CSSE1001/7030

Semester 2, 2016
Assignment 2
Version 1.0.4b
10 marks

Due Friday 30 September, 2016, 21:30

1 Introduction

For this assignment, you will be writing code that supports a simple Pokemon finding game. The basic idea of the application is that you explore multiple levels in search for Pokemon, which are then registered in your Dex. Rather than using functions, like assignment 1, you will be using Object-Oriented Programming (OOP). Further, you will be using the Model View Controller (MVC) design pattern. Your task is to write the Model. The View and Controller are provided, along with support code.

As is typical with projects where more than one person is responsible for writing code, there needs to be a way of describing how the various components interact. This is achieved by defining an Application Programming Interface (API). For this assignment, you must implement your classes according to the API that has been specified, which will ensure that your code will interact properly with the supplied View/Controller code.

One benefit to adhering to MVC is that the model can be developed and tested independently of the view or controller. It is recommended that you follow this approach. This means testing your model iteratively as you develop your code.

2 Assignment Tasks

2.1 Download files

The first task is to download `a2.py` and `a2_files.zip`. The file `a2.py` is for your assignment. **Do not modify the assignment file outside the area provided for you to write your code**.

2.2 Important Definitions

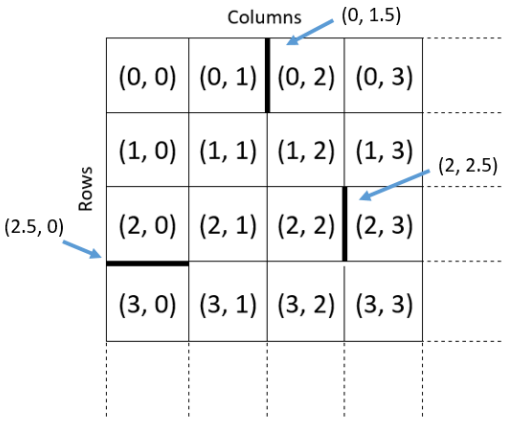
While Pokemon is an irregular plural (meaning that its plural takes the same form as its singular, like sheep and fish), for the purposes of this assignment, multiple Pokemon are referred to as Pokemons for clarity.

2.2.1 Positions & Coordinates

A **position** is represented by a (row, column) pair of numbers.

A **cell position** is a position where the row and column are both integers. It represents the position of where an object in the game could be located.

A **wall position** is a position where either the row or column value is a float ending in .5 and the other is an integer. The .5 represents that the wall is located at a boundary betweeen two cell positions for that row or column.



2.2.2 Expecting, Registering, & Catching

Catching a Pokemon refers to the player adding it to their collection. Whilst moving around the game world, the player catches a Pokemon by moving onto a cell in which that Pokemon exists. When a Pokemon is caught, it is removed from the game world.

A Pokedex, henceforth abbreviated as Dex, maintains a registry of Pokemon that the player is **expected** to catch. A Pokemon is registered in a Dex when the player catches that Pokemon. **Registering** a Pokemon that has already been registered has no effect.

2.2.3 Game Data

Game data can be loaded using either `load_game_file(file)` or `load_game_url(url)` from the support file. These functions will raise errors if the file/url do not exist or provide invalid JSON, as specified in their docstring comments. Game data consists of the following structure, comments added for clarity:

```
{
    levels: [                  # List of levels in game
        {
            terrain: str,      # Name of the terrain type
            rows: int,         # Number of rows
            columns: int,      # Number of columns
            player: (int, int), # Player's starting position
            pokemons: [        # List of pokemon existing in level
                {name: str, position: (int, int)},
                ...
            ],
            walls: [           # List of walls existing in level
```

```
                (int, int),
                ...
            ]
        },
        ...
    ]
}
```

For example, game data is a dictionary, containing `"levels"` as a key, whose value is a list of level data. Further, level data is a dictionary with multiple keys.`"terrain"` is one key that has a value that is a string indicating the name of the terrain. The `"walls"` is another key that has a value that is a list of pairs of integers, each representing the location of a wall in the level.

You may assume that all pokemon names in a game file are valid, along with the terrain.

