CS127 Project V.2

Due: Monday, November 16, 2020 at 11:59 PM

Warning about the due date: The project is due at literally the last minute of the term. As a result, no extensions will be granted nor late work accepted. Start early, submit early, make sure Moodle is updated with your draft work in case of last-minute emergencies.

Failure to do so will turn me into one of those professors that lie about due dates, setting them a weekly early, and then give the entire class "extensions" to the due date they really intended the whole time because they don't want to deal with students with poor time management skills coming to beg for extra time. As a new professor, my earnestness and innocence for the rest of my career are entirely in your hands.

Introduction

Partnerships

You may work with a partner for this project. It is highly recommended. To form a partnership, please email Brian informing him that you and your partner wish to work together, and **cc your partner** in the email.

Both partners are expected to contribute equally to the completion of this project. When you submit this work for credit with both your names on it, the expectation is that the submitted work is the product of your joint efforts. If it emerges that one partner did not contribute to this project, that partner **will receive a 0**, and one **or both** partners may possibly be cited for an academic honesty offense.

Submission Requirements

For this project, you are expected to submit the following file, with the following contents to Moodle:

1. `pokemove.py`
    - `PokeMove`
        * `__init__(self, ...)`
        * `__str(self)`
        * `eval_effect_chance(self)`
2. `pokemon.py`
    - `Pokémon`
        * `__init__(self, ...)`
        * `calc_stat(self, ...)`
        * `choose_random_move(self, ...)`
        * `__str__(self, ...)`
        * `__repr__(self, ...)`
3. `io_functions.py`
    - `read_moves(...)`

- `read_pokemon(...)`

Bonus Features

In addition to the minimum requirements, there are various bonus features that you may complete for extra credit. You may also suggest your own bonus features to implement, in addition or instead of the suggested ones.

You are **strongly** advised to complete the base project in its entirety before working on any bonus features, as any errors in the base game may cause your bonus features to become unusable/ungradeable.

Starter Code

Starter code for the Pokémon Project is available here: https://repl.it/@blaw/CS127-F20-Pokemon-Project-Starter-Code. You should fork your own version of the starter code to work with.

You are highly advised to work with the provided starter code; it is a fully functional version of the Pokémon program from your various homework assignments, with modifications made so that it will mesh properly with the new parts you will complete in this project. If you wish to continue work from your Pokémon code, you may do so, but this will likely increase the difficulty of this project.

The starter code provides you with the signatures of all the classes, functions and methods you are to write.

moves.txt

`moves.txt` is a plain-text file that contains all the data for all of the moves available in the first-generation Pokémon games. The file is arranged in tab-separated columns, with a header row and each move subsequently occupying one row.

You will be reading in this file in `iofunctions.py` to create a PokeMove object for each move in this file, all of which will then be stored in a dictionary in `pokeconstants.py` for easy look-up. All the data in this file, other than the first column, should be stored, even though our simulation currently does not have Power, Accuracy, Priority, Effect, or Effect Chance; for now, you can just treat those as ambiguous attributes that you may choose to examine for bonuses.

Note: Not all Pokémon moves have Accuracy, Effects, or Effect Chances. If a Pokémon move does not have one of these features, the corresponding entry in the corresponding column and row is left empty, though the tab characters around it remain.

pokedata.txt

`pokedata.txt` is a plain-text file that contains all the data for all of the Pokémon available in the first-generation Pokémon games. The file is arranged in tab-separated columns, with a header row and each Pokémon subsequently occupying one row.

The numbers in the 3rd-7th columns are the base stats for each Pokémon species. For example, all Bulbasaurs have a base HP stat of 45, a base Attack stat of 49, a base Defense stat of 49, a base Speed stat of 45, and a base Special stat of 65. In the final two columns are its types, Grass and Poison.

Note: Not all Pokémon have two types, in which case the last column is left blank, though the delimiting tab character remains.

You will be reading in this file in `iofunctions.py` to create a nested dictionary in `pokeconstants.py` that will allow quick look-up of a Pokémon's various base stats and types.

io_functions.py

This file contains two functions from Part P of Homework 7 that read in type-related data from two text files for use in the program.

