

Pipe

Assignment 2 - V1.1 (10/04/20)

CSSE1001/7030

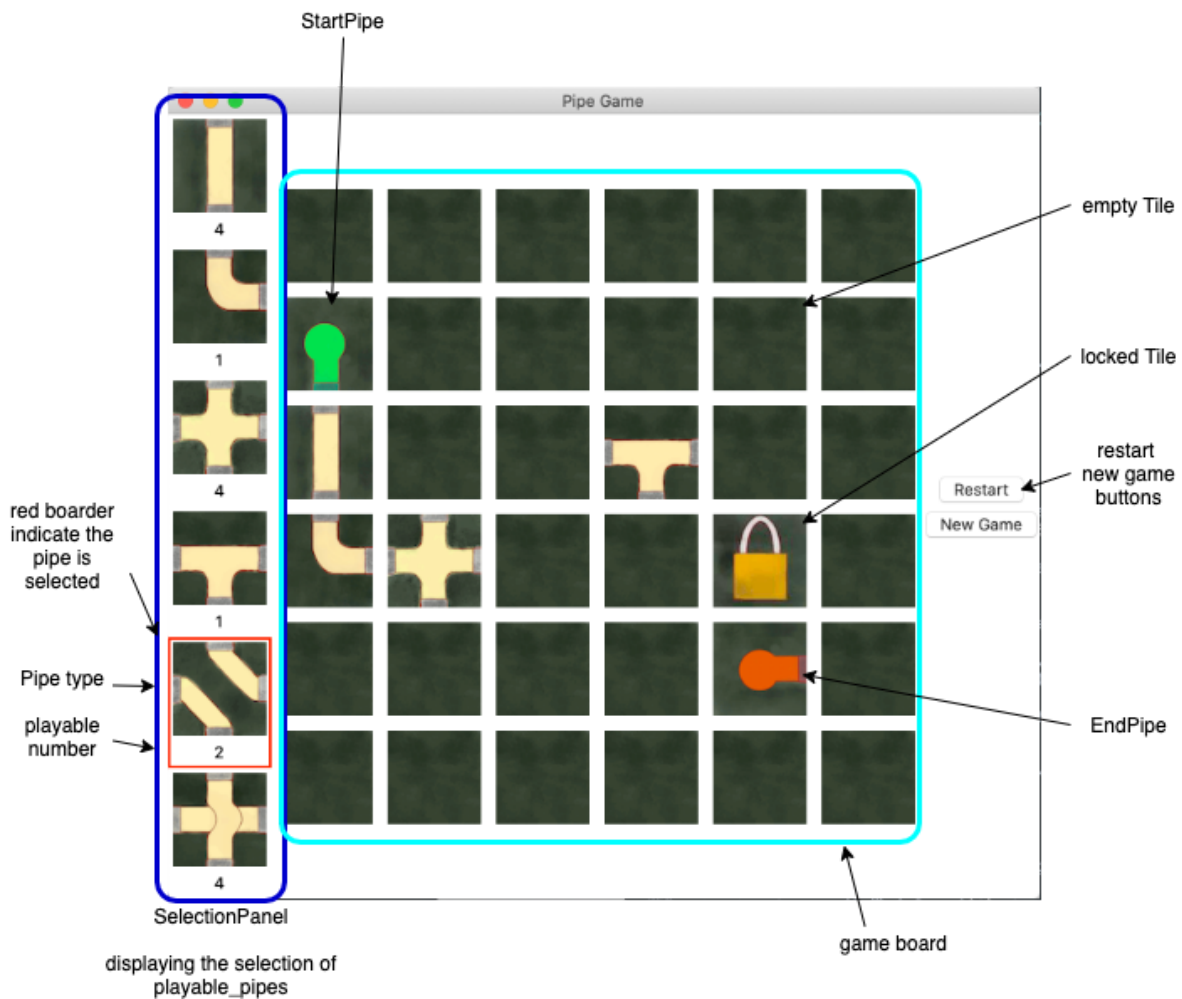
Semester 1, 2020

Due, 1st May

15 marks

Introduction

The model-view-controller (MVC) structure is a popular way to organize code for software projects that involve a Graphical User Interface (GUI). In this structure, the software is broken into three components - the model, the view and the controller. The goal of this assignment is to implement the model component of the software for the game *Pipe*. The code for the view and controller are supplied in support code.



Gameplay

Pipe is a single-player puzzle game where players place various pipe pieces in grid cells. The objective is for the player to connect the **start** pipe to the **end** pipe. The game play utilises simple mouse control.

Selecting and placing pipes

A left click on a pipe in the selection panel should toggle the selection of that pipe piece. A left click in the game board should then place a pipe of the selected type over the selected tile provided that an available pipe piece (i.e. quantity > 0) is selected in the selection panel.

Rotating and removing pipes

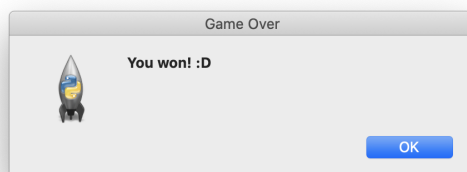
A left click on a pipe in the game board should rotate the given pipe, and a right click on a pipe in the game board should remove that pipe and return it to the selection board, ***provided the pipe was placed by the user***. If a pipe is loaded in from the beginning of the game, it will not be able to be rotated or removed.

