Pokémon

Assignment 2
Semester 2, 2021
CSSE1001/CSSE7030

Due date: 16:00, 8th of October, 2021 GMT+10

1 Introduction

Loosely speaking, the *engine* of a game *computes the state of the game* whereas the graphical user interface *displays the state*. For instance, in chess, `change_position` was part of the game engine whereas `print_board` is part of the GUI.

For this assignment implement a (simplified) game engine for Pokemon battles. For those unfamiliar with how Pokemon battles play out, a video example is provided here.[1]. The extent to which our engine differs from the real game is described in Section 3.

You are *not* developing a GUI because we have provided one (so you can play the game whose engine you wrote).

You are required to implement a number of classes and their subsequent methods which are described in Section 4. Any changes to these requirements will be listed in a changelog under the Assignment 2 folder in Blackboard.

Your program's output must match the expected output *exactly*; minor differences in output (such as whitespace or casing) *will* cause tests to fail, resulting in *zero marks* for that test.

2 Getting Started

Download `a2.zip` from Blackboard — this archive contains the necessary files to start this assignment.

Once extracted, the `a2.zip` archive will provide the following files/directories:

`a2.py`
: The game engine. This is *the only file you submit and modify. Do not* make changes to any other files.

`a2_support.py`
: Constants, functions, and classes you *may* find helpful for writing the engine. Read the docstrings in order to understand how to use them in your solution.

`game.py`
: If you wish, run this file to play the your Pokemon game.

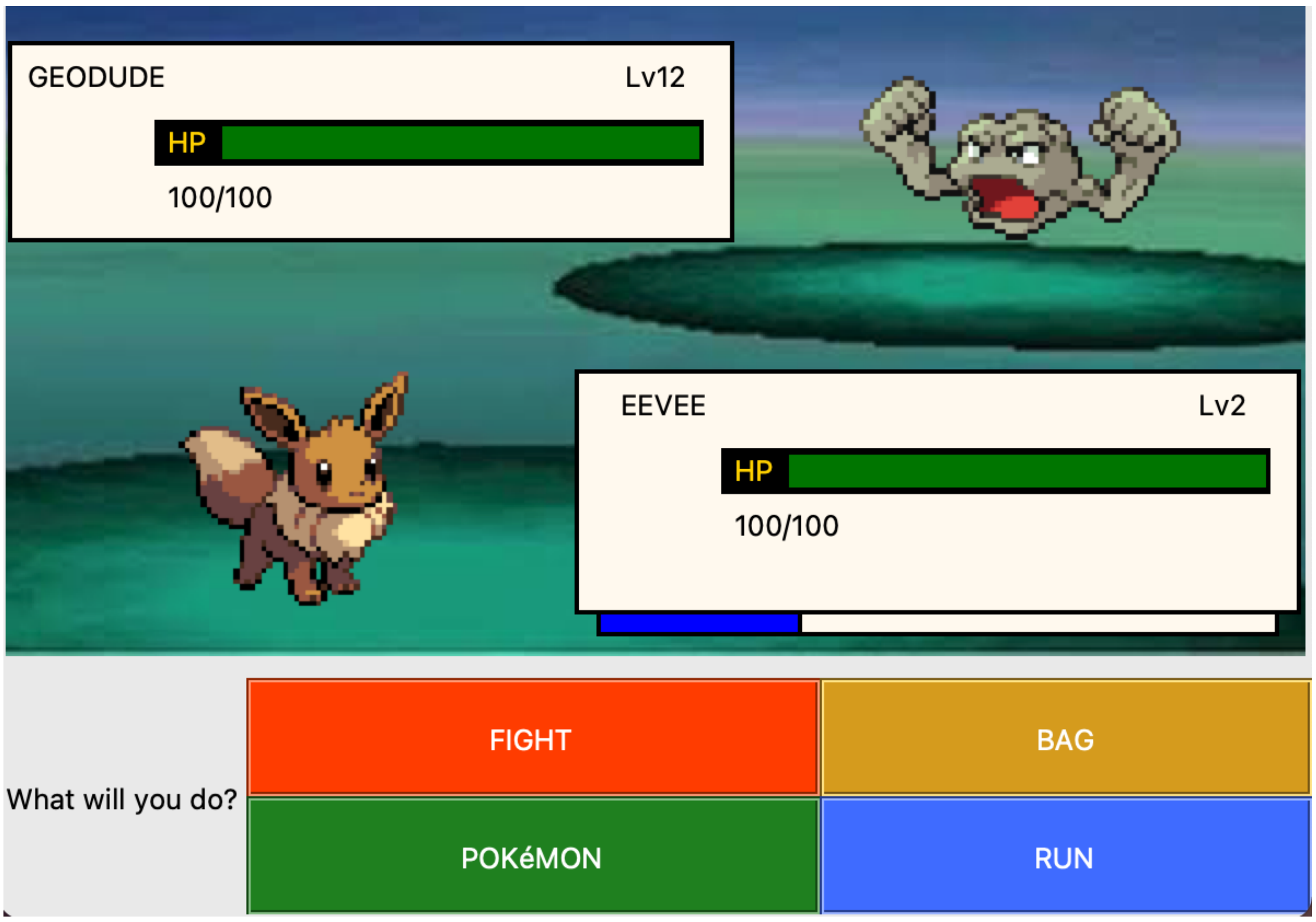
`battle_view.py`
: Instruction for drawing the game to the screen.