You will complete two more functions in this file to read in move data and Pokémon data from text files for use in the program. These are described below.

pokeconstants.py and pokemon_main.py

These two files have been updated and rewritten in anticipation of your completing the tasks required of you in this project. Once your code is complete, the entire program should run just like before again, albeit with a much simpler, better organized codebase.

In `pokeconstants.py`, the two new IO functions you will write in io_functions.py are called on lines #11 and 12. This will load up the move data and Pokémon data in the text files for use in the program. This will enable you to add new Pokémon to the game other than Pikachu and Blastoise, as well as give those Pokémon moves beyond the eight we had previously hard-coded into the program.

`pokemon_main.py` has been greatly rewritten, with `eval_crit(...)`, `calc_stat(...)`, and `calc_hp(...)` removed to be replaced by new methods you will write in the `Pokemon` class. Much of the attack-specific code in `simulate_battle()` has been moved to `process_attack(...)` now, and the battle loop in `simulate_battle()` itself has been greatly rewritten for simplicity.

Note that `process_attack(...)` now has a lot more functionality, as it handles all the pre-attack calculations and post-attack effects that were previously performed in the main battle loop. This is enabled by the new `Pokemon` and `PokeMove` classes: we no longer have to refer to specific variables or arrays, but rather just to the objects that contain all the data needed. **If you are going to attempt the bonus features involving new move effects, you will have add that code here in** `process_attack(...)`**.**

On the other hand, `simulate_battle()` is a lot simpler than it used to be, with much of the calculation work outsourced to `process_attack(...)` and a lot of the battle loop simplified with our new use of objects. This will enable you to add bonus features like larger Pokémon teams, user input, etc. more easily!

Your Tasks

In Brief

1. Write a new class `PokeMove` for Pokémon moves.
2. Write two new functions in io_functions.py to read in Pokémon move and Pokémon data from provided text files.
3. Write a new `Pokemon` class for Pokémon.

`PokeMove.py`

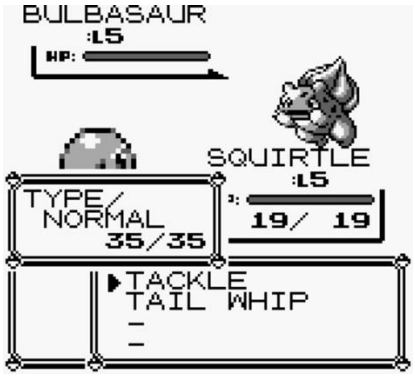
`PokeMove` is a new class used to represent a move usable by a Pokémon. `PokeMove`s will be automatically generated from reading in the `moves.txt` file.

Attributes

The `PokeMove` class should have eight attributes, corresponding to each of the columns in `moves.txt` other than the first.

Methods

1. `__init__(self, ...)`
    - The initializer for `PokeMove` should have 8 parameters, corresponding to each of the eight attributes and to each of the columns in `moves.txt` other than the first.
    - Each parameter should simply be passed in and set as an attribute inside the PokeMove object.
    - **Note**: Because some of the attributes/arguments will be missing for some Pokémon moves, those attributes/parameters should receive the value `None` instead.
    - **Note**: To provide a type hint for those parameters may receive None, use `Optional[<the type used when the argument exists>]`. This is how a type hint indicates that a None might be passed instead. (`Optional` is already imported for you at the top of the `pokemove.py`.)
2. `__str__(self)`
    - This method should return a string with the move's name and type. In theory, this method could be used to create an interface that looks something like this:



(Src: https://www.gamelib.app/games/pokemon-red, c/o Google Image Search)

3. `__repr__(self)`
    - This method should return a string with all of the `PokeMove`'s attributes, for debugging purposes.

