CSC148 Summer 2018: Assignment 2

Due: Sunday, July 29th @ 11PM

Overview

In this assignment, you'll be adding to the game from Assignment 1. The game follows the same rules, and we have provided you with a solution to Assignment 1 to work from. Whereas Assignment 1 was focused on giving you practice in reading client code and designing classes, this assignment aims to give you:

- More practice with inheritance
- More practice with Queues
- Practice understanding and adding to someone else's code
- Practice with Trees
- Practice with recursion
- An example of how to write a recursive function iteratively
- Practice writing good documentation

If you're unfamiliar with Assignment 1 and how the game works, read over a1.pdf as this document won't be repeating the rules.

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, nor 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, like with Assignment 1, 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:

```
python3 -m pip install pygame
```

Download a2.zip and extract the files in it. This contains all of the starter code and files used by the starter code:

- **a2_game.py**: This is the same as in Assignment 1; you will need to fill in the dictionaries at the top. The additions to this file are:
    - 'v' and 's' as keys to CHARACTER_CLASSES: These should correspond to the Vampire and Sorcerer classes that you'll be writing.
    - 'mr' and 'mi' as keys to PLAYSTYLE_CLASSES: These should correspond to the Recursive Minimax and Iterative Minimax playstyles that you'll be writing.
    - 'r' as a key in BATTLE_QUEUE_CLASSES which maps to RestrictedBattleQueue. The class definition for RestrictedBattleQueue has been provided for you, but you must implement the rest of it.
- **a2_ui.py and a2_ui_nonpygame.py**: **When you want to run the game, run a2_ui.py!** These are almost identical to the UI files from a1. If PyGame doesn't work for you, use a2_ui_nonpygame.py.
- **a2_battle_queue.py**: This file contains the BattleQueue class and the class definition for RestrictedBattleQueue. You're responsible for implementing and documenting RestrictedBattleQueue.
- **a2_characters.py**: This file contains the Character class, as well as Mage and Rogue. All of these are fully implemented and documented. You are not to modify these classes, but you are free to write additional classes (e.g. Vampire and Sorcerer) in this file.

- **a2_skills.py**: This file contains all of the skills used by the characters in a2_characters.py. You are to familiarize yourself with the classes defined here, how they're used, and write additional skills for your Vampire and Sorcerer.
- **a2_skill_decision_tree.py**: This file contains the header for the SkillDecisionTree class that you are to implement for your Sorcerer (see Skill Decision Tree for details).
- **a2_playstyles.py**: This file contains the Playstyle, ManualPlaystyle, and RandomPlaystyle classes. You are to implement 2 additional playstyles: a recursive version of minimax, and an iterative version of minimax.
- **a2_vampire_unittest.py**: The file that contains very basic test cases for the vampire 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).
- **a2_sorcerer_unittest.py**: The file that contains very basic test cases for the sorcerer 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).
- **a2_skill_decision_tree_unittest.py**: The file that contains very basic test cases for the SkillDecisionTree class 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).
- **a2_restricted_battle_queue_unittest.py**: The file that contains very basic test cases for the RestrictedBattleQueue class 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).
- **a2_minimax_recursive_unittest.py**: The file that contains very basic test cases for the Recursive Minimax Playstyle 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).
- **a2_minimax_iterative_unittest.py**: The file that contains very basic test cases for the Iterative Minimax Playstyle 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).
- **sprites** (folder): This folder contains the sprites used by a2_ui.py. This works in the same manner as the sprites from Assignment 1, with the addition of Vampire and Sorcerer sprites.
- **a2_pyta.txt**: The python_ta file to use for this assignment.

As we are providing unittest files for each of the classes (compared to Assignment 1 which withheld the BattleQueue tests), the total number of unittests in each are lower and test only very basic cases and functionality. **You are responsible for testing any edge cases and ensuring your own code works properly under all scenarios.**

Description of Classes

Below is a description of the classes you'll need to implement in this assignment.

The Character, Rogue, Mage, and BattleQueue classes that have been provided match the requirements outlined in Assignment 1.

Characters

Like with Assignment 1, you are to implement 2 additional characters. We've provided the superclass Character for you already, which makes use of Skill and its subclasses.

There are a few methods that have been added in, such as copy(), set_hp(), set_sp(), and some helper methods.

Vampire

A vampire is a character with 3 defense. All attacks that a vampire performs will restore their HP based on the amount of damage its enemy took. For example, if a vampire attacks a character with 30 HP and leaves them with 20 HP afterwards, the vampire would gain 10 HP.