“Restart” and “New Game” buttons

The *Restart* button will restart the game. The *New Game* button will prompt for the player to enter the name of the file they wish to load into the game. The name of the file will be automatically passed through to *PipeGame*. This much functionality is implemented already in the support code. However, the *New Game* button will only become functional (actually present the user with the game from the requested file) if you successfully implement the Bonus Task *load_file* function described at the end of this assignment sheet.

Game over

The player **wins** the game by connecting the pipes from start to finish, see Figure 1. On game completion, the player is informed of their outcome via a messagebox.



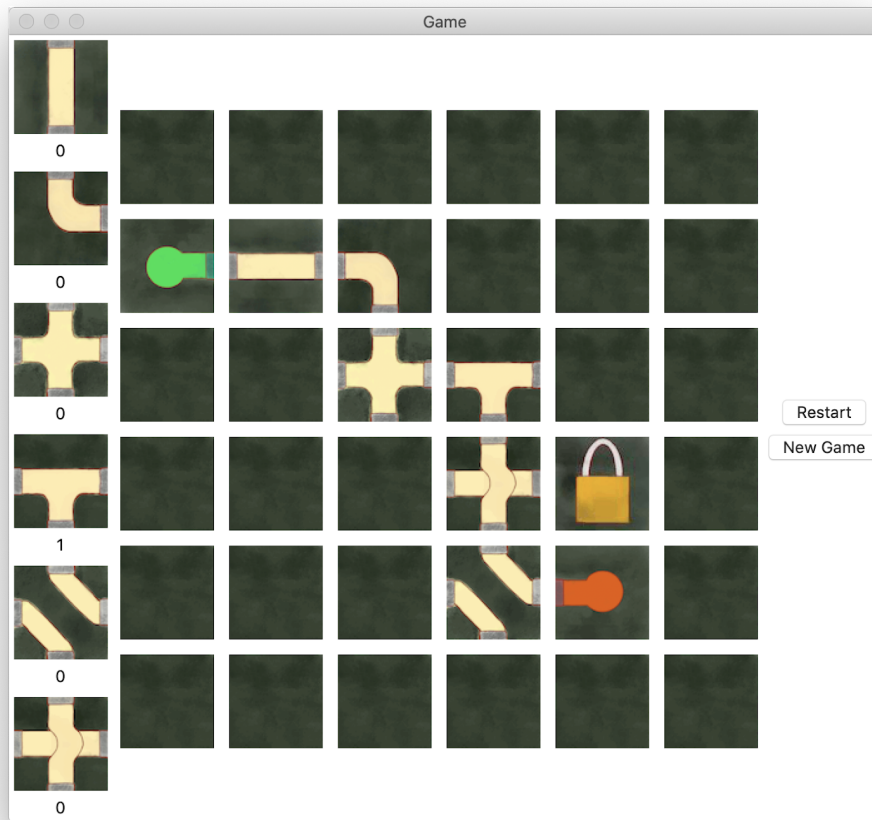


Figure 1. A completed game (pipes all connected)

Getting Started

Files

- file where you write your submission
gui.py - contains the view and part of the controller
game_n.csv - the game board
images/ - a directory of images

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 2. The red blocks represent classes that have already been written for you.

In Figure 2 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). An arrow with an open head indicates that one class is a subclass of another class.

You are encouraged to modify the class structure by adding classes to reduce code duplication. This can include adding your own private helper methods.

