, 4): Wall('#'), (4, 0): Wall('#'),
(4, 1): Wall('#'), (4, 2): Wall('#'), (4, 3): Wall('#'), (4, 4):
Wall('#')}
```

MoveIncrease

MoveIncrease is a special type of Item within the game.

The MoveIncrease Item can be collided with. MoveIncrease should be constructed with MoveIncrease(moves=5: int) where moves describe how many extra moves the Player will be granted when they collect this Item, the default value should be 5. MoveIncrease should have the following methods defined:

`get_id(self) -> str`: Returns a string that represents the MoveIncrease's ID.

`set_collide(self, collidable: bool)`: Set the collision state for the MoveIncrease to be True.

`can_collide(self) -> bool`: Returns True if the MoveIncrease can be collided with and False otherwise.

`on_hit(self, game: GameLogic) -> None`: When the player hits the MoveIncrease (M) item the number of moves for the player increases and the M item is removed from the game. These actions are implemented via the on_hit method. Specifically, extra moves should be granted to the Player and the M item should be removed from the game.
(hint: Move onto the Player and GameLogic class and come back to this when they are partly implemented.)

`__str__(self) -> str`: Returns the string representation of the MoveIncrease. See example output below.

`__repr__(self) -> str`: Same as str(self).

Examples

```
>>> move_increase = MoveIncrease()
>>> move_increase.get_id()
'M'
>>> move_increase.can_collide()
True
>>> str(move_increase)
"MoveIncrease('M')"
>>> repr(move_increase)
"MoveIncrease('M')"
(below are example of the on_hit() method)
>>> game = GameLogic()
>>> game.get_game_information()
{(1, 6): Key('K'), (6, 3): Door('D'), (0, 0): Wall('#'), (0, 1):
Wall('#'), (0, 2): Wall('#'), (0, 3): Wall('#'), (0, 4): Wall('#'),
(0, 5): Wall('#'), (0, 6): Wall('#'), (0, 7): Wall('#'), (1, 0):
Wall('#'), (1, 4): Wall('#'), (1, 7): Wall('#'), (2, 0): Wall('#'),
(2, 7): Wall('#'), (3, 0): Wall('#'), (3, 7): Wall('#'), (4, 0):
```

```
Wall('#'), (4, 1): Wall('#'), (4, 2): Wall('#'), (4, 7): Wall('#'),
(5, 0): Wall('#'), (5, 7): Wall('#'), (6, 0): Wall('#'), (6, 7):
Wall('#'), (7, 0): Wall('#'), (7, 1): Wall('#'), (7, 2): Wall('#'),
(7, 3): Wall('#'), (7, 4): Wall('#'), (7, 5): Wall('#'), (7, 6):
Wall('#'), (7, 7): Wall('#'), (6, 6): MoveIncrease('M')}
>>> player = game.get_player()
>>> player.set_position((6,6))
>>> player.moves_remaining()
12
>>> move_increase.on_hit(game)
>>> player.moves_remaining()
17
>>> game.get_game_information()
{(1, 6): Key('K'), (6, 3): Door('D'), (0, 0): Wall('#'), (0, 1):
Wall('#'), (0, 2): Wall('#'), (0, 3): Wall('#'), (0, 4): Wall('#'),
(0, 5): Wall('#'), (0, 6): Wall('#'), (0, 7): Wall('#'), (1, 0):
Wall('#'), (1, 4): Wall('#'), (1, 7): Wall('#'), (2, 0): Wall('#'),
(2, 7): Wall('#'), (3, 0): Wall('#'), (3, 7): Wall('#'), (4, 0):
Wall('#'), (4, 1): Wall('#'), (4, 2): Wall('#'), (4, 7): Wall('#'),
(5, 0): Wall('#'), (5, 7): Wall('#'), (6, 0): Wall('#'), (6, 7):
Wall('#'), (7, 0): Wall('#'), (7, 1): Wall('#'), (7, 2): Wall('#'),
(7, 3): Wall('#'), (7, 4): Wall('#'), (7, 5): Wall('#'), (7, 6):
Wall('#'), (7, 7): Wall('#')}
```

Door

A Door is a special type of an Entity within the game.

The Door Entity can be collided with (The Player should be able to share its position with the Door when the Player enters the Door.) Door should be constructed with Door(). Door should have the following methods defined:

`get_id(self) -> str`: Returns a string that represents the Door's ID.

`set_collide(self, collidable: bool)`: Set the collision state for the Door to be True.

`can_collide(self) -> bool`: Returns True if the Door can be collided with and False otherwise.

`on_hit(self, game: GameLogic) -> None`: If the Player's inventory contains a Key Entity then this method should set the 'game over' state to be True.

(hint: Move onto the Player and GameLogic class and come back to this when they are partly implemented.)

`__str__(self) -> str`: Returns the string representation of the Door. See example output below.

`__repr__(self) -> str`: Same as str(self).

If the Player's Inventory does not contain a Key Entity then the following message should be displayed: "You don't have the key!"

Examples