- **Hint**: You may wish to look at the `Student`, `Course`, and `Section` `__repr__`s provided in Extensive Student Class Example linked in the Week 11 notes for an example of how `__repr__`s can look: https://repl.it/@blaw/CS127-F20-Extensive-Student-Class-Example

4. `eval_effect_chance(self)`
    - This method should randomly determine if this `PokeMov`e's effect occurs by chance.
    - Return a boolean indicating whether this PokeMove's effect occurs or not.
    - This method will be used like the function `eval_crit(...)` is. When a Pokémon uses a move, this method will be called to determine whether its effect is applied or not based on the move's effect chance and a randomly generated number in the range [0, 100].
    - **Note**: If a move has no special effect, like Tackle which only does damage, then this method should always return `False`.
    - **Note**: In `moves.txt`, if a move has an effect but no effect chance, that means that the effect is always applied as the primary effect of the move. In this case, this method should return `True`.

`io_functions.py`

In addition to the existing functions `read_type_chart(...)` and `read_type_physical(...)`, you will add two more methods to read in Pokémon data files into the program.

read_moves(filename)

This method will be used to read `moves.txt` and create/return a dictionary containing all the moves knowable by Pokémon in the game.

1. Has 1 parameter, which should typically receive the argument `'moves.txt'`.
2. Returns a nested dictionary mapping the names of every Pokémon move in `moves.txt`, as a string, to a `PokeMove` representing that move.
3. When completed, this function will be called successfully in `pokeconstants.py`, on line 11 of the starter code, and make all Pokémon move data easily accessible from anywhere in the program in a dictionary named `pokeconstants.MOVES` This dictionary is then accessed in the `Pokemon` class' constructor to "teach" each Pokémon their moves.
    - **Example**: An example access of this dictionary would be `pokeconstants.MOVES['ThunderShock']` which should evaluate to a `PokeMove` containing all the relevant data for the ThunderShock move.

Note: In writing this method, you will have to deal with the fact that some entries in moves.txt are blank, as they are not applicable for that move. When creating the corresponding `PokeMove`, you will have to replace the blank/empty string with `None` as an argument.
Reminder: All the data you read out of `moves.txt` will be a string, but some of `PokeMove`s attributes/parameters are integers.
Warning: Remember not to try and turn the header row into a `PokeMove`!

read_pokemon(filename)

This method will be used to read `pokedata.txt` and create/return a dictionary containing all the data about the Pokémon species available in the game.

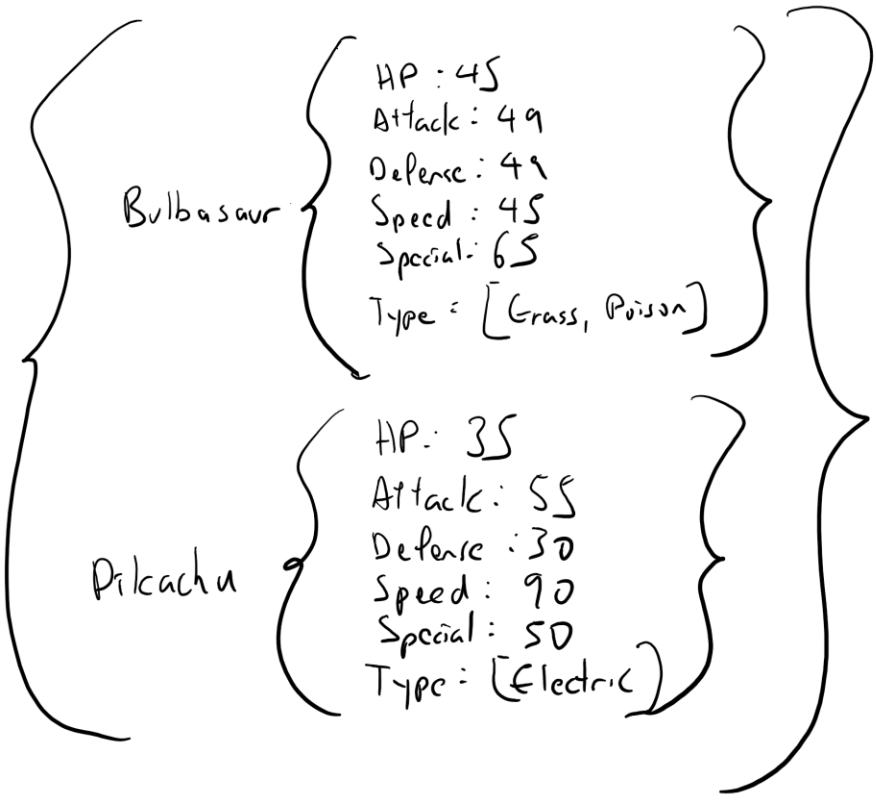
1. Has 1 parameter, which should typically receive the argument `'pokedata.txt'`.

