PROJECT DOCUMENTATION FINAL PROJECT COURSE PROGRAMMING WITH PYTHON (PATRICK PRIEWE)

07.11.2024

Inhalt

Project Goals and Requirements

The project relies on the exercise project for a console based „pokemon“ game (text based via console) that was created beforehand.

The exercise project already had a functionning game logic. The core requirements for the course's final project were:

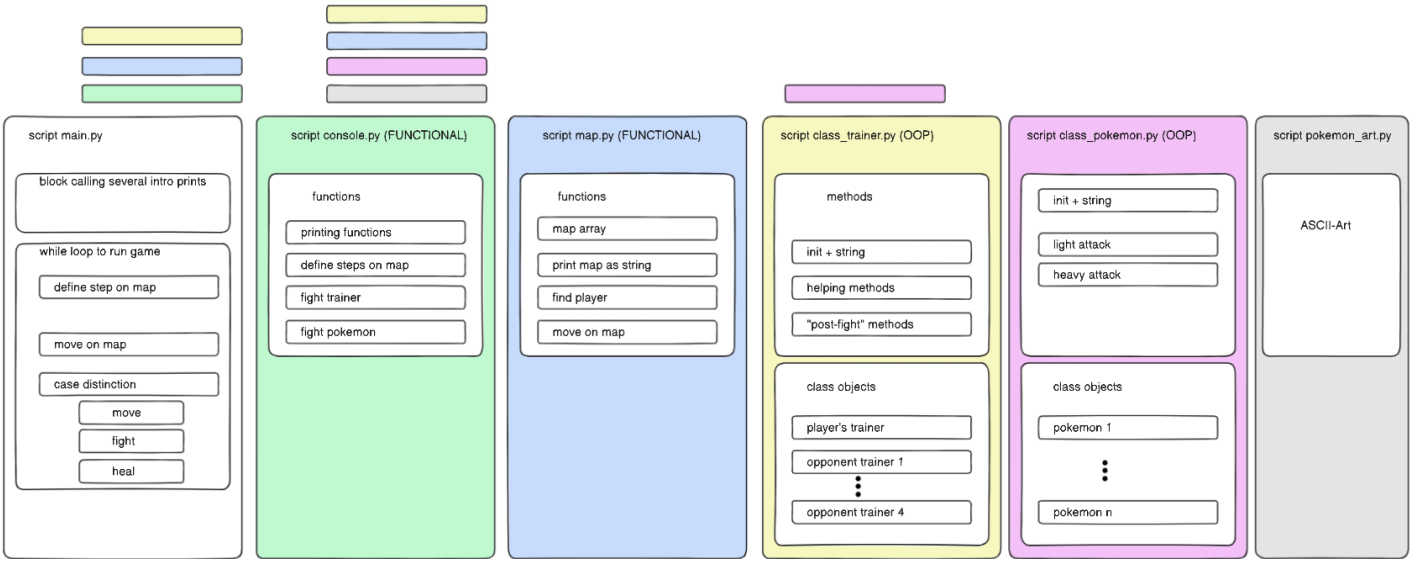
- Transfer all remaining scripts that contain functions into object oriented structures („OOP“, i.e. classes, class methods, class objects)
    - This affects the following structures of the exercise project:
        - script „console.py“
        - script „map.py“
- Refactor (simplify and structurally decompress) code structure of the main fight functions in the classes „Trainer“ and „Pokemon“
    - This affects the scripts „class_trainer.py“ and „class_pokemon.py“
- Create a UI using the PyCharm terminal
    - Beforehand, the console was used as UI
    - Thus, the introduction of terminal functions into the project was neccessary

1 Transforming Structure of the Exercise Project to full OOP

In the following, we layout the structure of the exercise project, in order to later show, where we had to apply changes.

The exercise project's structure (scripts) was roughly as shown below:

Schematic Structure of the Exercise Project (before Transformation to OOP)



The larger „cards“ represent the scripts. The smaller (bar-shaped) cards above represent the required imports that were neccessary in each script. The colors of the small bars above the large cards indicate, which script was imported to the script (e.g., class_trainer imported from class_pokemon, indicated by the magenta bar above class_trainer).

