CSSE1001/7030

Semester 2, 2016
Assignment 3
Version 1.0.0
25 marks/28 marks

Due Friday 28 October, 2016, 21:30

1 Introduction

For this assignment, you will be building a tile-matching game. A core mechanism of the game is the tile grid, in which the player must swap tiles in order to make runs of connected tiles. The bigger the run, the more points score.

2 Assignment Tasks

This assignment is broken down into three tasks, each of which are successively more challenging. It is recommended to proceed to the next task only when you are reasonably confident you have a fully or mostly working solution.

CSSE1001 students will be marked out of 25, and CSSE7030 students will be marked out of 28. See Table 1 for a more precise breakdown.

2.1 Download files

The first task is to download `a3.py` and `a3_files.zip`. The file `a3.py` is for your assignment. **Do not modify the assignment file outside the area provided for you to write your code**.

2.2 Important Concepts

You have been provided with copious amounts of code in the support file, `a3_support.py`. It is highly recommended that you review it before writing your code, as it contains many useful constants, functions, & classes. You needn't understand how all of the code works (i.e. you can trust that it works), but you do need to understand how to interact with some of it.

Table 1: Task/Mark Breakdown

	Sub-Task	**Marks**
Task 1 *Basic features*		**10 marks**
	SimplePlayer	2
	SimpleStatusBar	3
	GUI	5
Task 2 *Intermediate features*		**7 marks**
	Character, Player, Enemy	1.5
	VersusStatusBar	1.5
	ImageTileGridView	1
	GUI	3
Task 3 *Advanced features*	*open ended features*	**8 marks**
Postgraduate *CSSE7030 only*	*animating*	**3 marks for CSSE7030; 0 marks for CSSE1001**

2.2.1 Event Listeners

The `TileGrid` & `SimpleGame` classes, which are provided in the support file, follow a pattern which allows you to assign callbacks (i.e. a function, method, lambda expression) to be called when certain events take place. This is called **Event Listening**. (The file `ee.py` provides this functionality). This pattern provides a simple mechanism for classes to notify other classes of significant changes (such as when the a run is formed in the grid). This is similar to how code is executed when a tkinter button click is pressed.

The code that listens to the required events has been provided for task 1 in `a3.py`. For task 2, however, in order to implement the required functionality, you will need to listen to the `swap_resolution` and `runs` events that the `SampleGame` class emits. Task 3 students are encouraged to consider emitting their own events (this may be useful for task 2, but is not necessary).

2.2.2 TileGridView

The `TileGridView` class is a tkinter compatible widget that facilitates display and user interaction with a tile grid (`TileGrid`). The code required to create it is provided in `a3.py`.

The user is able to select a tile, which will increase its size. The user can either deselect that same tile, or select another tile. If they select another tile, the grid becomes disabled and the tiles swap

(`SimpleGame` emits `swap` at this point). The `TileGrid` then finds and removes any runs on the grid (`SimpleGame` emits `score` & `runs` at this point). New tiles are generated, and they fall into the grid, replacing the removed ones. This process is repeated, since more runs may be created by dropping new tiles, until no more runs can be found (`SimpleGame` emits `swap_resolution` at this point). The grid is then re-enabled so that the user can select tiles.

2.3 Write the code

There are several classes you need to write and these are described below. **Do not use global variables in your code.** Each class and method that you write must have a suitable docstring comment, as specified in the course notes. See `http://csse1001.uqcloud.net/notes/commenting`

3 Task 1 - Basic GUI

The first task is to write a basic GUI. The `SimplePlayer` class will represent the player and will keep track of their status, which will be displayed by the `StatusBar`.

3.1 SimplePlayer Class

Firstly, the `SimplePlayer` class must be implemented along with the following methods (you may add more if you wish):

__init__(self): Constructs the player.

add_score(self, score): Adds a score to the player's score. Returns the player's new score.

get_score(self): Returns the player's score.

reset_score(self): Resets the player's score.

record_swap(self): Records a swap for the player. Returns the player's new swap count.

get_swaps(self): Returns the player's swap count.

reset_swaps(self): Resets the player's swap count.

```
>>> player = SimplePlayer()
>>> player.add_score(9001)
9001
>>> player.get_score()
```

```
9001
>>> for i in range(10): player.record_swap()
>>> player.get_swaps()
10
>>> player.reset_swaps()
>>> player.reset_score()
>>> player.get_score()
0
>>> player.get_swaps()
0
```

3.2 SimpleStatusBar Class

The `StatusBar` must inherit from `tk.Frame` and display the following:

- The number of swaps that the player has made.
- The player's score.

3.3 SimpleTileApp Class

The `SimpleTileApp` is responsible for the top-level GUI. A stub of this class has been provided.