Additional game data can be randomly generated on the course website. For example:
`http://csse1001.uqcloud.net/2016s2a2/pokemon-map.json?levels=4&rows=12&columns=12`

2.3 Write the code

There are several classes you need to write and these are described below. It is highly recommended that you review the support file, `a2_support.py`, before writing your code, as this contains many useful constants and functions. **Do not use global variables in your code.**

N.b. The output for some of the example code is display using `__repr__` methods. These lines have been marked with `# displayed using __repr__`. While you are not required to implement these methods, it is recommended to do so, as it will make debugging simpler. You may find the format strings format strings in `a2_support.py` helpful (e.g. `PLAYER_REPR_FORMAT`).

2.3.1 Commenting

Each class and method that you write must have a suitable docstring comment, as specified in the course notes. See `http://csse1001.uqcloud.net/notes/commenting`

2.3.2 The GameObject Class

`GameObject` is the superclass for objects that exist in the game grid.

Instances of `GameObject` are to be constructed with `GameObject(name, position)`, where `name` is a string representing the name of the object and `position` is a grid position.

Further, the following methods are to be implemented:

set_position(self, position) Sets the position to `position`, which either is a cell position or `None`.

get_position(self) Returns the current position of the instance.

set_name(self, name) Sets the name to `name`.

get_name(self) Returns the name of the instance.

__str__(self) Returns a human readable representation of this instance, according to `GAME_OBJECT_FORMAT` in the support file.

2.3.3 The Pokemon Class

`Pokemon` inherits from `GameObject` and is used for managing the name and position of Pokemon within the game.

Instances of `Pokemon` are to be constructed with `Pokemon(name, position, terrain)`, where `name` and `position` are as they are for `GameObject`, and `terrain` is a string representing the terrain in which the Pokemon exists.

Further, the following methods are to be implemented:

set_terrain(self, terrain) Sets the terrain to `terrain`.

get_terrain(self) Returns the terrain of the instance.

__str__(self) Returns a human readable representation of this instance, according to `POKEMON_FORMAT` in the support file.

```
>>> mew = Pokemon("Mew", (20, 20), "Mountain")
>>> str(mew)
'Mew @ (20, 20) from Mountain'
>>> mew.get_terrain()
'Mountain'
>>> mew.set_terrain("Grass")
>>> str(mew)
'Mew @ (20, 20) from Grass'
```

2.3.4 The Wall Class

`Wall` inherits from `GameObject` and implements no additional functionality. It is used for representing a wall in the game.

2.3.5 The Player Class

`Player` inherits from `GameObject` and is used for representing a player within a grid.

Instances are to be constructed with `Player(name)`, where `name` is as it is for `GameObject`.

A Player must contain the following:

- A Dex to register all the Pokemon that the Player encounters (see section 2.3.6).
- A list of Pokemon that the Player has caught, in the order they were caught.

The following methods are to be implemented:

get_pokemons(self) Returns a list of all Pokemon that this Player has caught, in the order they were caught.

get_dex(self) Returns the Player's Dex.

reset_pokemons(self) Resets all the Pokemon caught by this Player, including their Dex.

register_pokemon(self, pokemon) Catches the pokemon and adds to the Player's Dex, where `pokemon` is a `Pokemon`, provided it is expected by the Player's Dex. Otherwise, this method should raise an `UnexpectedPokemonError`.

__str__(self) Returns a human readable representation of this instance, according to `PLAYER_FORMAT` in the support file.

```
>>> mew = Pokemon("Mew", (20, 20), "Grass")
>>> d1 = Pokemon("Dragonite", (1, 1), "Mountain")
>>> d2 = Pokemon("Dragonite", (1, 3), "Mountain")
>>> brock = Player(DEFAULT_PLAYER_NAME)
>>> str(brock)
'Ash @ None has caught 0'
>>> brock.set_name("Brock")
>>> brock.set_position((1,1))
>>> brock.register_pokemon(mew)
Traceback (most recent call last):
```

```
    ... # truncated for brevity
a2_support.UnexpectedPokemonError: Mew is not expected by this Dex.
>>> brock.get_dex().expect_pokemons(['Mew', 'Dratini', 'Dragonair', 'Dragonite'])
>>> brock.register_pokemon(mew)
>>> brock.register_pokemon(d1)
>>> brock.register_pokemon(d2)
>>> str(brock)
'Brock @ (1, 1) has caught 3'
>>> for pokemon in brock.get_pokemons(): print(pokemon)
Mew @ (20, 20) from Grass
Dragonite @ (1, 1) from Mountain
Dragonite @ (1, 3) from Mountain
>>> print(brock.get_dex())
2 Registered: Dragonite, Mew
2 Unregistered: Dragonair, Dratini
>>> brock.reset_pokemons()
>>> str(brock)
'Brock @ (1, 1) has caught 0'
>>> print(brock.get_dex())
0 Registered:
0 Unregistered:
```

2.3.6 The Dex Class

The `Dex` class manages a registry of Pokemon that have been encountered. For the `Dex` class, pokemon refers only to the name of a pokemon, and not an instance of the Pokemon class.