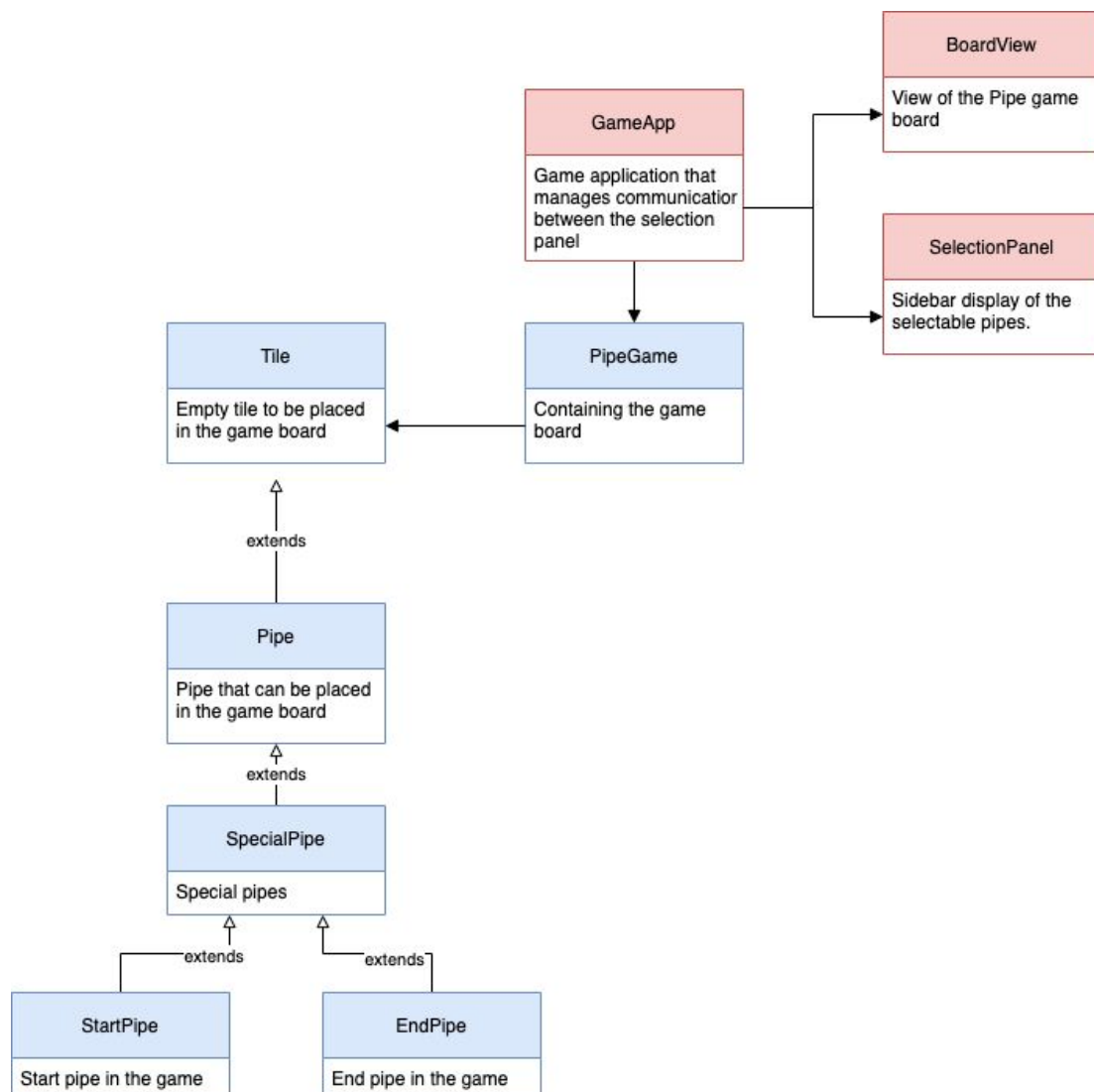


Figure 2. Class structure for the software program

Tiles and Pipes

Tile

A tile represents an available space in the game board. Each tile has a name, and can be unlocked/selectable (available to have pipes placed on them) or locked/unselectable (cannot have pipes placed on them). By default, tiles should be selectable. Tiles should be constructed with `Tile(name, selectable=True)`. Tiles should have the following methods defined:

`get_name(self) -> str`: Returns the name of the tile.

`get_id(self) -> str`: Returns the id of the tile class.

`set_select(self, select: bool)`: Sets the status of the select switch to True

`can_select(self) -> bool`: Returns True if the tile is selectable, or False if the tile is not selectable.

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

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

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

```
>>> tile = Tile('#')
>>> tile.get_name()
'#'
>>> tile.get_id()
'tile'
>>> tile.can_select()
True
>>> str(tile)
"Tile('#', True)"
>>> repr(tile)
"Tile('#', True)"
>>> locked = Tile('locked', False)
```

```
>>> locked.get_name()
'locked'
>>> locked.get_id()
'tile'
>>> locked.can_select()
False
>>> str(locked)
"Tile('locked', False)"
```

Pipe

A pipe represents a pipe in the game. Pipes are a special type of Tile, which can be connected to other pipes in the game board to form a path. Pipes should be selectable unless they are loaded in as part of the game board, e.g. from a file or as part of the initial hard coded game. Each pipe has a name, which defines its pipe type (see Appendix B for a list of pipe names and their corresponding types), and an orientation. A player can rotate a pipe **that they placed** by left-clicking on that pipe in the game board. If the pipe was already on the board when the game was set up or loaded, the pipe cannot be rotated. Each pipe has 4 orientations; 0, 1, 2 and 3:

0	Standard orientation
1	Rotated clockwise once from standard orientation
2	Rotated clockwise twice from standard orientation
3	Rotated clockwise thrice from standard orientation

Rotating a pipe that has the orientation of 3 will reset it back to the standard orientation of 0.

Furthermore, each pipe type has a unique connection profile. For example:

- 1) A straight pipe with the orientation of 0 should have a connection between the North and the South sides.
- 2) An over-under pipe with the orientation of 0 should have a connection between the North and South sides, and an additional connection between the East and West sides.

When a pipe is rotated, the connection profile must be updated accordingly. The representation of the connection profile is at your discretion. Pipes should be constructed with `Pipe(name, orientation=0, selectable=True)`. All pipes must have the following methods defined:

`get_connected(self, side: str) -> list<str>`: Returns a list of all sides that are connected to the given side. i.e. return a list containing some combination of 'N', 'S', 'E', 'W' or an empty list.