```
>>> door = Door()
>>> door.get_id()
'D'
>>> door.can_collide()
True
>>> str(door)
"Door('D')"
>>> repr(door)
"Door('D')"
(below are example of the on_hit() method)
>>> key = Key()
>>> game = GameLogic()
>>> game.get_game_information()
{(1, 3): Key('K'), (3, 2): Door('D'), (0, 0): Wall('#'), (0, 1):
Wall('#'), (0, 2): Wall('#'), (0, 3): Wall('#'), (0, 4): Wall('#'),
(1, 0): Wall('#'), (1, 2): Wall('#'), (1, 4): Wall('#'), (2, 0):
Wall('#'), (2, 4): Wall('#'), (3, 0): Wall('#'), (3, 4): Wall('#'),
(4, 0): Wall('#'), (4, 1): Wall('#'), (4, 2): Wall('#'), (4, 3):
Wall('#'), (4, 4): Wall('#')}
>>> player = game.get_player()
>>> game.won()
False
>>> player.set_position((3,2))
>>> door.on_hit(game)
You don't have the key!
>>> player.get_inventory()
[]
>>> player.set_position((1,3))
>>> key = Key()
>>> key.on_hit(game)
>>> player.get_inventory()
```

```
[Key('K')]
>>> player.set_position((3,2))
>>> door.on_hit(game)
>>> game.won()
True
```

Player

A Player is a special type of an Entity within the game.

The Player Entity can be collided with. The Player should be constructed with Player(move_count: int) where moves represents how many moves a Player can have for the given dungeon they are in (see GAME_LEVELS).

Player should have the following methods defined:

`get_id(self) -> str`: Returns a string that represents the Player's ID.

`set_collide(self, collidable: bool)`: Set the collision state for the Player to be True.

`can_collide(self) -> bool`: Returns True if the Player can be collided with and False otherwise.

`set_position(self, position: tuple<int, int>)`: Sets the position of the Player.

`get_position(self) -> tuple<int, int>`: Returns a tuple of ints representing the position of the Player. If the Player's position hasn't been set yet then this method should return None.

`change_move_count(self, number: int)`: number to be added to the Player's move count.

`moves_remaining(self) -> int`: Returns an int representing how many moves the Player has left before they reach the maximum move count.

`add_item(self, item: Entity)`: Adds the item to the Player's Inventory.

`get_inventory (self) -> list<Entity>`: Returns a list that represents the Player's inventory. If the Player has nothing in their inventory then an empty list should be returned.

`__str__(self) -> str`: Returns the string that shows how many moves the Player has left. See example output below.

`__repr__(self) -> str`: Same as str(self).

Examples

```
>>> player = Player(5)
>>> print(player.get_position())
None
>>> player.set_position((1,2))
>>> player.get_position()
(1, 2)
>>> player.moves_remaining()
5
>>> player.change_move_count(2)
>>> player.moves_remaining()
7
>>> player.get_inventory()
[]
>>> key = Key()
>>> player.add_item(key)
>>> player.get_inventory()
[Key('K')]
>>> str(player)
"Player('O')"
>>> repr(player)
"Player('O')"
```

(Hint: you might want to specify some attributes for this class including: move_count, inventory and position. You can create and initialise the values to be None if you wish.)

GameLogic

GameLogic contains all the game information and how the game should play out. By default, GameLogic should be constructed with `GameLogic(dungeon_name="game1.txt")`. GameLogic should have the following methods defined:

`get_dungeon_size(self) -> int`: Returns the width of the dungeon as an integer.

`init_game_information(self) -> dict<tuple<int, int>: Entity>`: This method should return a dictionary containing the position and the corresponding Entity as the

keys and values respectively. This method also sets the Player's position. At the start of the game this method should be called to find the position of all entities within the current dungeon.

`get_game_information(self) -> dict<tuple<int, int>: Entity>`: Returns a dictionary containing the position and the corresponding Entity, as the keys and values, for the current dungeon.

`get_player(self) -> Player`: This method returns the Player object within the game.

(hint: you should now be able to finish off all the Entity classes/subclasses.)

`get_entity(self, position: tuple<int,int>) -> Entity`: Returns an Entity at a given position in the dungeon. Entity in the given direction or if the position is off map then this function should return None.

`get_entity_in_direction(self, direction: str) -> Entity`: Returns an Entity in the given direction of the Player's position. If there is no Entity in the given direction or if the direction is off map then this function should return None.

`collision_check(self, direction: str) -> bool`: Returns True if a player can travel in the given direction, False otherwise.

`new_position(self, direction: str) -> tuple<int,int>`: Returns a tuple of integers that represents the new position given the direction.

`move_player(self, direction: str) -> None`: Update the Player's position to place them one position in the given direction.

`collision_check(self, direction: str) -> bool`: Returns True if a Player can travel in a given direction, False otherwise.

`check_game_over(self) -> bool`: Return True if the game has been won and False otherwise.

`game_over(self, win: bool) -> None`: Set the game's win state to be True or False.

See example outputs below.
Note that the type hints (eg. select: bool) shown in the methods above are purely for your own understanding. Type hints do not need to be used in your assignment.

Examples