The GUI must have the following features:

- A `TileGridView` to play the game.
- A `StatusBar` to display the player's score and how many swaps they've made.
- A Reset Status button to reset the player's status and update the StatusBar - this does not reset the tiles.
- A `File` menu, with the following items:
  - `New Game`: Begins a new game (i.e. resets the tiles), including resetting the `StatusBar`.
  - `Exit`: Closes the game.
- The title of the window must be set to something sensible, such as "Simple Tile Game".

If the player attempts to reset their status or start a new game while the grid view is resolving (see `TileGridView.is_resolving`), the GUI must show a message box informing the player of such.

For starting a new game, see `SimpleGame.generate` and `TileGridView.draw`.

4 Task 2 - Extended Game

This task is based around implementing more features to extend the game.

The extended game introduces the concept of the player fighting an enemy. After making a swap, the player will attack the enemy based upon the quality of the run(s) formed (i.e. the more connected tiles, the better). After a set number of swaps, the player's turn will end, and the enemy will attack the player. This repeats until either the player or enemy dies. If the player dies, they game is over. If the enemy dies, the player advances to the next level (to fight a different, more challenging enemy).

4.1 Character Class

The `Character` class implements the basic functionality of a player/enemy within the game, and is the superclass of `SimplePlayer` & `Enemy`. It implements at least the following methods (you may add more if you wish):

__init__(self, max_health): Constructs the character, where max_health is an integer representing the maximum amount of the player can have, and the amount of health they start with.

get_max_health(self): Returns the maximum health the player can have.

get_health(self): Returns the player's current health.

lose_health(self, amount): Decreases the player's health by amount. Health cannot go below zero.

gain_health(self, amount): Increases the player's health by amount. Health cannot go above maximum health.

reset_health(self): Resets the player's health to the maximum.

4.2 Enemy Class

The `Enemy` class represents an enemy, and inherits from `Character`. It implements at least the following methods (you may add more if you wish):

__init__(self, type, max_health, attack): Constructs the player, where type is the enemy's type, max_health is an integer representing the amount of health the enemy has, and attack is a pair of the enemy's attack range, (minimum, maximum).

get_type(self): Returns the enemy's type.

attack(self): Returns a random integer in the enemy's attack range.

```
>>> enemy = Enemy('fire', 200, (80, 125))
>>> enemy.get_type()
'fire'
>>> for i in range(10): print(enemy.attack()) # output will likely differ
81
94
98
89
86
121
```

4.3 Player Class

The `Player` class represents a human player, and inherits from `Character`. It implements at least the following methods (you may add more if you wish):

__init__(self, max_health, swaps_per_turn): Constructs the player, where max_health is an integer representing the amount of health the player has, and swaps_per_turn is an integer representing the number of swaps a player makes each turn.

record_swap(self): Decreases the player's swap count by 1, which cannot go below zero. Returns the player's new swap count.

get_swaps(self): Returns the player's swap count.

reset_swaps(self): Resets the player's swap count to the maximum swap count.

attack(self, runs): Takes a list of Run instances. Returns a pair of (type, damage), where attack is a list of pairs of the form (tile, damage), listing damage amounts for each type, in the order the attacks should be performed. For example, if the result were a 150 damage fire attack and then a 90 damage water attack, this method would return the following:

```
[('fire', 150), ('water', 90)]
```

You may implement any algorithm you wish to calculate attack damage. If there are multiple runs of the same type, the algorithm should give a bonus (more than simply adding the individual damages together), and this bonus should increase as 3 or more runs of the same type are formed. The algorithm used in the example code is $damage = tiles\_in\_run \times longest\_straight\_in\_run \times player\_attack$. (See `Run.__len__ & Run.get_max_dimension`). More interesting algorithms will attract slight bonus marks (i.e. making each type super effective and/or not very effective against other types).

```
>>> player = Player(200, 5, 10)
>>> player.get_swaps()
5
>>> for i in range(6): player.record_swap()
4
3
2
1
0
0
>>> fires = {(5,4): Tile('fire'), (4,4): Tile('fire'), (3,4): Tile('fire')}
>>> waters = {(0,1): Tile('water'), (0,2): Tile('water'), (1,1): Tile('water'),
(0,3): Tile('water'), (1,3): Tile('water')}
>>> runs = [Run(fires), Run(waters)]
>>> player.attack(runs, 'coin')
{'water': 150, 'fire': 90}
```

4.4 VersusStatusBar Class

The `VersusStatusBar` is responsible for displaying the game/player/enemy's status. It must inherit from `tk.Frame` - (it is also permissible to inherit from `SimpleStatusBar`).

It must display the following:

- The current level, starting from 1 (in the centre of the first row).
- The player's health (on the left of the second row).
- The enemy's health (on the right of the second row).
- The number of swaps the player has made (in the centre of the second row).