A peculiar aspect of this structure were the huge fight functions (fight_trainer and fight_pokemon) in the console script, that had to be broken down (more on that later). Due to the mixture of OOP and functional programming, the fight mode was overly complicated:

- It contained lots of unneccessary code:
  - executing several very similar sequences multiple times for the player's turn and the opponent's turn
  - Input validations with similar structures were scattered all over the code within the two functions (and in other scripts as well)
- Generally, the code within the two fighting functions was far from readable.

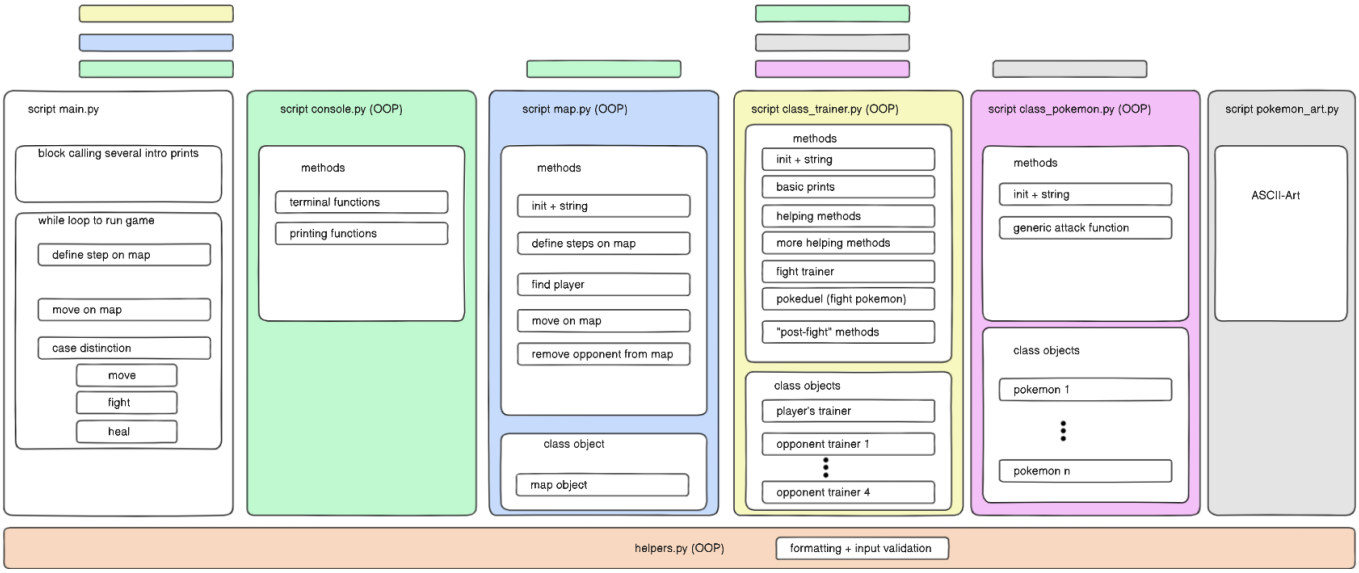
Transforming the functional scripts into OOP was generally not a huge deal by itself, however:

- It was very challenging to maintain the correct passing of parameters and results to the corresponding calling functions, as the „fight trainer“ method (in the green card above) was intertwined to the pokemon class and the trainer class (yellow and magenta above).

Though, the transformation could be done with no loss of functionality or cause of errors in the running program (the program was thoroughly tested after the transformation to OOP). For the resulting structure, see the next chapter.

Schematic Structure of the Final Project (after Transformation to OOP)

For a schematic view of the project after the transformation to full OOP, see the following graphic. For colors and objects in the graphic, the same logic holds as in the last chapter.