Their skills are as follows:

- "A" (Attacks): Deals 20 damage and adds the Vampire to the end of the BattleQueue. Takes 15 SP.
- "S" (Special Attacks): Deals 30 damage and adds the Vampire to the BattleQueue twice, and then adds its enemy to the BattleQueue once. Takes 20 SP.
  - For example, if the BattleQueue were to look like:
    V -> R
    Where V is the Vampire and R is its enemy, then after using a Special Attack, the BattleQueue would look like:
    V -> R -> V -> V -> R

Sorcerer

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

- "A" (Attacks): Deals damage and performs corresponding to the move chosen by its SkillDecisionTree (described below). Takes 15 SP regardless of what skill was used.
  - **EDIT:** In other words, this should use the skill returned by the SkillDecisionTree's pick_skill() method, but only use 15SP.
- "S" (Special Attacks): Deals 25 damage and resets the BattleQueue so that it only contains only 1 copy of each character before adding itself back into the BattleQueue. Takes 20 SP.
  - For example, if the BattleQueue looks like this before the Sorcerer attacks:
    S -> R -> R -> S -> R -> S
    Where S is the Sorcerer and R is its enemy, then after using a Special Attack, the BattleQueue would look like:
    S -> R -> S

Sorcerers must also have a method called set_skill_decision_tree() which takes in a SkillDecisionTree as a parameter. This SkillDecisionTree will be what the Sorcerer's Attack should use.

Skill Decision Tree

A Skill Decision Tree is a Tree which has the following attributes:

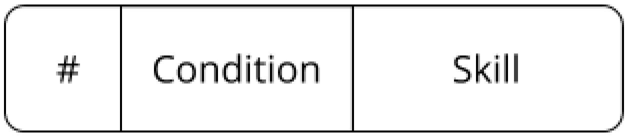
- Value: A skill.
- Condition: A function that takes in 2 parameters (the caster and their target, in that order), which returns a boolean.
  - The term 'caster' refers to the character that's attacking, and the term 'target' refers to the character that's being attacked.
- Priority number: A number representing the priority of this node. A lower number means that it has a higher priority over other nodes (i.e. 1 would have a higher priority than 2.)
- Children: A list of its subtrees.

The ___init___ for the SkillDecisionTree has been provided for you. A SkillDecisionTree is required to have the **pick_skill** method which takes 2 characters as parameters (the caster and their target, in that order), and returns a skill.

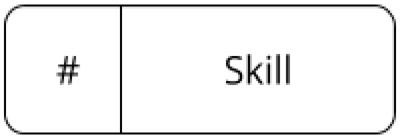
The skill chosen is the skill with the highest priority, and which fulfills all conditions in any trees above it, but fails on the tree's own condition if they're not a leaf. If the tree is a leaf, then it simply returns its own skill, regardless of what its condition is.

The skill returned by pick_skill is the skill that gets used in a Sorcerer's Attack.

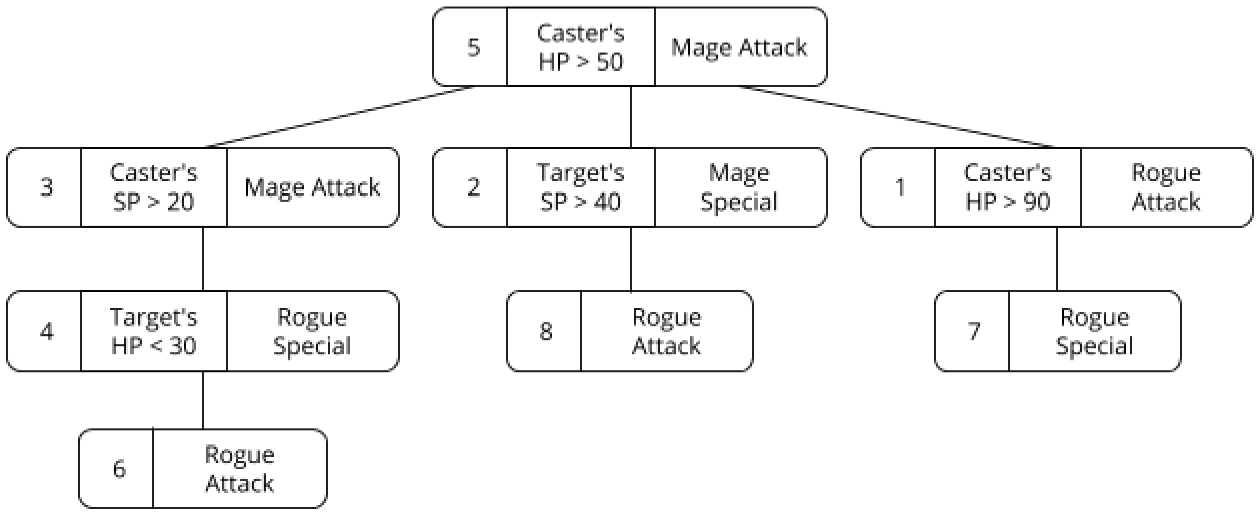
For the examples below, non-leaf nodes are represented like so:



Whereas leaves are denoted as:



Suppose we have a SkillDecisionTree that looks like:



Given the following scenario:

- Caster: HP = 100, SP = 40
- Target: HP = 50, SP = 30

The tree nodes that would fulfill these criteria are the ones with priorities 4, 2, and 7, which fulfill the conditions of all parent trees, but either fail their own conditions, or are leaves.

For example, for the node with priority 4:

- The root has a condition of Caster's HP > 50. Since the Caster's HP is 100, this condition is passed.
- The node with priority 3 has the condition of Caster's SP > 20. Since the Caster's SP is 40, this condition is passed.
- The node with priority 4 has the condition of Target's HP < 30. Since the Target's HP is 50, this condition fails, so its skill is now a possible choice (i.e. Rogue Special might be the skill used).

Since we want to return the skill with the highest priority (i.e. it has the lowest priority number), we would return the one associated with priority 2 (Mage Special). So, an attacking Sorcerer would use Mage Special and only consume 15 SP.

As another example, suppose we have the following scenario:

- Caster: HP = 80, SP = 40
- Target: HP = 20, SP = 50

In this case, the nodes with priority 1, 6, and 8 are the ones that fulfill this criteria, so we'd want the skill associated with priority 1 (Rogue Attack).

All methods that you write in SkillDecisionTree must be fully documented with type annotations, docstring descriptions, and docstring examples.

Additionally, you are to fill out the create_default_tree() function in a2_skill_decision_tree.py, which should create a tree matching the one in the above examples. You may pick any functions for the conditions in the leaves. The passes_condition function from the Additional Recursion Practice might be helpful in understanding how to pass functions around as parameters.

Note: If you have to create a function which doesn't use a parameter, replace the name with an __ (i.e. instead of function(caster, target), you could do function(__, target)).

RestrictedBattleQueue

A RestrictedBattleQueue is a type of BattleQueue with restrictions on when characters are able to add to it. In the original BattleQueue, copies of characters that were added to the BattleQueue would persist for the entirety of the game, leading to unbalanced games.

The first rule for a RestrictedBattleQueue is as follows:

- The first time each character is added to the RestrictedBattleQueue, they're able to add to it.

For the remaining rules, you may always assume that the character at the front of the RestrictedBattleQueue is the one adding to it:

- Characters that are added to the RestrictedBattleQueue by a character other than themselves cannot add to the RestrictedBattleQueue.
    - For example, given the following RestrictedBattleQueue:

Character order:	A ->	B	
Able to add:	Y	Y	

      Adding B would result in the following, since B is added by A:

Character order:	A ->	B ->	B
Able to add:	Y	Y	N

      Consequently, if the RestrictedBattleQueue reaches this state:

Character order:	B ->	A ->	B
Able to add:	N	Y	Y

      Nothing would happen if B tries to add themself, so the RestrictedBattleQueue would remain the same.
- Characters can only have two instances of themselves with the ability to add to the RestrictedBattleQueue at any given time.
    - For example, given the following RestrictedBattleQueue:

Character order:	A ->	A ->	B	
Able to add:	Y	Y	Y	

      If A attempts to add themself in, the RestrictedBattleQueue would look like:

Character order:	A ->	A ->	B ->	A
Able to add:	Y	Y	Y	N

      If we remove from the RestrictedBattleQueue and try to add A in again, it would then look like:

Character order:	A ->	B ->	A ->	A
Able to add:	Y	Y	N	Y

Make sure you also implement the copy() method for RestrictedBattleQueue.

Minimax Playstyles

For this assignment, you are to implement 2 additional playstyles. Both of these do the same thing but are to be implemented differently: one recursively, and the other iteratively (i.e. not using recursion at all). It's recommended that you understand and implement the recursive version before working on the iterative one, as it's conceptually easier.