Instances are to be constructed using `Dex(pokemon_names)`, where `pokemon_names` is a list of pokemon names to be expected by this Dex. In order for a Dex to be complete, all the Pokemon that are expected must also be registered.

A Dex must contain a dictionary whose keys are pokemon names that are expected by this Dex, and whose values indicate whether the corresponding pokemon is registered in this Dex (True: registered; False: unregistered).

Further, the following methods must be defined for the Dex class.

expect_pokemons(self, pokemon_names) Instructs the Dex to also expect all pokemon in the list of `pokemon_names` (that are not already expected).

expect_pokemons_from_dex(self, other_dex) Instructs the Dex to also expect all pokemon that `other_dex` expects (that are not already expected).

register(self, pokemon_name) Registers the pokemon (name) in the Dex. Returns True if the pokemon was already registered, else False. This method raises an `UnexpectedPokemonError` if the

pokemon is not expected by this Dex.

register_from_dex(self, other_dex) Registers each pokemon from another Dex, `other_dex`, provided it is expected by this Dex and registered in the other Dex. This method must never raise an `UnexpectedPokemonError`.

get_pokemons(self) Returns a list of (name, registered) pairs for each pokemon expected by this Dex, where `name` is the name of the pokemon, and `registered` is True if the pokemon is registered, else False. This list must be sorted alphabetically by `name`.

get_registered_pokemons(self) Returns an alphabetically sorted list of names of pokemon registered in this Dex.

get_unregistered_pokemons(self) Returns an alphabetically sorted list of names of pokemon unregistered in, but expected by, this Dex.

__len__(self) Returns the total number of pokemon expected by this Dex.

__contains__(self, name) Returns True iff pokemon with `name` is registered in this Dex, else False.

__str__(self) Returns a human readable string representation of this Dex, according to `DEX_FORMAT` in the support file. The string contains two lines, which are of the format "number status (capitalised): pokemon names separated by comma, sorted alphabetically", with the first being for registered and the second being for unregistered. For example, if Squirtle and Charmander were registered, but Bulbasaur was unregistered:

```
2 Registered: Charmander, Squirtle
1 Unregistered: Bulbasaur
```

```
>>> dex1 = Dex(['Lugia', 'Mewtwo', 'Latios', 'Latias'])
>>> str(dex1)
'0 Registered: \n4 Unregistered: Latias, Latios, Lugia, Mewtwo'
>>> dex1.register('Latios')
False
>>> dex1.register('Latios')
True
>>> dex1.register('Latias')
False
>>> for pokemon in dex1.get_unregistered_pokemons(): print(pokemon)
Lugia
Mewtwo
>>> for pokemon in dex1.get_registered_pokemons(): print(pokemon)
Latias
Latios
>>> dex2 = Dex(['Entei', 'Suicune', 'Raikou', 'Lugia', 'Ho-Oh'])
```

```
>>> len(dex2)
5
>>> 'Lugia' in dex2
False
>>> dex2.register('Lugia')
False
>>> 'Lugia' in dex2
True
>>> dex2.register('Suicune')
False
>>> dex1.register_from_dex(dex2)
>>> str(dex1)
'3 Registered: Latias, Latios, Lugia\n1 Unregistered: Mewtwo'
>>> dex1.expect_pokemons_from_dex(dex2)
>>> dex1.register_from_dex(dex2)
>>> print(dex1)
4 Registered: Latias, Latios, Lugia, Suicune
4 Unregistered: Entei, Ho-Oh, Mewtwo, Raikou
```

2.3.7 The Level Class

The `Level` class manages data pertaining to an individual level in the game. A `Level` is considered complete when its `Dex` has no unregistered pokemon. **N.b.** It is possible to complete a level without having caught all pokemon that exist in that level, since there may be duplicate Pokemon.

Instances are to be constructed with `Level(player, data)`, where `player` is an instance of Player, and data is a dictionary of a single level's data. When initialised, a Level should instruct the player's Dex to expect all the Pokemon that could be encountered in the current level. If a level contains an invalid position (player start, pokemon, wall, etc.), it must raise an `InvalidPositionError`.