```
>>> game = GameLogic()
```

```
>>> game.get_positions(PLAYER)
[(2, 1)]
>>> game.get_positions(WALL)
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 0), (1, 2), (1, 4), (2,
0), (2, 4), (3, 0), (3, 4), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]
>>> game.init_game_information()
{(1, 3): Key('K'), (3, 2): Door('D'), (0, 0): Wall('#'), (0, 1):
Wall('#'), (0, 2): Wall('#'), (0, 3): Wall('#'), (0, 4): Wall('#'),
(1, 0): Wall('#'), (1, 2): Wall('#'), (1, 4): Wall('#'), (2, 0):
Wall('#'), (2, 4): Wall('#'), (3, 0): Wall('#'), (3, 4): Wall('#'),
(4, 0): Wall('#'), (4, 1): Wall('#'), (4, 2): Wall('#'), (4, 3):
Wall('#'), (4, 4): Wall('#')}
>>> game.get_player()
Player('O')
>>> game.get_positions(KEY)
[(1, 3)]
>>> game.get_entity((2,1))
>>> game.get_entity((1,3))
Key('K')
>>> game.get_entity_in_direction("W")
>>> game.get_entity_in_direction("D")
>>> game.get_entity_in_direction("A")
Wall('#')
>>> game.get_game_information()
{(1, 3): Key('K'), (3, 2): Door('D'), (0, 0): Wall('#'), (0, 1):
Wall('#'), (0, 2): Wall('#'), (0, 3): Wall('#'), (0, 4): Wall('#'),
(1, 0): Wall('#'), (1, 2): Wall('#'), (1, 4): Wall('#'), (2, 0):
Wall('#'), (2, 4): Wall('#'), (3, 0): Wall('#'), (3, 4): Wall('#'),
(4, 0): Wall('#'), (4, 1): Wall('#'), (4, 2): Wall('#'), (4, 3):
Wall('#'), (4, 4): Wall('#')}
>>> game.get_dungeon_size()
5
>>> game.get_player().get_position()
(2, 1)
>>> game.move_player("W")
>>> game.get_player().get_position()
(1, 1)
>>> game.collision_check("W")
True
>>> game.collision_check("S")
False
>>> game.new_position("W")
(0, 1)
```

```
>>> game.new_position("S")
(2, 1)
>>> game.check_game_over()
False
>>> game.won()
False
>>> game.set_win(True)
>>> game.won()
True
```

GameApp

GameApp acts as a communicator between the GameLogic and the Display. GameApp should be constructed with `GameApp()`. GameApp should have the following methods defined:

`play(self) -> None`: Handles the player interaction.
(Hint: this is similar to the main function in a1)

`entity_in_direction(self, direction) -> Entity`: Returns the Entity in a given direction.

`draw(self) -> None`: Displays the dungeon with all Entities in their positions.

See example outputs below.
Note that the type hints (eg. select: bool) shown in the methods above are purely for your own understanding. Type hints do not need to be used in your assignment.

Examples 1

```
#####
# #K#
#O  #
# D #
#####
Moves left: 7

Please input an action: H
Here is a list of valid actions: ['I', 'Q', 'H', 'W', 'S', 'D', 'A']
#####
# #K#
#O  #
# D #
#####
```