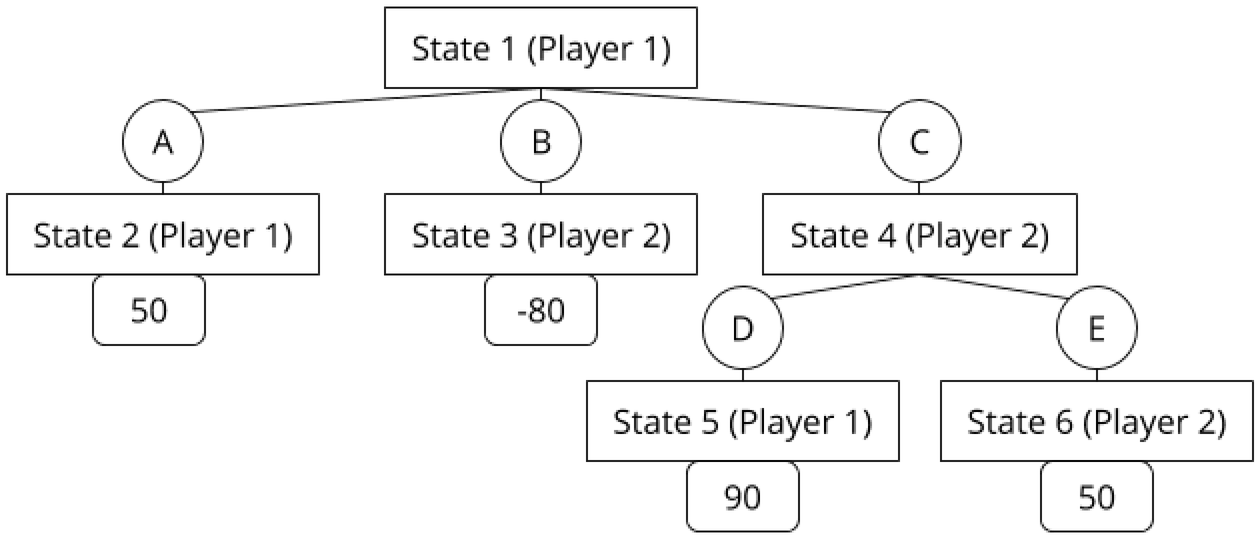
These Playstyles should be usable with any combination of characters and any type of BattleQueue (e.g. BattleQueue, RestrictedBattleQueue, and other subclasses of BattleQueue if we were to write any other alterations).

Minimax

Minimax works by choosing a move that guarantees the best end state no matter what the opponent does. For example, suppose we have a game with 3 moves available to us: A, B, and C.

Suppose using the move A would guarantee a score of 10, B would guarantee a score of 20, and C would guarantee a score of -10. In this scenario, Minimax would pick move B.

Suppose we've mapped out every possible combination of actions for some abstract game in the diagram below, where State # represents how the current game looks and the current player, and the letters represent moves that they can make. A number being below a state represents that the game is over, and the number is the score for the current player of the state. The score for the opposite player would be -1 * that score (i.e. if Player 2's score is -10, then Player 1's score would be 10).



If we were to pick a move for Player 1 in State 1, we would want to determine which of A, B, or C would guarantee the highest score.

- If Player 1 picks A, then they would be able to end the game with a score of 50.
- If Player 1 picks B, then they would be able to end the game with a score of 80 -- since Player 2 has a score of -80, Player 1's score must be -1 * -80 = 80.
- If Player 1 picks C, then the score depends on Player 2's choice:

    - If Player 2 picks D, then Player 2 would be able to end the game with a score of -90, and thus Player 1 would have a score of 90.
    - If Player 2 picks E, then Player 2 would be able to end the game with a score of 50, and thus Player 1 would have a score of -50.
    - If Player 2 was playing optimally, this means they would always pick the move E, since that provides them with the highest score.
        - This also means that Player 1 would only be able to guarantee themselves a score of -50 if they pick C.
        - Thus:
          If Player 1 picks C, then they would be able to end the game with a score of at least -50.

Player 1 would want to pick the move that guarantees the best score for them, which would be move B, so that's what minimax would choose. With minimax, we leave nothing up to chance: Assume that the both players would make moves that could guarantee the absolute best for themselves no matter what the opponent does (i.e. assuming the opponent also plays perfectly.)

Score for Our Game

For our game (which is carried out in a BattleQueue), the scores are as follows for a game that is over (i.e. BattleQueue.is_over() returns True):

- If it's a tie (there's no winner), then the score is 0.
- If the current player of the state is the winner, then we return their HP.
- If the current player of the state is the loser, then we return -1 * the winner's HP.