2. Returns a 3-level nested dictionary mapping the names of every Pokémon in `pokedata.txt`, as a string, to another dictionary mapping a stat or type string to the relevant data.
    - **Note**: This is complicated, so the desired dictionary structure in more detail further down.
3. When completed, this function will be called successfully in `pokeconstants.py`, on line 12 of the starter code, and make all Pokémon species data easily accessible from anywhere in the program in a dictionary named `pokeconstants.POKE_DATA` This dictionary is then accessed in the `Pokemon` class' various methods to, for example, calculate the Pokemon's stats, or in the `simulate_battle()` to get a `Pokemon`'s types.

    Dictionary structure
    This dictionary needs to contain all the relevant data for a Pokémon species, which is why it is so complex. In practice though, its structure and use are rather straightforward, though difficult to describe. As such, here is an example of what this dictionary should look like, if only Bulbasaur was stored within it:

    ```
    {'Bulbasaur': {'HP': 45, 'Attack': 49, 'Defense': 49, 'Speed': 45, 'Special': 65,
    'Type': ['Grass', 'Poison']}}
    ```

    or in an alternative representation, with both Bulbasaur and Pikachu stored:



    You will notice that this data corresponds exactly with line #2 (and #25) of `pokedata.txt`. The key is we're restructuring the data in a way that will make it easy to access later. For example, if we wanted the base Attack value of any Bulbasaur, the following call would suffice:

```
pokeconstants.POKE_DATA['Bulbasaur']['Attack']
```

which would return an integer value representing Bulbasaur's base Attack value (`49`), and similarly for the other 4 stats. On the other hand, the following call:

```
pokeconstants.POKE_DATA['Bulbasaur']['Type']
```

should return a length-2 list of Bulbasaur's types (`['Grass', 'Poison']`). Again, this makes it easy to look up the types of any Pokémon anywhere in the program.

Note: In writing this method, you will have to deal with the fact that some Pokémon have one type while others have two. This will change the length of the list stored in the dictionary.
Reminder: All the data you read out of `poke_data.txt` will be a string, but some of the stored values are integers.
Warning: Remember not to try and turn the header row into a Pokémon!

`Pokemon`

The `Pokemon` class represents a Pokémon. What more do you want me to say?

Attributes

The `Pokemon` class should have the following attributes:

1. Name
   - The name of the Pokémon, as a string.
2. Level
   - The level of the Pokémon, as an integer in the range [1, 100]
3. IVs
   - A dictionary of the Pokémon's IV stats.
   - This dictionary should map the stat's name, as a string, to the corresponding IV stat value for this Pokémon.
   - For example, using this dictionary, we should be able to look up a Pokémon's Attack IV value using the code *`<pokemon_variable>.<iv_attribute>`*`['Attack'].`
4. EVs
   - A dictionary of the Pokémon's EV stats.
   - This dictionary should map the stat's name, as a string, to the corresponding EV stat value for this Pokémon.
   - For example, using this dictionary, we should be able to look up a Pokémon's Special EV value using the code *`<pokemon_variable>.<ev_attribute>`*`['Special'].`
5. Moves
   - A list of the Pokémon's moves, stored as `PokeMove`s.
   - **Note**: Each Pokémon must know from 1 to 4 moves.
   - **Note**: All Pokémon should work with the same 165 `PokeMove`s from `pokeconstants.py`. That is to say, if a Pikachu and an Electabuzz both know ThunderShock, they should both use the same ThunderShock `PokeMove`, rather than have two separate `PokeMove` objects, to save memory.
6. Current HP
   - The Pokémon's current health total, as an integer.