```
Moves left: 7

Please input an action: D
#####
# #K#
# O #
# D #
#####
Moves left: 6

Please input an action: D
#####
# #K#
#  O#
# D #
#####
Moves left: 5

Please input an action: I E
Wall('#') is on the E side.
#####
# #K#
#  O#
# D #
#####
Moves left: 4

Please input an action: W
#####
# #O#
#   #
# D #
#####
Moves left: 3

Please input an action: S
#####
# # #
#  O#
# D #
#####
Moves left: 2
```

```
Please input an action: S
#####
# # #
#   #
# DO#
#####
Moves left: 1

Please input an action: A
You have won the game with your strength and honour!
```

Examples 2

```
#####
# #K#
#O  #
# D #
#####
Moves left: 7

Please input an action: D
#####
# #K#
# O #
# D #
#####
Moves left: 6

Please input an action: W
That's invalid.
#####
# #K#
# O #
# D #
#####
Moves left: 5

Please input an action: S
You don't have the key!
#####
# #K#
#   #
# D #
```

```
#####
Moves left: 4

Please input an action: D
#####
# #K#
#   #
# DO#
#####
Moves left: 3

Please input an action: W
#####
# #K#
#  O#
# D #
#####
Moves left: 2

Please input an action: W
#####
# #O#
#   #
# D #
#####
Moves left: 1

Please input an action: S
You have lost all your strength and honour.
```

Example 3

```
########
#   # K#
#O     #
#      #
###    #
#      #
#  D  M#
########
Moves left: 12

Please input an action: D
```

```
########
#    # K#
# O     #
#       #
###     #
#       #
#   D   M#
########
Moves left: 11

Please input an action: D
########
#    # K#
#  O    #
#       #
###     #
#       #
#   D   M#
########
Moves left: 10

Please input an action: D
########
#    # K#
#    O  #
#       #
###     #
#       #
#   D   M#
########
Moves left: 9

Please input an action: D
########
#    # K#
#     O #
#       #
###     #
#       #
#   D   M#
########
Moves left: 8
```

```
Please input an action: D
########
#   # K#
#     O#
#      #
###    #
#      #
#  D  M#
########
Moves left: 7

Please input an action: W
########
#   # O#
#      #
#      #
###    #
#      #
#  D  M#
########
Moves left: 6

Please input an action: S
########
#   #  #
#     O#
#      #
###    #
#      #
#  D  M#
########
Moves left: 5

Please input an action: S
########
#   #  #
#      #
#     O#
###    #
#      #
#  D  M#
########
Moves left: 4
```

```
Please input an action: S
########
#   #  #
#      #
#      #
###   O#
#      #
#  D  M#
########
Moves left: 3

Please input an action: S
########
#   #  #
#      #
#      #
###    #
#     O#
#  D  M#
########
Moves left: 2

Please input an action: S
########
#   #  #
#      #
#      #
###    #
#      #
#  D  O#
########
Moves left: 6

Please input an action: A
########
#   #  #
#      #
#      #
###    #
#      #
#  D O #
########
```

```
Moves left: 5

Please input an action: A
########
#    #  #
#       #
#       #
###     #
#       #
#   DO  #
########
Moves left: 4

Please input an action: A
You have won the game with your strength and honour!
```

Marking

Item	**Marks**
Entity (including Wall and Door)	1
Item (including Key and MoveIncrease)	1.5
Player	2.5
GameLogic	3
GameApp	3

Functionality Assessment

The functionality will be marked out of 11. Your assignment will be put through a series of tests and your functionality mark in any given item category will be proportional to the number of tests you pass. You will be given the majority (but not all) of the functionality tests before the due date for the assignment so that you can gain a good idea of the correctness of your assignment yourself before submitting. You should make sure that your program meets all the specifications given in the assignment. That will ensure that your code passes all the tests. Note: Functionality tests are automated, therefore, string outputs need to exactly match what is expected.

Code Style

The style of your assignment will be assessed by one of the tutors, and you will be marked according to the style rubric provided with the assignment. The style mark will be out of 4.

Assignment Submission

Your assignment must be submitted via the Assignment 2 submission link on Blackboard. You must submit a Python file containing your implementation of the assignment. Late submission of the assignment will not be accepted. 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, so ensure that you have submitted an almost complete version of the assignment well before the submission deadline. Your latest on-time, submission will be marked. Ensure that you submit the correct version of your assignment. In the event of exceptional circumstances, 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 no later than 48 hours prior to the submission deadline. The expectation is that with less than 48 hours before an assignment is due it should be substantially completed and a2.py submittable. Applications for extension, and any 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.