Regarding the display of health, partial marks will be awarded for displaying the numerical value. Full marks will be awarded for also displaying a rectangular bar whose width changes according to the percentage of health remaining.

4.5 ImageTileGridView

The `ImageTileGridView` class inherits from `TileGridView` and display each tile using images instead of rectangle. You may use any image set you wish (including one of your own creation), provided it is appropriate and differs reasonably from a solid coloured rectangle (i.e. different shapes, icons, etc.). See `TileGridView.draw_tile_sprite`.

You may find it helpful to search the internet for "sprite sets".

4.6 SinglePlayerTileApp

The `SinglePlayerTileApp` class is responsible for displaying the top-level GUI. The `SimpleTileApp` class from task 1 must not be altered for task 2. Instead, create a class `SinglePlayerTileApp` that inherits from `SimpleTileApp` to reuse any common functionality.

The GUI must have the following features:

- A `ImageTileGridView` to play the game.
- A `VersusStatusBar` to display the game status, above the grid.
- A graphic to display the player (to the left of the grid)
- A graphic to display the enemy (to the right of the grid).
- A `File` menu, with the following items:
  - `New Game`: Begins a new game at level 1.
  - `Exit`: Closes the game.
- The title of the window must be set to something sensible, including the current level, such as "Tile Game - Level 1".

You may find `generate_enemy_stats` from the support file helpful in making your enemies stronger as the levels progress, but you are not required to use it.

Regarding the display of the player & enemy, partial marks will be awarded for using canvas shapes; full marks will be awarded for using custom images (see note in 4.5 on sprite sets).

If the player attempts to start a new game while they are playing, the GUI must show a dialog box to the user asking them to confirm. You may assume the player will not attempt to start a new game while the grid is resolving.

5 Task 3

This task is open ended. It's entirely up to you to decide what to do for this task. Marks will be awarded based on the sophistication of the chosen additions, the style of coding, and the supporting documentation. Be sure to discuss your intentions with course staff **before you start coding your extensions**, in order to receive feedback and ensure that your extensions will be sufficiently sophisticated. For inspiration you might look at tile-matching style games, e.g. `Bejeweled`, `Spellfall`, `Gyromancer`, etc.

You must include a description of your task 3 features in your submission, in a file called `a3_task3.pdf`. Ensure that you clearly draw attention to each feature using a relevant heading such that the reader can get a quick overview of all of your features in a few seconds.

Note: Your code for task 3 must still comply with the specific requirements for each of task 1 & 2. You are encouraged to reuse functionality by inheriting from these classes and overriding methods.

5.0.1 Suggestions

- Powerups, items, weapons, etc.
- Special tiles.
- Different game types, loading/saving to file.
- Animated characters, attacks, etc.
- Multiplayer (local or network).

6 Postgraduate Task - CSSE7030 only

This task is only for CSSE7030 students. CSSE1001 students may attempt it, but will be awarded no marks.

CSSE7030 students are required to make the following modifications:

Task 1: A performance bar must be added to the `StatusBar`, which displays the player's performance factor, $pf = \frac{score}{swaps+1} \times PERFORMANCE\_SCALE$

The width of the performance bar rectangle must be set according to the following:

$$width = \begin{cases} \frac{pf}{2} & \text{if } pf \leq 1 \\ 1 - \frac{1}{2 \times pf} & \text{if } pf \geq 1 \end{cases}$$

Further, this bar must be animated smoothly after swapping/resetting; it must not have its width changed immediately. In the same vein, the score field in the status bar must quickly (but not immediately) tick up/down progressively when the score changes.

Task 2: The health bars in `VersusStatusBar` must be animated in the same way as for task 1.

7 Assessment and Marking Criteria

Marks will be awarded based on a combination of the correctness of your code and on your understanding of the code that you have written. **A technically correct solution will not elicit a pass mark unless you can demonstrate that you understand its operation.**

A partial solution will be marked. If your partial solution causes problems in the Python interpreter please comment out that code and we will mark that.

8 Assignment Submission

You must submit your file(s) electronically through Blackboard. You may submit multiple times before the deadline, but only the last submission will be marked.

You must submit a single zip archive, called `a3.zip`, containing all the files required to run your assignment, including:

- `a3.py`
- `a3_task3.pdf` *task 3 only*
- Any images/sprites used *task 2/3 only*

Late submission of the assignment will not be accepted. In the event of exceptional circumstances, you may submit a request for an extension.

All requests for extension must be submitted on the UQ Application for Extension of Progressive Assessment form:
`http://www.uq.edu.au/myadvisor/forms/exams/progressive-assessment-extension.pdf`
by 48 hours prior to the submission deadline. The application and supporting documentation must be submitted to the ITEE Coursework Studies office (78-425) or by email (`enquiries@itee.uq.edu.au`).

Changelog

Any subsequent modifications to this document will be listed here.