`rotate(self, direction: int)`: Rotates the pipe one turn. A positive direction implies clockwise rotation, and a negative direction implies counter-clockwise rotation and 0 means no rotation.

`get_orientation(self) -> int`: Returns the orientation of the pipe (orientation must be in the range [0, 3]).

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

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

Examples

```
>>> pipe = Pipe('straight')
>>> pipe.get_name()
'straight'
>>> pipe.get_id()
'pipe'
>>> str(pipe)
"Pipe('straight', 0)"
>>> repr(pipe)
"Pipe('straight', 0)"
>>> pipe.get_connected('N')
['S']
>>> pipe.get_connected('E')
[]
>>> pipe.get_connected('S')
['N']
>>> pipe.rotate(1)
>>> pipe.get_connected('E')
['W']
>>> pipe.get_orientation()
1
>>> pipe.rotate(1)
>>> pipe.rotate(1)
>>> pipe.rotate(1)
>>> pipe.get_orientation()
0
>>> new_pipe = Pipe("straight", 1)
>>> new_pipe.get_connected("E")
['W']
>>> new_pipe.get_id()
'pipe'
```

SpecialPipe

SpecialPipe is an abstract class used to represent the start and end pipes in the game (see Appendix A). Neither the start nor end pipe should be selectable, and the orientations should be fixed. The id should be set to "special_pipe".

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

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

StartPipe

A StartPipe represents the start pipe in the game. In addition to the special pipe restrictions listed above, the start pipe should implement its `get_connected` method slightly differently to a regular pipe:

`get_connected(self, side=None) -> list<str>`: Returns the direction that the start pipe is facing.

Examples

```
>>> start = StartPipe()
>>> str(start)
"StartPipe(0) "
>>> start.get_id()
'special_pipe'
>>> repr(start)
"StartPipe(0) "
>>> start.get_orientation()
0
>>> start.get_connected()
['N']
>>> new_start = StartPipe(1)
>>> new_start.get_connected()
['E']
```

EndPipe

An EndPipe represents the end pipe in the game. In addition to the special pipe restrictions listed above, the **end** pipe should implement its `get_connected` method slightly differently to a regular pipe:

`get_connected(self, side=None) -> list<str>`: Returns the direction that the end pipe is facing.

Examples

```
>>> end = EndPipe()
>>> str(end)
"EndPipe(0) "
>>> repr(end)
"EndPipe(0) "
>>> end.get_id()
'special_pipe'
>>> end.get_connected()
['S']
>>> new_end = EndPipe(1)
>>> new_end.get_connected()
['W']
```

PipeGame

PipeGame represents the overall information associated with a game of *Pipe*. You will need to implement the following methods in the PipeGame class. This will also require you to add to existing methods in the class.

`get_board_layout(self) -> (list<list<Tile, ...>>)`: Returns a list of lists, i.e. a list where each element is a list representation of the row (each row list contains the Tile instances for each column in that row). (See first example below).

`get_playable_pipes(self) -> (dict<str:int>)`: Returns a dictionary of all the playable pipes (the pipe types) and number of times each pipe can be played.

`change_playable_amount(self, pipe_name: str, number: int)`: Add the quantity of playable pipes of type specified by `pipe_name` to `number` (in the selection panel).

`get_pipe(self, position) -> (Pipe|Tile)`: Returns the Pipe at the position or the tile if there is no pipe at that position.

`set_pipe(self, pipe: Pipe, position: tuple<int, int>)`: Place the specified pipe at the given position (row, col) in the game board. The number of available pipes of the relevant type should also be updated.

`pipe_in_position(self, position) -> Pipe`: Returns the pipe in the given position (row, col) of the game board if there is a Pipe in the given position. Returns None if the position given is None or if the object in the given position is not a Pipe.

`remove_pipe(self, position: tuple<int, int>)`: Removes the pipe at the given position from the board (Hint: create an empty tile at the given (row, col) position and increase the playable number of the given pipe).

`position_in_direction(self, direction, position) -> tuple<str, tuple<int, int>>`: Returns the direction and position (row, col) in the given direction from the given position, if the resulting position is within the game grid, i.e. valid. Returns None if the resulting position would be invalid.

`end_pipe_positions(self)`: Find and save the start and end pipe positions from the game board. This should be called in the `__init__()` method so the start and end pipe positions could be found when the `PipeGame` class is first constructed.

`get_starting_position(self) -> (tuple<int, int>)`: Returns the (row, col) position of the start pipe.

`get_ending_position(self) -> (tuple<int, int>)`: Returns the (row, col) position of the end pipe.