A Level must contain the following:

- A dictionary whose keys are cell positions and whose values are Pokemon instances that exist at the corresponding position.
- A dictionary whose keys are wall positions for every wall that exists in the level (value can be anything). This includes walls positions for each boundary wall (on the edge of the grid), which are not necessarily specified in the level data.
- An instance of Dex that expects exactly all of the pokemon in this level.

Further, the following methods must be defined:

get_size(self) Returns the size of the level grid.

get_terrain(self) Returns the terrain type of this level.

get_dex(self) Returns the Dex for this level.

get_starting_position(self) Returns the player's starting position for this level.

is_obstacle_at(self, position) Returns True iff an obstacle exists at given `position`, else False.

get_obstacles(self) Returns a list of positions of all obstacles (walls) that exist in this level, including boundary walls.

get_pokemons(self) Returns a list of all Pokemon that exist in this level.

get_pokemon_at(self, position) Returns the Pokemon that exists at the given `position`, else None.

catch_pokemon_at(self, position) Catches and returns the Pokemon that exists at the given position. If no pokemon exists at the given position, this method raises an `InvalidPositionError`.

is_complete(self) Returns True iff this `Level` is complete, else False.

```
>>> player = Player(DEFAULT_PLAYER_NAME)
>>> data = load_game_file('game1.json')
>>> level = Level(player, data['levels'][0])
>>> level.get_size()
(10, 10)
>>> level.get_starting_position()
(1, 1)
>>> level.get_terrain()
'Ice'
>>> print(level.get_dex())
0 Registered:
4 Unregistered: Bulbasaur, Charmander, Pikachu, Squirtle
>>> str(player)
'Ash @ None has caught 0'
>>> level.get_obstacles()
[(5, 9.5), (6, 9.5), (9, -0.5), (9.5, 1), (3, -0.5), ...] # truncated for brevity
>>> level.is_obstacle_at((5, 9.5))
True
>>> level.is_obstacle_at((0.5, 0))
False
>>> for pokemon in level.get_pokemons(): print(pokemon) # order may differ
Bulbasaur @ (2, 7) from Ice
Charmander @ (7, 7) from Ice
Squirtle @ (7, 2) from Ice
Pikachu @ (2, 2) from Ice
>>> level.get_pokemon_at((2, 6))
```

```
>>> level.get_pokemon_at((2, 7)) # displayed using __repr__
Pokemon('Bulbasaur', (2, 7), 'Ice')
>>> level.catch_pokemon_at((2, 7)) # displayed using __repr__
Pokemon('Bulbasaur', (2, 7), 'Ice')
>>> level.get_pokemon_at((2, 7))
>>> level.is_complete()
False
>>> for pokemon in level.get_pokemons(): print(pokemon) # order may differ
Charmander @ (7, 7) from Ice
Squirtle @ (7, 2) from Ice
Pikachu @ (2, 2) from Ice
>>> for pokemon in level.get_pokemons(): level.catch_pokemon_at(pokemon.get_position())
... # truncated for brevity
>>> level.is_complete()
True
>>> str(player)
'Ash @ None has caught 4'
```

2.3.8 The Game Class

The `Game` class manages data pertaining to an entire game. Its constructor requires no arguments.

- An instance of the `Player` class, which will be used when instantiating each `Level`.
- A list of Levels in the order in which they are loaded. This list will be empty until either the `load_file` or `load_url` method is called, but upon loading a game, this list must contain instances of the Level class, one for each level in the game data.

Further, the following methods must be defined:

load_file(self, game_file) Loads a game from a file, given by `game_file`, using `load_game_file` from the support file.

load_url(self, game_url) Loads a game from a url, given by `game_url`, using `load_game_url` from the support file.

The following applies to both `load_file` & `load_url`:

- The player's Dex should not be reset by `load_file/url`.
- The player's list of caught pokemon should not be reset.
- Errors raised by `load_game_file/url` or `Level`'s constructor should be ignored (i.e. reraised and not handled or suppressed).

start_next_level(self) Attempts to start the next level of the game. Returns True iff the game is completed (i.e. already on the final level), else False. This method should raise an `InvalidPositionError` if the level contains any invalid positions. **N.b.** It can be assumed that this method will not be called unless the current level is completed.

get_player(self) Returns the player of the game.

get_level(self) Returns the current level, an instance of `Level`, else None if the game hasn't started.