Note: A Pokémon's *base stat values* are **not** stored in the `Pokemon` object itself. This is because a Pokémon's base stat values are the same for all members of its species, so they are stored centrally in `pokeconstants.py` instead. Only a Pokémon's IVs and EVs are stored within each `Pokemon` object.

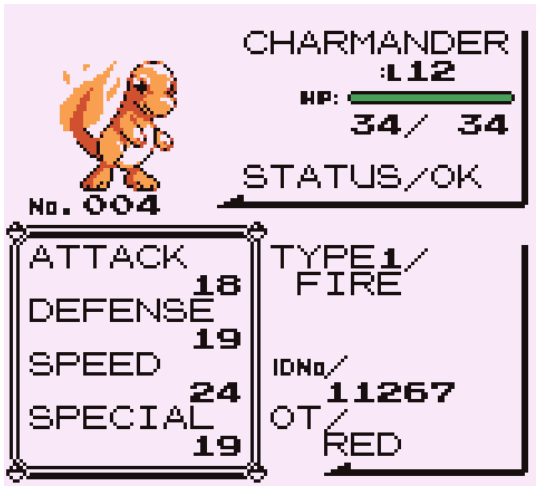
Methods

The `Pokémon` class should have the following methods:

1. `__init__(self, ...)`
    - This method should have 5 non-self parameters:
        - The Pokémon's name/species.
        - The Pokémon's level.
        - The Pokémon's IVs, as a list of 5 integers in the order Attack, Defense, Speed, Special, HP.
        - The Pokémon's EVs, as a list of 5 integers in the order Attack, Defense, Speed, Special, HP.
        - The Pokémon's known moves, as a list of 1 to 4 strings.
    - The arguments passed to these parameters should be saved in the Pokémon's attributes, after being transformed into the right format/types.
        - **Note**: The initializer accepts the **names** of the Pokémon's moves, but the class attribute stores the moves as PokeMove objects. You will need to access `pokeconstants.MOVES` to do this.
    - **E.g.** Sample calls for this initializer are found in the starter code for `pokemon_main.py`, on lines 98 and 101 where a Pikachu and a Blastoise are instantiated.
2. `calc_stat(self, ...)`
    - This method should essentially be the `calc_stat(...)` function previously found in `pokemon_main.py`, but now rewritten into a method.
    - Has a single non-self parameter: the name of the stat being calculated as a string, one of `'Attack'`, `'Defense'`, `'Speed'`, `'Special'`, or `'HP'`.
    - Return the specified stat value of this Pokémon, calculated using its level, corresponding IV, corresponding EV, and its species' corresponding base stat value.
        - **Note**: To access its species' corresponding base stat value, you will need to access `pokeconstants.POKE_DATA`.
3. `choose_random_move(self, ...)`
    - This method should essentially be the `choose_random_move(...)` function previously found in `pokemon_main.py`, but now rewritten into a method.
    - Has no non-self parameters.
    - Returns a `PokeMove`, randomly selected from one of this `Pokemon`'s known `PokeMoves`.
4. `__str__(self)`
    - This method should return a string with the Pokémon's name and level. In theory, this method could be used to create an interface that looks something like this:

(Src: http://www.thepokedex.com/pokemon-games/pokemon-blue-pokemon-red/pokemon-red-screenshots-pokemon-blue-screenshots/pokemon-blue-and-red-screenshots-3/ c/o Google Image Search)

5. `__repr__(self)`
    - This method should return a string with the `Pokemon`'s name, level, type, current HP, maximum HP, calculated stat values, and known moves. In theory, this method could be used to create an interface that looks something like this:



(Src: https://imgur.com/gallery/cSSqG/comment/396971922 c/o Google Image Search)

- **Hint**: You may wish to look at the `Student`, `Course`, and `Section` `__repr__`s provided in Extensive Student Class Example linked in the Week 11 notes for an example of how `__repr__`s can look: https://repl.it/@blaw/CS127-F20-Extensive-Student-Class-Example
- **Note**: To access the Pokemon's type, which is not stored in the `Pokemon` object, you will need to access `pokeconstants.POKE_DATA`.