When comparing the two graphics, see the following main differences for each script:

- main.py:
  - no major differences on structure and content.
- console.py:
  - shifting the fight functions out of the script
  - reducing the required imports/dependencies (no more imports of other project scripts)
  - adding terminal functions for allowing better UI in terminal usage
- map.py:
  - new map object
  - step definition and movement on map now included in this class (before: console.py)
  - additional method to remove (defeated) opponent from map (this functionality was buried in the console functions before)
  - additional import from console, mostly to make use of terminal functions/UI-related functions in console
- class_trainer.py:
  - more helping methods (to reduce the size and complexity in the fighting functions)
  - additional import from console, mostly to make use of terminal functions/UI-related functions in console
  - additional import from pokemon_art (as prints in fight methods were transfered from console.py to class_trainer.py, terminal and UI-related methods remain in dconsole.py)
- class_pokemon.py
  - light attack and heavy attack were merged into a generic attack function.
- new class in helpers.py:
  - two additional methods for formatting (UI-related) and input validation (to reduce/simplify structures in fight methods, but also used on different parts of the final project).

Summary:

- The transformation was successfull (no errors, better structure)
- Generally, the structures are more logical and follow the functionality

2 Simplifying the Structure of the Fight Methods

This chapter is about the requirement to simplify the code structure of the fighting methods, which was the main challenge of the final project (both in time and complexity aspects).

Attack logic/structure in final project

After breaking down the fight logic and adding several helping methods, we were able to reduce the complexity (and abundancy) of the code to the following structure:

fight_trainer („outer“ fight method in class Trainer)

- # step 1 - see opponent's stats and make **fight decision**
  - CALL **new helping method**
- # step 2 - if fight is on, print **fight intro**
  - CALL **new helping method**
- # step 3 - outer loop: runs until all of the player's pokemon are defeated (or fight is fled from within loop)
  - # step 3.1 - player **chooses undefeated pokemon**
    - CALL **new helping method**
  - # step 3.2 - inner loop for player's chosen pokemon:
    - # runs until player's pokemon is defeated, player flees, or opponent has no more pokemon left
    - # opponent keeps deploying his pokemon in inner loop
    - WHILE player’s pokemon alive and opponent has pokemons left:
      - Opponent deploys his pokemon one after another (if defeated)
      - CALL **pokeduel_main** to **pair dueling pokemon**
  - # step 3.3 - multiple checks:
    - # if opposing trainer was defeated, print message and return his id (or a string, if final boss)
    - # if "regular" opponent, level up player's trainer
- # step 4: if fight was lost return

pokeduel_main („inner“ fight method in class „Trainer“)

- While-loop: runs until player flees or one of the dueling pokemon is defeated
  - # step 1 – player’s move
    - # step 1.1 – **get attack mode**
      - CALL **new helping method**
    - # step 1.2 – **execute attack**
      - CALL **new generic attack method** (no case distinction light/heavy neccessary)
    - # step 1.3 – check if opponent’s pokemon was defeated by attack
  - # step 2 – opponent’s move
    - # step 2.1 – **get attack mode**
      - CALL **new helping method**
    - # step 2.2 – **execute attack**
      - CALL **new generic attack method** (no case distinction light/heavy neccessary)
    - # step 2.3 – check if player‘s pokemon was defeated by attack

The new (helping) method calls above that are highlighted with **bold green text**, as well as the merging of heavy and light attack into a **generic attack method** allowed us to increase the readability (and maintainability!) of the code drastically.

Concerning the inner fight method:

- At first glance, the similar structure between the player's move and the opponent's move seems a bit abundant.
- However, to keep the structure like this, had the following advantages:
    - It allows us to include a flight option in step # 1.1 (which is not available for the opponent in step # 2.1)
    - It allows us to separately count the use of heavy attacks for both the player and the opponent in # step 1.2 / # step 2.2 (and to print different pokemon_arts respectively)
    - It allows us to implement different usage of terminal clearance in # step 1.3 / # step 2.3, which is crucial for the sequence of screen actualizations in the UI

Improving the User Interface

Last but not least, we were able to improve the UI as follows:

- In the exercise project, all prints were made in the console (where the game was run). This means, that all prints were rolling up, when a nex print was added at the bottom.
    - In the final project (running in the terminal), we apply a clearing method from the package os to simulate a static screen for sequences of the program
- In the exercise project, there was no coloring
    - In the final project, we included the package colorama to colorize and style important text fragments in the terminal.