The score for a game that's not over is the highest score that can be guaranteed for the current player.

'Current player' here refers to the next player who should have chosen a move if the game wasn't over (i.e. whoever would be returned by BattleQueue.peek()). If both players have 0 SP left when one character has 0 HP, then the 'current player' defaults to the first player added to the BattleQueue. See BattleQueue's peek() for details.

We have provided you with the header for a function called get_state_score(), which takes in a BattleQueue as a parameter, and which should return the highest guaranteed score for the current player in the BattleQueue. All documentation for this function has also been provided, but the implementation is up to you. This function should be implemented recursively.

Since both attack() and special_attack() methods modify the BattleQueue, you'll want to work with a copy of a BattleQueue instead of operating on the original BattleQueue passed in. To make your life a bit easier, we've implemented the copy() method in the starter code for BattleQueue (although you'll have to implement this yourself for RestrictedBattleQueue). The BattleQueue should be in its original state after get_state_score() is called, as well as after select_attack() is called for your Minimax Playstyles, detailed below.

Minimax (Recursive)

As with the other Playstyle classes, select_attack() should return a string ('A' or 'S'). The skill chosen should be the one that can guarantee the highest score, and so should make use of the get_state_score() function.

Minimax (Iterative)

This should produce identical results to the recursive version, but you should be implementing this without recursion. For this, you will need to keep track of the various versions of the game being played out via a Stack. You may either implement a Stack class, use a list and treat it as a stack, or use the Stack class that was provided in class. **If you are implementing a Stack/using the one from class, make sure you submit the file containing it when submitting your assignment.**

For your iterative minimax, you'll need to use a class that has a tree-node-like structure (though all that's needed are the attributes, not any methods), which keeps track of 3 things:

- The BattleQueue at that point of the game (i.e. the 'state' of the game)
- The children that are reachable from this state (i.e. a list of other tree-node-like structures). Initially, this should be `None` as opposed to an empty list (explained later).
- The 'highest guaranteed score' reachable from this state (similar to what would have been returned by the get_state_score function() from Recursive Minimax, but you can't use that here). Initially, this should be set to None.

We'll wrap our initial BattleQueue (hereby referred to the "state" of the game) in the tree-node-like structure, and add it to our stack. Afterwards, use a loop to progressively remove from the top of the stack.

If the state is over, then the 'highest guaranteed score' should be updated to be the score of the state (i.e. as described in 'Score for Our Game').

If the state's not over, then we try the following:

- If the children is `None`, then we'll want to look at all of the available actions and have the states that correspond to those actions being used as the children. Then we add this state back into the stack, and then add all of those children in.
- If the children is a list, then that means we've gone through all of the children and they all have their guaranteed scores filled out. Thus, using the scores of those children (and their states/finding the current player of their states), you can figure out the highest guaranteed score for the current state.

Once we know the score for our initial state and its children, figuring out the move to be returned will be similar to Recursive Minimax.

Playstyle Restrictions

If you use recursion in the iterative minimax, you will receive a 0 for all iterative minimax tests. Conversely, **if your recursive minimax doesn't use recursion at all, you will receive a 0 for all recursive minimax tests.** As get_state_score() is a recursive function, you must ***not*** use it within your iterative minimax.

Your Playstyles must work with the original BattleQueue provided and any other BattleQueues that work properly, along with Rogue, Mage, and any other characters that correctly implement the Character class. In other words: **Don't modify the original BattleQueue, Mage, Rogue, or Character classes** (or use any additional methods you've created for them)**.** This is to make sure your solution is 1) generalizable, and 2) able to be tested despite any changes you might make to BattleQueue/Mage/Rogue/Character.

Documentation

As we have provided fully documented classes (Character, Mage, Rogue, BattleQueue), the grading scheme for documentation will become more strict. In particular, we will be following the Docstring guidelines from CSC108.

