CSC148 Summer 2018: Assignment 1

Due: Sunday, June 17th @ 11PM

Overview

In this assignment, you'll be implementing a Battle Game. This game consists of two types of characters which can perform various attacks against each other. In this assignment, you'll be implementing various classes and designing their functionality (in order to fit the requirements of the client code), getting practice with inheritance and composition, as well as implementing a type of queue.

This assignment contains lots of small parts for you to implement, and may require some planning. It's recommended that you organize your code on paper first (write out what classes you'll need, what attributes they should have, what methods, their relationship to other classes) before you start coding in order to save yourself from re-writing and re-organizing your code.

This document is fairly verbose, as we've decided to explain parts of the client code and classes provided in some detail. Most of this documentation is also present in the starter code, of which a description is provided below. Please take your time to read through it!

You will still have to read the client code to understand the names of methods that we expect.

At the end of this document, the grading scheme for this assignment has also been provided to you. There should be no surprises when the assignments are graded, as you should be able to estimate 60% of your final grade prior to submitting.

Plagiarism

As with all exercises and assignments: there is no tolerance for plagiarism. We do have tools to detect plagiarism, and it does so extremely well. This assignment is also being used for the first time, so searching online for a solution won't be too helpful. Remember: If you can search for a solution online, so can we.

Absolutely do not post any of your assignment code or show it to anyone aside from the CSC148 course staff! If someone plagiarises your work, you're both at risk.

A typical penalty for a first offence is a zero on the assignment. The case will also be entered into the UofT academic offence database. (If you get caught a second time ever as an undergrad, the penalties are much, much more severe.)

Our tips to avoiding plagiarism:

- Remember that figuring out what code needs to be written is the most important part of the assignment. Typing out code based on steps that someone else gave you is considered cheating, as you are submitting someone else's ideas as your own.
- Don't search the web for solutions. We've already done that and will be comparing what we found with what you submit. It doesn't help to change variable names and move functions around. Our software will still find you.

- Don't ask your friend for their solution, even if you just want it to "see how to solve it". Don't show your friends your solution either.
- Don't get solutions, or partial solutions, from a tutor. They frequently provide the same code or ideas to more than one person. We know because we frequently catch people who do it.
- Only ask for detailed help on your code from official CSC148 course staff, including the Help Centre.
- If you can't figure out how to write a function, it's often because you haven't fully understood an underlying concept. Review the materials on that topic, experiment in the Python shell, and ask us for help!
- Remember that it is better to submit an assignment with a few missing functions than to cheat!

Please also see the Collaboration Policy on the Assignment page.

Your Task

For this assignment, you will need the pygame library.

If you don't have it: You can download this through Pycharm (similar to how we installed python-ta), or by opening terminal/cmd.exe (depending on what system you're using) and running the command:

```
py -m pip install pygame
```

Download a1.zip and extract the files in it. This contains all of the starter code and files used by the starter code:

- **a1_game.py**: The file with client code for you to read and understand. Fill in the dictionaries at the top of the file, putting in the class names as needed.
    - The `perform_attack()` function gets called whenever an action is to be made. It tries to decide on a move to make by using the next player's playstyle and selecting an attack. It then tries to make the next player in the battle_queue perform that action, removing the player from the queue if it can. This is called whenever a key is pressed if the next player in the `battle_queue` is using the `Manual` playstyle, or every few seconds if the next player in the `battle_queue` is using a non-Manual playstyle.
    - The `set_up_game()` function is called before a game is created. It sets up the parameters of which characters to use, what names to give them, which playstyle to use, and which player acts first.
    - The `update_ui()` function is called repeatedly (every 100 milliseconds) and updates the ui by giving the next sprites to draw and updating the Health Points (HP) and Skill Points (SP) displays. Most of the getters you'll need to write are called here.
- **a1_ui.py**: **When you want to run the game, run a1_ui.py!** You do not need to read or understand any of the code in this file (although you can if you want). a1_ui.py is the

file that actually runs your code and draws it in pygame, making calls to the functions and looking at the variables defined in a1_game.py. We've moved it to another file so that you can focus on the client code as opposed to the details of pygame.