Examples:

```
>>> game = PipeGame()
>>> game.get_board_layout()
[[Tile('tile', True), Tile('tile', True), Tile('tile', True),
Tile('tile', True), Tile('tile', True), Tile('tile', True)],
[StartPipe(1), Tile('tile', True), Tile('tile', True), Tile('tile',
True), Tile('tile', True), Tile('tile', True)], [Tile('tile', True),
Tile('tile', True), Tile('tile', True), Pipe('junction-t', 0),
Tile('tile', True), Tile('tile', True)], [Tile('tile', True),
Tile('tile', True), Tile('tile', True), Tile('tile', True),
Tile('locked', False), Tile('tile', True)], [Tile('tile', True),
Tile('tile', True), Tile('tile', True), Tile('tile', True),
EndPipe(3), Tile('tile', True)], [Tile('tile', True), Tile('tile',
True), Tile('tile', True), Tile('tile', True), Tile('tile', True),
Tile('tile', True)]]
>>> game.get_playable_pipes()
{'straight': 1, 'corner': 1, 'cross': 1, 'junction-t': 1,
'diagonals': 1, 'over-under': 1}
>>> game.change_playable_amount('straight', 2)
```

```

>>> game.get_playable_pipes()
{'straight': 3, 'corner': 1, 'cross': 1, 'junction-t': 1,
 'diagonals': 1, 'over-under': 1}
>>> game.change_playable_amount('straight', -1)
>>> game.get_playable_pipes()
{'straight': 2, 'corner': 1, 'cross': 1, 'junction-t': 1,
 'diagonals': 1, 'over-under': 1}
>>> straight = Pipe('straight')
>>> game.set_pipe(straight, (0,0))
>>> game.get_playable_pipes()
{'straight': 1, 'corner': 1, 'cross': 1, 'junction-t': 1,
 'diagonals': 1, 'over-under': 1}
>>> game.get_board_layout()
[[Pipe('straight', 0), Tile('tile', True), Tile('tile', True),
 Tile('tile', True), Tile('tile', True), Tile('tile', True)],
 [StartPipe(1), Tile('tile', True), Tile('tile', True), Tile('tile',
 True), Tile('tile', True), Tile('tile', True)], [Tile('tile', True),
 Tile('tile', True), Tile('tile', True), Pipe('junction-t', 0),
 Tile('tile', True), Tile('tile', True)], [Tile('tile', True),
 Tile('tile', True), Tile('tile', True), Tile('tile', True),
 Tile('locked', False), Tile('tile', True)], [Tile('tile', True),
 Tile('tile', True), Tile('tile', True), Tile('tile', True),
 EndPipe(3), Tile('tile', True)], [Tile('tile', True), Tile('tile',
 True), Tile('tile', True), Tile('tile', True), Tile('tile', True),
 Tile('tile', True)]]
>>> game.remove_pipe((0,0))
>>> game.get_board_layout()
[[Tile('tile', True), Tile('tile', True), Tile('tile', True),
 Tile('tile', True), Tile('tile', True), Tile('tile', True)],
 [StartPipe(1), Tile('tile', True), Tile('tile', True), Tile('tile',
 True), Tile('tile', True), Tile('tile', True)], [Tile('tile', True),
 Tile('tile', True), Tile('tile', True), Pipe('junction-t', 0),
 Tile('tile', True), Tile('tile', True)], [Tile('tile', True),
 Tile('tile', True), Tile('tile', True), Tile('tile', True),
 Tile('locked', False), Tile('tile', True)], [Tile('tile', True),
 Tile('tile', True), Tile('tile', True), Tile('tile', True),
 EndPipe(3), Tile('tile', True)], [Tile('tile', True), Tile('tile',
 True), Tile('tile', True), Tile('tile', True), Tile('tile', True),
 Tile('tile', True)]]
>>> game.get_playable_pipes()
{'straight': 2, 'corner': 1, 'cross': 1, 'junction-t': 1,
 'diagonals': 1, 'over-under': 1}
>>> game.position_in_direction('E', (0,0))

```

```

('W', (0, 1))
>>> game.position_in_direction('E', (0,1))
('W', (0, 2))
>>> game.position_in_direction('N', (0,0))
>>> print(game.position_in_direction('N', (0,0)))
None
>>> game.pipe_in_position((0,0))
>>> print(game.pipe_in_position((0,0)))
None
>>> game.set_pipe(straight, (0,0))
>>> game.pipe_in_position((0,0))
Pipe('straight', 0)
>>> game.set_pipe(straight, (0,1))
>>> game.get_playable_pipes()
{'straight': 0, 'corner': 1, 'cross': 1, 'junction-t': 1,
'diagonals': 1, 'over-under': 1}
>>> game.get_board_layout()
[[Pipe('straight', 0), Pipe('straight', 0), Tile('tile', True),
Tile('tile', True), Tile('tile', True), Tile('tile', True)],
[StartPipe(1), Tile('tile', True), Tile('tile', True), Tile('tile',
True), Tile('tile', True), Tile('tile', True)], [Tile('tile', True),
Tile('tile', True), Tile('tile', True), Pipe('junction-t', 0),
Tile('tile', True), Tile('tile', True)], [Tile('tile', True),
Tile('tile', True), Tile('tile', True), Tile('tile', True),
Tile('locked', False), Tile('tile', True)], [Tile('tile', True),
Tile('tile', True), Tile('tile', True), Tile('tile', True),
EndPipe(3), Tile('tile', True)], [Tile('tile', True), Tile('tile',
True), Tile('tile', True), Tile('tile', True), Tile('tile', True),
Tile('tile', True)]]
>>> game.get_starting_position()
(1, 0)
>>> game.get_ending_position()
(4, 4)

```

load_file(self, filename) - *(Bonus task)*

This task is worth bonus marks; it will enable you to get a maximum of 2 additional marks, which will be added to your score out of 15. Your score, however, will be capped at 15 out of 15.