The documentation requirements are as follows:

- For all methods and functions that you write, you must include:
    - Type annotations for every parameter (except for self in the case of methods)
    - Type annotations for the return type
    - A docstring description
    - At least 1 docstring example
        - The docstring example **must** pass **doctest**
- For all classes that you write, you must include:
    - A docstring description that describes your class
    - A description for every **public** attribute (this is to be written within the class docstring)
    - A type annotation for every public attribute (this is to be written below/outside of the class docstring)

For functions and methods, the guidelines are as follows when writing a docstring description:

1. Describe precisely *what* the function does. Do not reveal *how* the function does it.
    - For example, the following docstring is bad:
      "Return a list of the available actions for this Character by checking whether the .__sp attribute of this character is greater than the cost of its Attack and its Special Attack."
      Whereas the following docstring is good:
      "Return a list of the available actions for this Character."
2. Make the purpose of *every* parameter clear. Use the name of every parameter in the description.
    - For methods, you may omit mentioning the name self if you use 'this <Class>'. For example, saying "Return the leaves in this Tree" is fine, and so is "Return the leaves in self". However, saying "Return the leaves" is unclear and bad style.
    - If you have a function with the header get_sum(lst, starting_value), the description should look like:
      "Return the sum of lst, starting from starting_value."
        - This may seem strange at some times. For example:
          "Initialize this Mage with the name name, BattleQueue battle_queue, and Playstyle playstyle."
          However, you should still be writing it this way if you're using

descriptive parameter names. If the parameter names were x, y, and z instead, then the docstring would look like:
"Initialize this Mage with the name x, BattleQueue y, and Playstyle z."
But if you didn't mention what the parameters were used for, you would just have:
"Initialize this Mage with x, y, and z."
Or, if you only mentioned what they're used for:
"Initialize this Mage with a name, BattleQueue, and Playstyle."
Which is difficult to understand when your parameters are named x, y, and z.

3. For functions that return values, be clear on what the return value represents.
4. Explain any conditions that the function assumes are true. These preconditions should **not** refer to type requirements, because those are already in the function header. However, sometimes there are conditions not covered by types, and you should include those. For example, if a function requires parameters `x: int` and `y: int` to be even and parameter `s: str` to not be empty, include lines like:
```
Precondition: x and y must both be even.
              len(s) > 0.
```
after the description and before the example.
5. Be concise and grammatically correct.
6. Write the description as a command (e.g., Return the first ...) rather than a statement (e.g., Returns the first ...)

Each of these criteria will be considered when we are grading your documentation.

Submission

Exercises are to be submitted through MarkUs in the a1 folder. You only need to **submit a2_game.py, a2_battle_queue.py, a2_playstyle.py, a2_skills.py, a2_characters.py, a2_skill_decision_tree.py, and any other files you've written/imported into a2_game.py.** You do not need to submit any of the icons provided, a2_ui.py, a2_ui_nonpygame.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 under special circumstances.

Grading Scheme

Below is the grading scheme for this assignment. Your final grade should not come to a surprise to you, since you should be able to estimate at least 60% of your grade using the following rubric.

20% Code Style

- **10% PythonTA**: For every PythonTA violation, you will lose a portion of this grade. Make sure all of the files you submit are PythonTA compliant.
    - This excludes a2_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. See the Documentation section.
    - Documentation will be marked very strictly for this assignment, as you now have multiple examples at your disposal.

80% Code Correctness

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

- **40% for the Provided Unittest**s: You will lose a portion of your mark for each test case you fail (proportional to how many test cases are in the file).
    - 5% for a2_vampire_unittest.py
    - 2% for a2_sorcerer_unittest.py
    - 3% for a2_skill_decision_tree_unittest.py
    - 10% for a2_restricted_battle_queue_unittest.py
    - 10% for a2_minimax_recursive_unittest.py
    - 10% for a2_minimax_iterative_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. Since we have provided unittests for every class, there are fewer test cases provided for each of them, and only very basic scenarios are checked. It's up to you to make sure your code works properly in all cases.
- **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 for a RestrictedBattleQueue.
        - Whether or not you're using inheritance.
        - Functionality of your RestrictedBattleQueue class.
            - Especially testing your BattleQueue to make sure it ignores characters without enough SP to act.
            - We may forcibly add/remove from a RestrictedBattleQueue to put things into a certain order, and make sure they can only add when they're supposed to.
        - Using a different SkillDecisionTree for a Sorcerer instead of the tree returned by create_default_tree().
        - Various combinations of characters in Minimax (e.g. Sorcerer vs. Sorcerer, Vampire vs. Sorcerer, Rogue vs. Mage, and so on).
            - We might also test with characters that are completely new (i.e. not a Sorcerer, Mage, Vampire, or Rogue).
    - We will **not** test on any invalid input. For example:
        - We will not try to call the Minimax playstyles on a game that's over (i.e. there's no move to be chosen).
            - We may call get_state_score() on states that are over, however.
        - 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 perform certain attacks if they're not valid, but we will want to 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. you may assume that a player and their enemy will always be 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.

- 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.