- **a1_rogue_unittest.py**: The file that contains basic test cases for the rogue character you'll be implementing. Passing these tests ensures that our test scripts can run on your code, and will determine a portion of your mark (see Grading Scheme).
- **a1_mage_unittest.py**: The file that contains basic test cases for the mage character you'll be implementing. Passing these tests ensures that our test scripts can run on your code, and will determine a portion of your mark (see Grading Scheme).
- **a1_battle_queue.py**: The file containing your BattleQueue class. The method headers and descriptions have been provided for you already, but not the implementation.
    - Fill in the docstrings, type annotations, implementation, and make sure to test this on your own. No test cases have been provided for the battle queue!
- **a1_playstyle.py**: The file containing various Playstyle classes. The Manual playstyle class has been provided for you, but you are tasked with implementing the Random playstyle, which will pick one of the available skills at random.
- **sprites** (folder): This folder contains the sprites used by a1_ui.py. The sprites all have the name format: [character]_[type]_#.png where [character] is rogue or mage, [type] is idle, attack, or special, and # is a number from 0 - 9. Your get_next_sprite() method for the classes you implement should return a name in that format. For example:
    - rogue_idle_0.png
    - mage_attack_2.png
    - mage_special_9.png
- **a1_pyta.txt**: The python_ta file to use for this assignment.

Description of Classes

Below is a description of the classes you'll encounter in this assignment. Note that you may want to implement additional classes beyond those that are mentioned below, and that it's highly recommended for you to do so.

Characters

In this assignment, there are 2 characters that are available: Rogue and Mage. These are the characters that can be chosen to fight. All characters start with 100 Health Points ("HP"), 100 Skill Points ("SP"), and their respective defense stats (unique to each character). They also have a name that they start with, a playstyle, an "animation state", certain skills that they can use, and an enemy that they target. The BattleQueue that's being used is also passed into them as an attribute for convenience.

There are 2 types of skills which the Manual Playstyle has key mappings for:

- "A": Attacks; these are the basic attacks performed by a character. Details are in the character descriptions.
    - They are called using the `attack()` method.
- "S": Special attacks; these are special skills performed by a character. Details are in the character descriptions. These skills can have additional effects beyond simply causing damage to their enemy.
    - They are called using the `special_attack()` method.

When a character is hit by a skill, their HP is reduced by (Damage - Defense) amount. The character performing a skill loses a certain number of SP based on the skill used.

The "animation state" keeps track of which sprite needs to be drawn next, and is based on the action taken by the character. If the character isn't doing anything, or if an animation has just ended, it should provide the sprites for its idle animation (e.g. `[character]_idle_0`, then `[character]_idle_1`, and so on).

When a character attacks, the next sprite to be drawn should be the corresponding attack frame (e.g., `[character]_attack_0`, then `[character]_attack_1`, and so on until `[character]_attack_9`, after which it goes to `[character]_idle_0`) -- once that ends, it should return to the idle animation.

The same goes for its special attack (e.g., `[character]_special_0`, then `[character]_special_1`, and so on until `[character]_special_9`, after which it goes to `[character]_idle_0`).

In this assignment, we won't be forcing you to use any specific implementation for this. The only thing you need to do is make sure the `get_next_sprite()` method returns the next sprite to be drawn (this should always return `[character]_idle_0` first for a character that has yet to perform any action, but if we call an action before our first call to `get_next_sprite()` such as attacking, we should return the corresponding 0th frame, e.g. `[character]_attack_0`).

For more information about sprites/get_next_sprite():
Go to the end of the document "Appendix A: Sprites"

See the `set_up_game()` function in `a1_game.py` to see how a character is initialized. In addition to its initialization, characters **must** have an `.enemy` attribute which will be set prior to any attacks being called. This is also done in the `set_up_game()` function.

Rogue

A rogue is a character with 10 defense. Their skills are as follows:

- "A" (Attacks): Deals 15 damage and adds the Rogue to the end of the BattleQueue. Takes 3 SP.
- "S" (Special Attacks): Deals 20 damage and adds the Rogue to the BattleQueue twice, so it can attack twice in succession. Takes 10 SP.