Currently, the initial game setup (e.g. board_layout and playable_pipes) is hardcoded at the beginning of the constructor for Game. Implement the following function to allow board_layout

and playable_pipes to be set according to a .csv file (you can assume the file and its formatting is valid), and integrate this functionality as a method in the PipeGame class:

```
load_file(self, filename: str) -> tuple<dict,<str,int>,  
list<list<Tile>>> : Returns appropriate values for playable_pipes and  
board_layout respectively, according to the file with the given filename.
```

Game file protocol

The game file should be in a .csv (comma-separated values) format. The file should contain the representation of the game grid (which must be a square), followed by the number of playable pipes available for the given game. Please refer to SPECIAL_TILES and PIPES for the naming convention of all the different pipes.

Example 1

```
##,  
S,##,  
##,E  
1,2,3,4,5,6
```

Example 1 should place:

- Starting pipe in position (1, 0);
- Ending pipe in position (2, 2); and

The player should have access to 1 straight, 2 corner, 3 cross, 4 junction-t, 5 diagonals and 6 over-under playable pipes.

Example 2

```
###,##,  
S,##,L,##,  
##,ST1,##,  
###,##,E3  
1,1,1,1,1,1
```

Example 1 should place:

- Starting pipe in position (1, 0);
- Ending pipe in position (3, 3) with orientation of 3;
- Locked tile in position (1, 2); and
- Straight pipe in position (2, 2) with orientation of 1.

The player should have access to 1 of each playable pipe.

Marking

Item	Marks
Tile	1
Pipe	2
LockedTile	1
SpecialPipe	1
StartPipe	0.5
EndPipe	0.5
PipeGame	6
Bonus marks	
load_file	2

Functionality Assessment

The functionality will be marked out of 12. 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 in that category. If, say, there are 5 functionality tests in the 'Tile' category and you pass 4 of them, then your functionality mark would be $20/25 * \text{Marks allocated for the 'Tile' category}$. 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

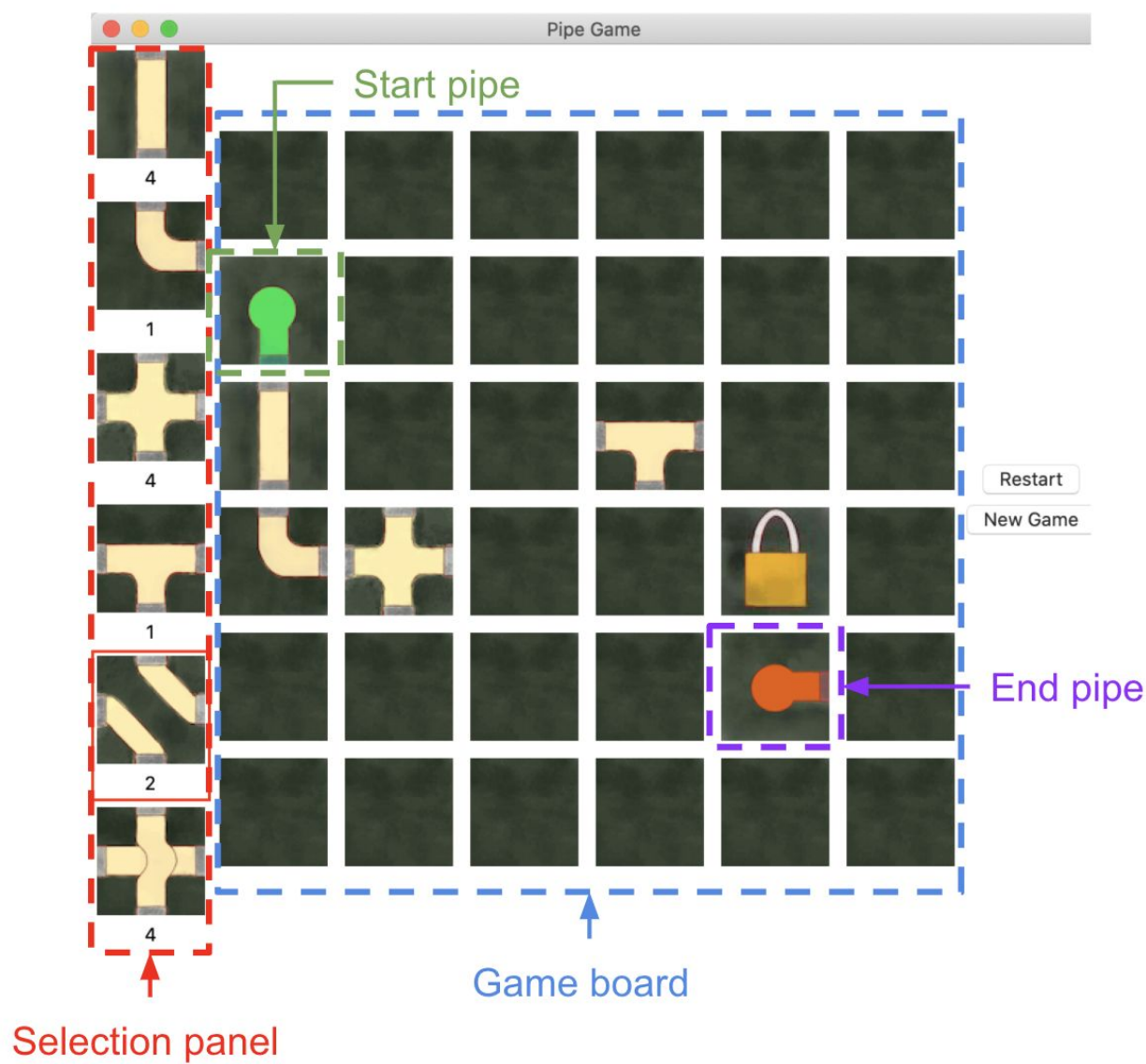
The style of your assignment will be assessed by one of the tutors, and you will be marked according to the style rubric provided with the assignment. The style mark will be out of