[1] `https://youtu.be/s37zGwC0Z7Y?t=32`

`data.py`
: Data about Pokemon, moves and trainers, used in the game.

`images`
: A folder with images used by the `battle_view`.

3 Gameplay



The general structure of a Pokemon battle is given in the next section, followed by a more in-depth description of each component.

3.1 Overview

Pokemon battles take place between two trainers, say, Ash and Brock. Both Ash and Brock have a roster of (at most) six Pokemon, that battle until all Pokemon in one roster have *fainted*. Trainers can only ever have *one* Pokemon on the battle field while the rest are in reserve.

Battles are turn-based. Each turn, a trainer may select an action to perform. These actions have priority, and are enacted in order of that priority.

If a Pokemon faints in battle, the only valid action its trainer may take, is to switch it out for another non-fainted Pokemon from the roster, presuming one exists. If all Pokemon in a roster have fainted, the trainer whose Pokemon are left standing is the winner, and the battle ends.

3.2 Actions

Each turn, a trainer may perform *one* of the following actions:

Fight: Order the Pokemon on the field of battle to use a move.

Item: Use an item from their inventory

Switch: Swap the active Pokemon with one from the roster.

Flee: The trainer attempts to flee from the battle.

When valid actions are *applied*, they alter the battle from one state to the next.

3.3 Pokemon

Each Pokemon in the roster has the following attributes:

Name: A name.

Type: An elemental type which influences how vulnerable the Pokemon is to certain moves.

Level: A general measure of how strong a Pokemon is.

Health: *Also referred to as HP (health points)*, when a Pokemon's health reaches 0, it is said to have fainted.

Stats: A description of the Pokemon's attack and defense capabilities. Pokemon stats increase with level.

Moves: A maximum of four moves that the Pokemon can use while battling.

3.4 Moves

Like the Pokemon itself, each move has an elemental type which determines how effective it is against other Pokemon. Each move also has a number of uses before it cannot be used again.

Moves have the further subcategories:

Attacks: If successful, reduce the health of the enemy Pokemon.

Buffs: Increase (i.e. strengthen) the current Pokemon's stats.

Debuffs: Decrease (i.e. weaken) the enemy Pokemon's stats.

4 Implementation

There are many more classes required for this assignment that *are not* describe in the gameplay section. The class diagram below lays out *all* of these classes, and the basic relationships between them.

- Hollow-arrowheads indicate *inheritance* (i.e. the "is-a" relationship).
- Dotted arrows indicates *composition* (i.e. the "has-a" relationship).