Mage

A mage is a character with 8 defense. Their skills are as follows:

- "A" (Attacks): Deals 20 damage and adds the Mage to the end of the BattleQueue. Takes 5 SP.
- "S" (Special Attacks): Deals 40 damage and adds its enemy to the BattleQueue once before adding the Mage itself into the BattleQueue (i.e. the enemy gets to attack an additional time before the Mage is able to attack again). Takes 30 SP.

Other Notes

While only existent in the docstring examples for BattleQueue and in the unittests, your Characters should have a ___repr___ which takes the following form:

```
Name (Character): HP/SP
```

For example, for a Rogue named Sophia with 13 HP and 37 SP, the repr should return

```
Sophia (Rogue): 13/37
```

Playstyle

The Playstyle classes represent ways for us to choose attacks. There are 2 attributes for a Playstyle: one representing whether it's a Manual mode or not, and another representing the battle queue that the Playstyle is used in. When it's called to perform an action, it'll try to decide an action for the first person in that BattleQueue. The Manual Playstyle has been provided for you. It simply takes the key that is pressed and returns that key if it's a proper attack ("A" or "S").

You are tasked with implementing the Random Playstyle, which returns a key for a move that can be used by the player (e.g. if the character has enough SP for only a normal attack, it should return 'A'. If they have enough SP for either a normal or a special attack, it should return either 'A' or 'S') at random.

Docstring examples are not required for Playstyles.

BattleQueue

A BattleQueue is a queue that lets our game know in what order various characters are going to attack. The method headers and descriptions have all been provided for you, but the implementation will depend on you.

In addition to the regular queue methods (`add()`, `remove()`, and `is_empty()`), three additional methods are required:

- `peek()` should return (but not remove) the next person in the BattleQueue who can use a skill (similar to how `remove()` and `is_empty()` ignores characters who can't use skills).

    - You may assume that we will call `is_over()` before calling `peek()`, and that we won't call `peek()` on an empty BattleQueue.

- `is_over()` should check whether the game being carried out in the BattleQueue is over or not. The game is considered over if any of these conditions are true:
    - Both players have no skills that they can use (because they have too little/no SP)
    - One player has 0 HP
    - The BattleQueue is empty (which should only happen if no players have any skills that they can use).
- `get_winner()` should return the `winner` of the game, if there is one. If not, or if the game is not over yet, it should return `None`. The winner of the game is considered to be the one who has more than 0 HP when the game is over if the other player has 0 HP. In all other cases (e.g. the game not being over or both players being unable to use any skills), the winner is `None`.
    - The winner returned should refer to the object, not their name.

Additionally, `remove()` and `is_empty()` should ignore any characters that have no skills at their disposal. I.e.:

- `remove()` should return the next character in the BattleQueue who can use a skill.
    - When testing, we will not try to remove from a BattleQueue that is empty (i.e. if `is_empty()` returns `True`, we won't try to remove from the BattleQueue).
- `is_empty()` should return `True` if there are no characters in the BattleQueue who can use skills.

You may add additional methods to your BattleQueue if you would like, but they won't be tested. The documentation will, however, still be graded.

Submission

Exercises are to be submitted through MarkUs in the a1 folder. You only need to s**ubmit a1_game.py, a1_battle_queue.py, a1_playstyle.py and any files you've written/imported into a1_game.py**. You do not need to submit any of the icons provided, a1_ui.py, or the unittests provided.

To log in to MarkUs, use your UTORid as the log-in name. The password is your teaching labs password. If you have not set this up or have forgotten your password, go to the Teaching Lab's Account Management Page and (re)set your password.

Lateness Penalty

If you are unable to complete course work due to an emergency (e.g. illness, injury, or other serious situations), please e-mail Sophia as soon as possible.