Assignment Submission

Your assignment must be submitted via the Assignment 1 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 a1.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.

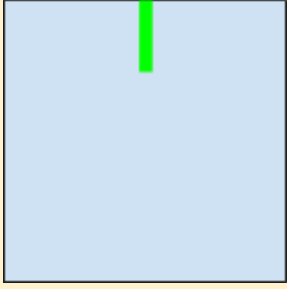
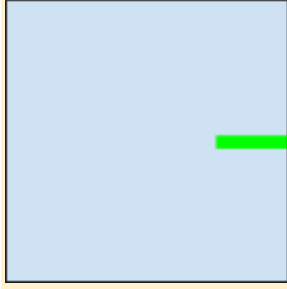
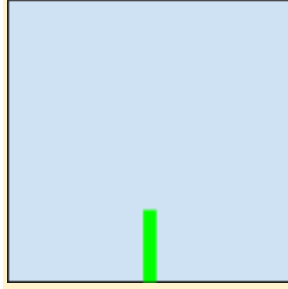
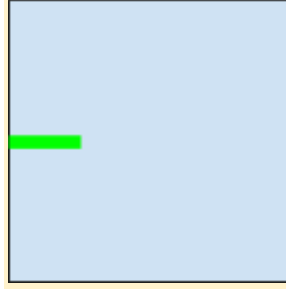
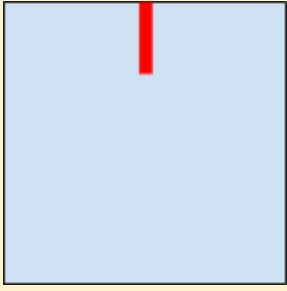
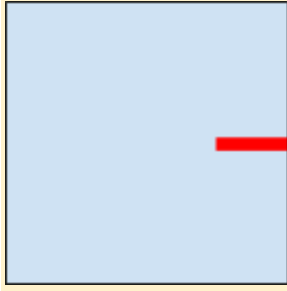
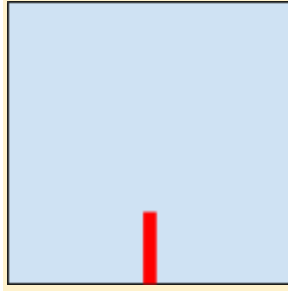
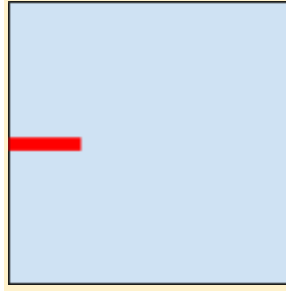
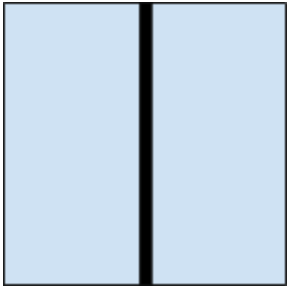
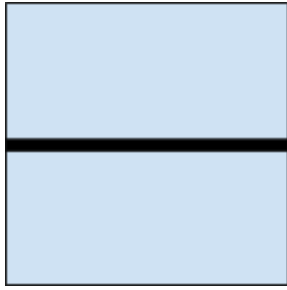
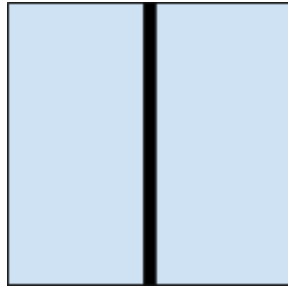
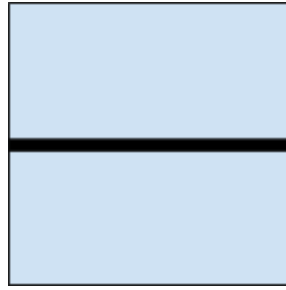
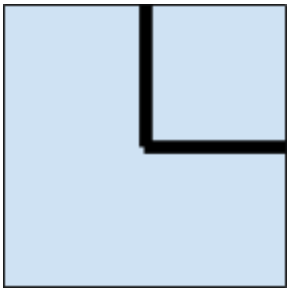
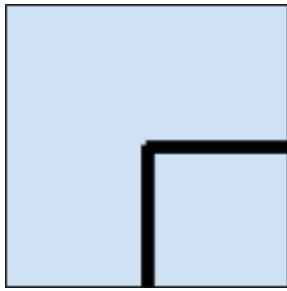
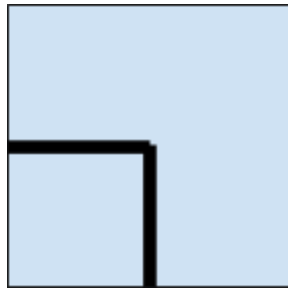
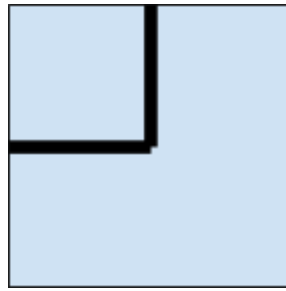
Appendices