The Simulation

Once your `PokeMove` and `Pokemon` classes, and the IO functions are completed, the entire Pokemon program should be functional once again. This extensive code did not improve or enhance the program in any way, but by reorganizing the code and incorporating a lot of new data, you will have made it possible to continue on with new additions to the program, to be completed as bonus marks.

Marking Scheme

Note: The correctness of the program remains worth only 50% of the overall project grade, despite the additional complexity. Keep this in mind when you are planning how to spend your time and energy on this project!

Criterion	% of Grade	Excellent (100%)	Adequate (80%)	Poor (60%)	Not Met (0%)
Correctness - `PokeMove`	15%	No errors, program works correctly and meets specifications	Minor details of specifications violated, and/or program functions incorrectly for some inputs	Significant details of specifications violated; program often exhibits incorrect behaviour	Program only functions correctly in very limited cases or not at all
Correctness – `io_functions`	15%*	No errors, program works correctly and meets specifications	Minor details of specifications violated, and/or program functions incorrectly for some inputs	Significant details of specifications violated; program often exhibits incorrect behaviour	Program only functions correctly in very limited cases or not at all
Correctness – `Pokemon`	20%*	No errors, program works correctly and meets specifications	Minor details of specifications violated, and/or program functions incorrectly for some inputs	Significant details of specifications violated; program often exhibits incorrect behaviour	Program only functions correctly in very limited cases or not at all
Readability	25%	No errors, code is clean, understandable, and well-organized	Minor issues with consistent indentation, use of whitespace, and/or general organization	At least one major issue with indentation, whitespace, variable names, or organization	Major problems with at least three or four of the readability subcategories
Documentation	25%	No errors, code is well-commented	One or two places that could benefit from comments are missing them, or code is overly commented	File header missing, complicated lines or sections of code without descriptive comments	No file header or comments present.

Bonus

Below are a list of suggested possible bonus features, as well as the grade % allocated for each. You may attempt other bonus features yourself for credit, but you are **strongly** advised to contact Brian beforehand to verify that these features are achievable and to determine what % these features will be worth.

Note: Only up to 50% extra bonus credit may be received.

Bonus Item #1 – Bigger Battles: 10%

In the Pokémon games, battles can be between teams of up to 6 Pokémon. For this bonus feature, Implement larger Pokémon teams rather than just a 1vs1 battle between Pikachu and Blastoise.

These teams may be of your own creation in the program. They do not need to be distinct (i.e. you could have 6 Pikachu battling 6 Blastoise). You may determine how the Pokémon swap in after another is defeated at your discretion.

Bonus Item #2 – User-Selected Moves: 5%

During the battle, prompt the user to choose which move each Pokémon should use of the 1-4 that they know. The prompt may request a numeric value or the move's name, at your discretion.

You do not need to handle the case of users providing bad input.

Note: Your `Pokemon.choose_random_move()` method will still be evaluated even if it is not used due to your implementation of this feature.

Bonus Item #3 – Status Effect #1 - Paralysis: 10%

Paralysis is one of many status effects in the Pokémon games. When a Pokémon is paralyzed, with the text, "*<pokemon>* `is paralyzed! It may not attack!`" its speed is cut by 75% and each turn, it has a 25% chance of skipping its turn (with the text output "*<pokemon>* `is paralyzed! It can't move!`"

See https://bulbapedia.bulbagarden.net/wiki/Paralysis_(status_condition)#Effect for details.

This feature will require you to implement paralyzing moves like ThunderShock, Thunderbolt, and Thunder Wave, and also the status effect on Pokémon who are paralyzed in combat.

Bonus Item #4 – Status Effect #2 - Poison: 10%

Poison is one of many status effects in the Pokémon games. When a Pokémon is poisoned, "*<pokemon>* `was poisoned.`", it loses 1/16th of its max health at the end of its turn, with the text "*<pokemon>*`'s hurt by poison!`"

See https://bulbapedia.bulbagarden.net/wiki/Poison_(status_condition)#Effect for details.