You have 6 grace tokens shared across both assignments, with each of them being worth 2 hours each. Submitting an assignment late will reduce the total number of grace tokens (i.e. 1

minutes late will remove 1 token, 2 hours and 1 minute late will remove 2 tokens in total, etc.). You can distribute these across your assignments however you like. Work submitted beyond the extensions provided by grace tokens (i.e. you ran out) will not be graded unless you have special circumstances.

Grading Scheme

Below is the grading scheme for this assignment. As this is provided, there shouldn't be many surprises when you receive your mark, as you should be able to estimate at least 60% of your grade following the below rubric.

20% Code Style

- **10% PythonTA**: For every PythonTA violation, you'll lose a portion of this grade. Make sure all of the files you submit are PythonTA compliant.
    - This excludes a1_game.py -- that file is not to have PythonTA run on it.
- **10% Documentation:** Documentation includes: type annotations, docstrings (the existence of them, and the quality of them), and docstring examples for all code that you write. Docstring examples are not needed for a1_playstyle.py, and only the type annotations and docstring examples in a1_battle_queue.py will be marked (as we have done the majority of the documentation for you in that file).

80% Code Correctness

Code correctness is based **entirely** on passing test scripts.

- **40% for the Provided Unittest**s: You lose a portion of your mark for each test case you fail (proportional to how many test cases are in the file).
    - 20% for a1_rogue_unittest.py
    - 20% for a1_mage_unittest.py

    Passing the provided unittests does not guarantee that your code is flawless. The tests only provide a basic check for functionality, and ensures that our hidden tests are able to run.
    - These tests also do not include detailed tests for the BattleQueue -- there are part of our hidden tests. You are to test your BattleQueue on your own.
- **40% for Hidden Tests**: The hidden tests will test for correctness of your code in various scenarios. If you do not pass the provided unittests, you will likely also not pass most of the hidden tests.
    - Examples of what the hidden tests may check for include (but are not limited to):
        - Edge cases.

        - Various combinations of (valid) method calls.
        - Various ways of 'finishing' a battle.
        - Whether you're using inheritance or not.
        - Whether you're using inheritance correctly.
        - Functionality of your BattleQueue class.
            - Especially testing your BattleQueue to make sure it ignores characters without enough SP to act.
            - We may forcibly add/remove from a BattleQueue to put things into a certain order.
    - We will **not** test on any invalid input. For example:
        - We will not try to remove from or peek at a BattleQueue if it's empty (unless it's empty when it shouldn't be).
        - We will not try to make perform certain attacks if they're not valid, but we will want check if it's valid or not.
        - We will not pass in anything unexpected: if a method is supposed to take in an int, then we'll pass in an int.
        - We will not have a player's enemy in a different BattleQueue than what the player has. (i.e. assume that a player and their enemy are both in the same BattleQueue.)
        - We will not call is_empty() if is_over() is True because a character has 0 HP.
            - We will, however, call is_empty() on cases where both players don't have enough SP to act.
    - One thing that we may do (and which is done in the unittests provided) is **ignore the BattleQueue all together and just use skills.**
        - We will **not** test the state of the BattleQueue if we do this though, since it doesn't make sense; this is just to check other methods (such as get_next_sprite(), is_valid_actions(), etc.)
    - **We will also not test your the correctness of your Playstyle class.** This is just for you to be able to use your class, and to become familiar with how the Playstyle functions should work, as they'll be important for A2.
        - With that said, your code should still be documented properly, as we will check that. Docstring examples, however, are not required for Playstyle classes.

- If you have any questions about what might be tested, ask Sophia (Piazza is particularly good for these types of questions). A list of clarifications and assumptions you can make will be maintained.