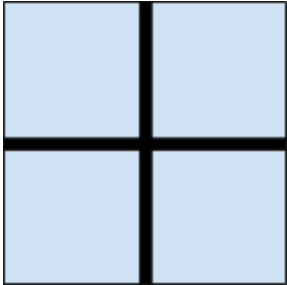
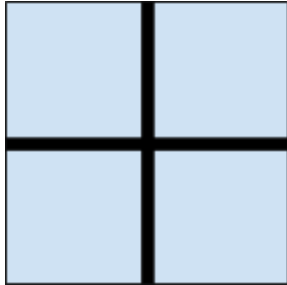
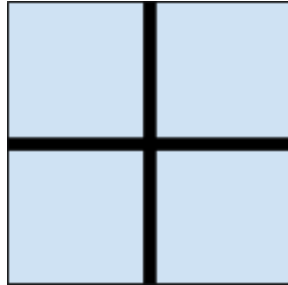
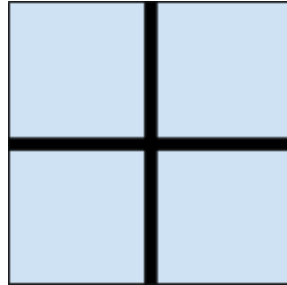
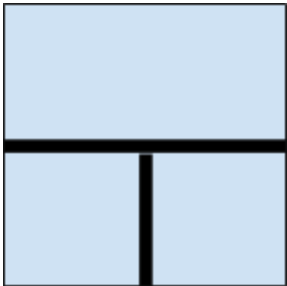
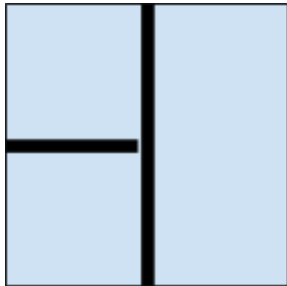
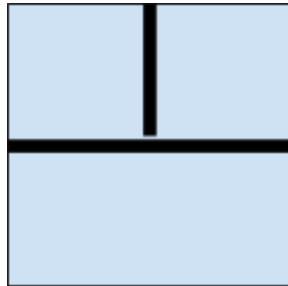
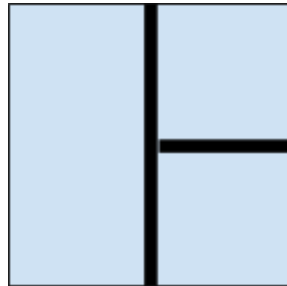
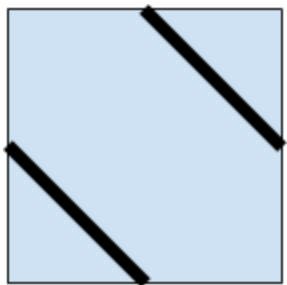
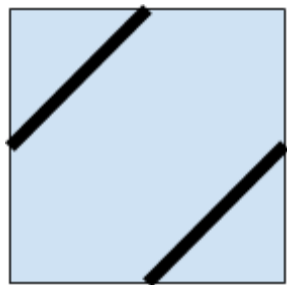
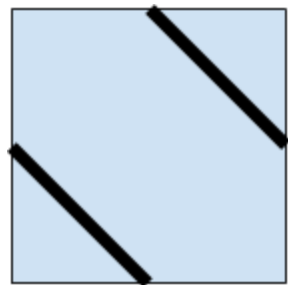
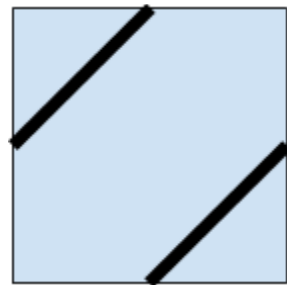
Appendix A



Appendix B

Orientation 0	Orientation 1	Orientation 2	Orientation 3
start			

			
end			
			
straight			
			
corner			
			
cross			

			
t-junction			
			
diagonals			
			
over-under			