This feature will require you to implement poisoning moves like Poison Powder, Poison Sting, and Smog, and also the status effect on Pokémon who are poisoned in combat.

Note: Technically, Poison-type Pokémon can't be poisoned, but you won't have to account for that detail if you don't want to.
Note: This does not require you to implement the "badly poisoned" status condition caused by Toxic.

Bonus Item #5 – Status Effect #3 - Sleep: 10%

Sleep is one of many status effects in the Pokémon games. When a Pokémon is first put to sleep, with the text "`<pokemon> fell asleep!`" a random number in the range [2, 7] is generated. The sleeping Pokemon then loses its next 2-7 turns, with the text "`<pokemon> is fast asleep.`" On the last turn it misses, it wakes up with the text, "`<pokemon> woke up!`"

See https://bulbapedia.bulbagarden.net/wiki/Sleep_(status_condition)#Effect for details.

This feature will require you to implement sleeping moves like Sleep Powder, Hypnosis, and Spore, and also the status effect on Pokémon who are asleep in combat.

Note: Rest also puts Pokémon to sleep, but you don't have to (fully) implement Rest for this bonus.
Note: Dream Eater interacts with Pokémon who are asleep, but you don't have to implement Dream Eater for this bonus.

Bonus Item #6 – Status Effect #4 - Burn: 5%

Burn is one of many status effects in the Pokémon games. When a Pokémon is burned, with the text "`<pokemon> was burned!`", it loses 1/16 of its maximum health at the end of each turn, with the text "`<pokemon>'s hurt by the burn!`" It also does 50% less damage with physical attacks.

See https://bulbapedia.bulbagarden.net/wiki/Burn_(status_condition)#Generation_I for details.

This feature will require you to implement burning moves like Ember, Flamethrower, and Fire Blast, and also the status effect on Pokémon who are asleep in combat.

Note: Technically, Fire-type Pokémon can't be burned, but you won't have to account for that detail if you don't want to.

Bonus Item #6 – Status Effect #5 - Freeze: 5%

Freeze is one of many status effects in the Pokémon games. When a Pokémon is frozen, with the text "`<pokemon> was frozen solid!`" it loses all its turns for the rest of the combat, with the text "`<pokemon> is frozen solid!`"

See https://bulbapedia.bulbagarden.net/wiki/Freeze_(status_condition)#Generation_I for details.

This feature will require you to implement freezing moves like Ice Beam and Blizzard, and also the status effect on Pokémon who are asleep in combat.

Note: Technically, a frozen Pokémon can be unfrozen when hit by a Fire-type move, but you won't have to account for that detail if you don't want to.
Note: Technically, Ice-type Pokémon can't be frozen, but you won't have to account for that detail if you don't want to.

Bonus Item #7 – Accuracy: 5%

Pokémon moves have accuracy values that may cause them to miss if less than 100, with the text, "`It missed!`"

This feature will require you to implement Accuracy by having Pokémon moves with Accuracy less than 100 to miss, based on a percentage chance. (I.e. a move with 80 Accuracy should miss 20% of the time.)

Note: Technically, there's a glitch in the Gen I games that cause moves with 100 Accuracy to miss 1/256 times, due to how Accuracy was calculated. We'll ignore that for this feature.

Bonus Item #8 – Swapping Pokemon: 10%

Requires: Bonus Items #1 & #2

When prompted to select a move for the Pokémon to use, the user is also prompted to, if they want, swap Pokémon to another Pokémon on the team.

If the user chooses to switch Pokémon, some text output should be displayed showing the current status of the team (the Pokémon's names and their current HPs, and any status effects if implemented). The user is then prompted to select a Pokémon to switch in, using a numeric value (as multiples of the same Pokémon may be on a team).

You do not need to handle the case of users providing bad input, but you should provide a way to cancel out of swapping, in case there are no valid Pokémon to swap to!

Note: Swapping in a Pokémon usually costs a turn, but you will not be required to implement that for this bonus (though you may if you want).