Appendix A: Sprites

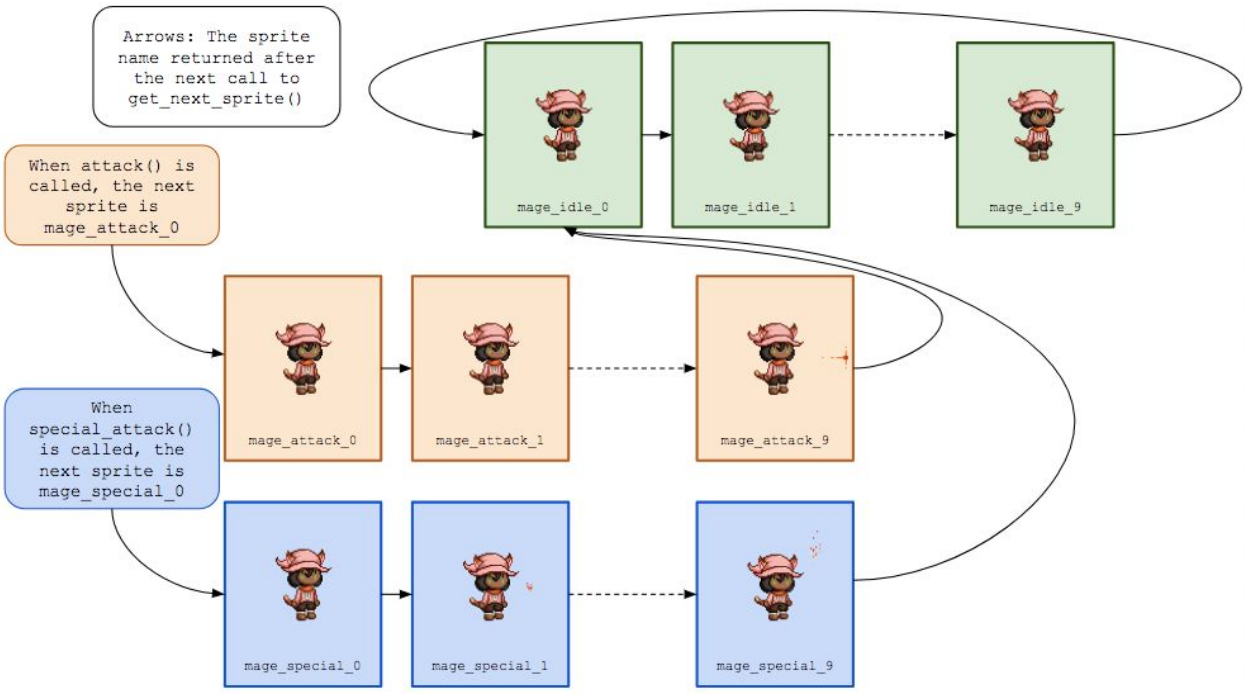
For context: Sprites are images. In a game sense, they're images that float above others (like ghosts or sprites, hence the term, sprites). In the context of our: our characters use sprites which float above the background of the game. So each character has "sprites" and each "sprite" is a frame for an animation.

If you're not familiar with animation frames: imagine you have a bunch of images. You flip through each of them, and when played sequentially, they make an animation. If you go through the images in the sprites folder, you'll be able to run through the sprites and see that, when played sequentially, an animation is formed.

- As an example of what `get_next_sprite()` should return: Let's say you have a new Mage.

- The first time you call `get_next_sprite()` it should return `mage_idle_0`
- The second time you call `get_next_sprite()` it should return `mage_idle_1`
- Continue in this way for each call, up until you return `mage_idle_9`
- The next call to `get_next_sprite()` should start back at `mage_idle_0`
- Suppose your mage calls `attack()`. Then the next call to `get_next_sprite()` should be `mage_attack_0`
- The call after will return `mage_attack_1`
- And so on until `mage_attack_9` (unless another call to `attack()` or `special_attack()` happens, in which case you go to their animation at frame 0)
- Then the next call will return `mage_idle_0`
- The call after returns `mage_idle_1`
- And suppose we call `special_attack()`. Then the next call to `get_next_sprite()` should return `mage_special_0`
- And so on until `mage_special_9`

Here's a visual guide for reference (the dotted arrows are jus omitting frames 2 ~ 8 for space).


Arrows: The sprite name returned after the next call to get_next_sprite()
When attack() is called, the next sprite is mage_attack_0
When special_attack() is called, the next sprite is mage_special_0
mage_idle_0
mage_idle_1
mage_idle_9
mage_attack_0
mage_attack_1
mage_attack_9
mage_special_0
mage_special_1
mage_special_9