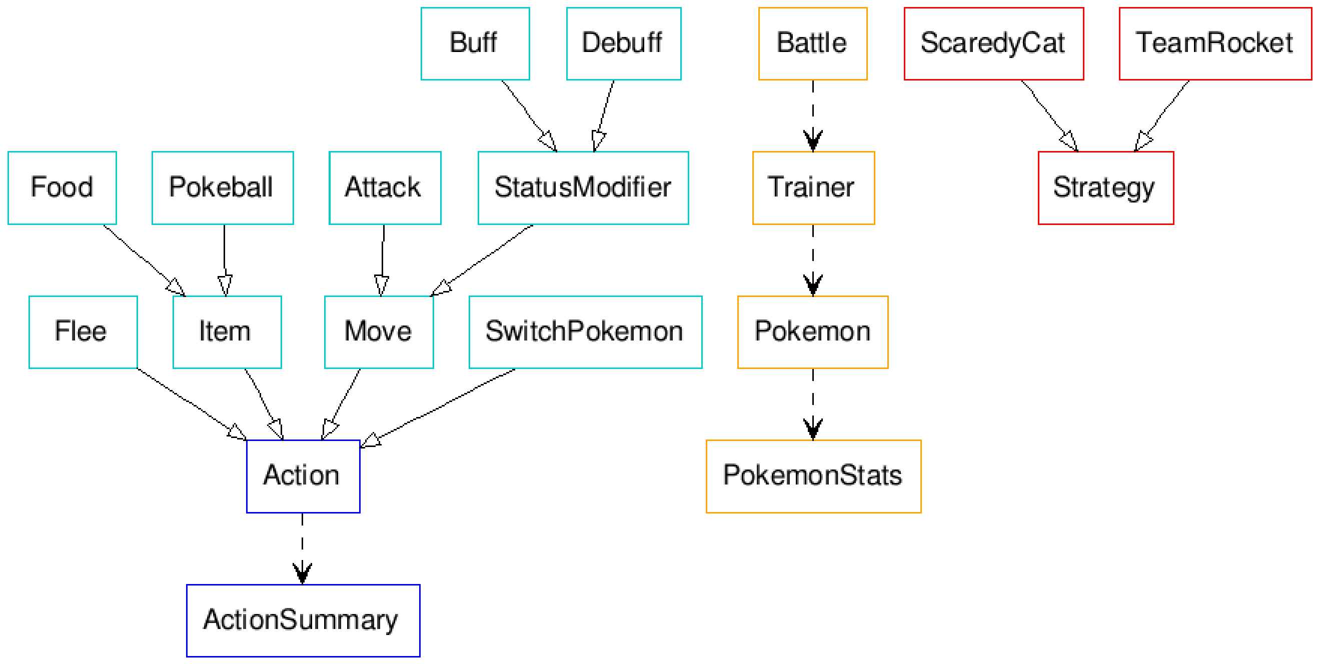


Figure 1: Basic class relationship diagram for the classes which need to be implemented for this assignment. Strategy and its subclasses are Masters-only tasks.

A few of the relationships (e.g. between Action and Battle) have been ommitted for the sake of formatting. The precise relationships are captured in the documentation referenced in the next section.

4.1 What goes inside each class?

This assignment uses online documentation, that is accessible on the CSSE1001 website.[2] You will find a description of each class and its required methods at the link.

4.2 Where to start

When implementing a spec where each component has dependencies, it is often best practice to start with the components that have the least dependencies. In this case, both the `ActionSummary` and `PokemonStats` classes have no dependencies and should therefore be implemented first. It is also recommended that you leave `Action` and its subclasses until last.

The CSSE1001 youtube channel has some video examples [3] of how to attack a spec like this, that you might find useful.

[2] `http://csse1001.uqcloud.net/assignments/a2.html`
[3] `https://youtu.be/DcVqhoCJDx0`

4.3 `ActionSummary` Messages

Every time an action is 'applied' to the game state, it returns a corresponding `ActionSummary` containing the effects of that action. This section details the messages expected from applying each action.

Flee:

- On success: `Got away safely!`
- On failure: `Unable to escape a trainer battle.`

SwitchPokemon:

- If trainer's turn and their Pokemon has not yet fainted: `<PokemonName>, return!`
- On success: `<TrainerName> switched to <NextPokemonName>.`

Food:

- On consumed: `<PokemonName> ate <itemName>.`

Pokeball:

- In trainer battle: `Pokeballs have no effect in trainer battles.`
- On failure: `It was so close, but <PokemonName> escaped!`
- On success: `<PokemonName> was caught!`
- If full roster: `<PokemonName> was caught, but there was no more room.`

Moves:

- On use: `<PokemonName> used <moveName>.`
- On miss: `<PokemonName> missed!`
- On buff: `<PokemonName> was buffed for <int> turns.`
- On debuff: `<PokemonName> was debuffed for <int> turns.`
- If move kills enemy: `<EnemyPokemonName> has fainted.` Then another message `<PokemonName> gained <int> exp.`

4.4 Terminology

`player/enemy` Player refers to the `Trainer` controlled by the person playing the game, while enemy is the enemy AI that represents the other `Trainer` in the battle.

`stat` Stat is a word traditionally used in games to represent some information on a character, e.g. a measure of their hit points or the strength of their attacks. In our game, they are tied to a Pokemon and give a rough idea of their strengths and weaknesses.

`modifier` In our implementation of Pokemon, a modifier is a tuple of changes to be 'applied' to the Pokemon's stats. The exact structure of these modifiers is explained in the online documentation.

4.5 Documentation Requirements

There are a significant number of classes and contained methods you have to implement for this assignment. For each one, *you must provide documentation* in the form of a docstring. The only exception is for overridden methods on subclasses, as python docstrings are inherited.

5 Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. apply basic object-oriented concepts such as classes, instances and methods,
3. read and analyse code written by others,
4. analyse a problem and design an algorithmic solution to the problem,
5. read and analyse a design and be able to translate the design into a working program, and
6. apply techniques for testing and debugging.

5.1 Marking Breakdown

Your total grade for this assessment piece will be a combination of your functionality and style marks. For this assignment, functionality and style have equal weighting, meaning you should be devoting at least as much time towards proper styling of your code as you do trying to make it functional.

In order to be allocated a non-zero grade for this Assignment you *must* attend an interview during your scheduled practical the week following your submission deadline (October 11–15).

5.2 Functionality Marking

Your program's functionality will be marked out of a total of 50 marks. As in assignment 0, your assignment will be put through a series of tests and your functionality mark will be proportional to the number of tests you pass. You will be given a *subset* of the functionality tests before the due date for the assignment. You may receive partial marks within each class for partially working methods, or for implementing only a few classes.

For this assignment, **abstract classes will not be subject to functionality marking.** You should use your abstract classes to group common functionality on their subclasses. We are intentionally setting things up this way to give you freedom in applying object-oriented design principles. Your efficacy in doing so will instead be evaluated as part of your style marks.

The remaining non-abstract classes all have equal weighting for functionality. For CSSE7030 students, the `create_encounter` function is considered part of the `ScaredyCat` class for marking purposes.

You need to perform your *own* testing of your program to make sure that it meets *all* specifications given in the assignment. Only relying on the provided tests is likely to result in your program failing in some cases and you losing some functionality marks. Note: Functionality tests are automated, so string outputs need to match *exactly* what is expected.

Your program must run in the Python interpreter (the IDLE environment). Partial solutions will be marked, but if there are errors in your code that cause the interpreter to fail to execute your program, you will get zero for functionality marks. If there is a part of your code that causes the interpreter to fail, comment out the code so that the remainder can run. Your program must run using the Python 3.9 interpreter. If it runs in another environment (e.g. Python 3.8 or PyCharm) but not in the Python 3.9 interpreter, you will get zero for the functionality mark.

5.3 Style Marking

The style of your assignment will be assessed by a tutor. Style will be marked according to the style rubric provided with the assignment. The style mark will also be out of 50.

The key consideration in marking your code style is whether the code is easy to understand. There are several aspects of code style that contribute to how easy it is to understand code. In this assignment, your code style will be assessed against the following criteria.

- Readability
  - Program Structure: Layout of code makes it easier to read and follow its logic. This includes using whitespace to highlight blocks of logic.
  - Descriptive Identifier Names: Variable, constant, function, class and method names clearly describe what they represent in the program's logic. Do **not** use what is called the *Hungarian Notation* for identifiers. In short, this means do not include the identifier's type in its name (e.g. `item_list`), rather make the name meaningful. (e.g. Use `items`, where plural informs the reader it is a collection of items and it can easily be changed to be some other collection and not a list.) The main reason for this restriction is that most people who follow the *Hungarian Notation* convention, use it poorly (including Microsoft).
  - Named Constants: All non-trivial fixed values (literal constants) in the code are represented by descriptive named (symbolic) constants.
- Documentation
  - Comment Clarity: Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0.`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.
  - Informative Docstrings: Every class, method and function should have a docstring that summarises its purpose. This includes describing parameters and return values so that others can understand how to use the method or function correctly.
  - Description of Logic: All significant blocks of code should have a comment to explain how the logic works. For a small method or function, the logic should usually be clear from the code and docstring. For long or complex methods or functions, each logical block should have an in-line comment describing its logic.

Structure will be assessed as to how well your code design conforms to good object-oriented programming practices.

- Object-Oriented Program Structure
  - Classes & Instances: Objects are used as entities to which messages are sent, demonstrating understanding of the differences between classes and instances.
  - Encapsulation: Classes are designed as independent modules with state and behaviour. Methods only directly access the state of the object on which they were invoked. Methods never update the state of another object.
  - Inheritance & Polymorphism: Subclasses are designed as specialised versions of their superclasses. Subclasses extend the behaviour of their superclass without re-implementing behaviour, or breaking the superclass behaviour or design. Subclasses redefine behaviour of appropriate methods to extend the superclasses' type. Subclasses do not break their superclass' interface.

- Algorithmic Logic
  - Single Instance of Logic: Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a method or function.
  - Variable Scope: Variables should be declared locally in the method or function in which they are needed. Attributes should be declared clearly within the `__init__` method. Class variables are avoided, except where they simplify program logic. Global variables should not be used.
  - Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. loops and conditional statements).

5.4 Assignment Submission

This assignment follows the assignment submission policy as assignment 0. Please refer to the assignment 0 task sheet.

You must submit your assignment as a single Python file called `a2.py` (use this name – all lower case), and *nothing else*. Your submission will be automatically run to determine the functionality mark. If you submit a file with a *different name*, the tests will *fail* and you will get *zero* for functionality. Do *not* submit the `a2_support.py` file, or any other files. Do *not* submit any sort of archive file (e.g. zip, rar, 7z, etc.).

5.5 Plagiarism

This assignment follows the same plagiarism policy is as per assignment 0. Please refer to the assignment 0 task sheet.