__len__(self) Returns the total number of levels in the game.

is_complete(self) Returns True iff no levels remain incomplete, else False.

move_player(self, direction) Attempts to move the player in the given direction. Returns whatever the player would hit (an instance of `GameObject`) in attempting to move, else `None`. If `direction` is not one of `NORTH, EAST, SOUTH, WEST`, this method raises a `DirectionError` (see support file). **N.b.** It can be assumed that this method will not be called unless a level has been started.

```
>>> game = Game()
>>> game.load_file('game2.json')
>>> len(game)
3
>>> game.start_next_level()
False
>>> game.move_player(EAST)
>>> game.move_player(SOUTH) # displayed using __repr__
Wall('#')
>>> game.move_player(EAST)
>>> game.move_player(SOUTH)
>>> game.move_player(WEST) # displayed using __repr__
Pokemon('Pikachu', (2, 2), 'Ice')
>>> game.is_complete()
False
>>> game.get_level().is_complete()
False
```

3 Assessment and Marking Criteria

In addition to providing a working solution to the assignment, you are also required to discuss your submission with a tutor. This will take place in the practical session you have signed up to in week 10. You **must** attend that session in order to obtain marks for the assignment.

In preparation for your discussion with a tutor you may wish to consider:

- any parts of the assignment that you found particularly difficult, and how you overcame them;
- whether you considered any alternative ways of implementing a given function;
- where you have known errors in your code, their cause and possible solutions (if known).

It is also important that you can explain to the tutor how each of the functions that you have written operates (for example, why using a for loop or a while loop in a function was the right choice).

Marks will be awarded based on a combination of the correctness of your code and on your understanding of the code that you have written. **A technically correct solution will not elicit a pass mark unless you can demonstrate that you understand its operation.**

A partial solution will be marked. If your partial solution causes problems in the Python interpreter please comment out that code and we will mark that.

4 Assignment Submission

You must submit your copy of `a2.py` electronically through Blackboard. You may submit multiple times before the deadline, but only the last submission will be marked.

Late submission of the assignment will not be accepted. In the event of exceptional circumstances, you may submit a request for an extension.

All requests for extension must be submitted on the UQ Application for Extension of Progressive Assessment form:
`http://www.uq.edu.au/myadvisor/forms/exams/progressive-assessment-extension.pdf`
by 48 hours prior to the submission deadline. The application and supporting documentation must be submitted to the ITEE Coursework Studies office (78-425) or by email (`enquiries@itee.uq.edu.au`).

Changelog

Final Version 1.0.4 (September 22, 2016)

- Tweaked handling of level completion in `gui.py`.
- Condensed 3 & 4.
- Fixed typos in 2.3.6 & 2.3.7.
- Added `__repr__` methods to `GameObject` and its subclasses to make example code easier to read (2.3.7 & 2.3.8); outlined in 2.3.

Version 1.0.3 (September 20, 2016)

- Released game data server (2.2.3).
- Added error handling to load operations in `gui.py`.
- Fixed typo in `load_game_url` in `a2_support.py`.
- Tweaked game dynamics to improve play in completed levels in `gui.py` & `game2.json`.

Version 1.0.2 (September 19, 2016)

- Removed "Mew" that appeared out of nowhere in example code for Dex class (2.3.6).
- Clarified `Dex.register_from_dex` (2.3.6).
- Fixed `UnicodeDecodeError` in `gui.py` and `a2_support.py`.

Version 1.0.1 (September 16, 2016)

- Added `pokemon.txt` and `images/pikachu.gif` to `a2_files.zip`.
- Added optional GUI autorun code to `a2.py`.
- Added example code for most classes.
- Modified support code in `a2_support.py`:
  - `is_position_valid` no longer incorrectly rejects walls on top or left boundary.
  - `load_game_file/url` now raise standardised errors that are referenced in their docstrings.
- Added `__str__` method to `Pokemon`/`Player` class in 2.3.3/2.3.5.
- Added error handling to `register_pokemon` in 2.3.5.
- Corrected typo with `pokemons` incorrectly being a list of tuples to a list of dictionaries 2.2.3.
- Clarified that Dex only deals with string names of Pokemon in 2.3.6.
- Clarified `register_from_dex` in 2.3.6.
- Corrected typo in `__contains__` in 2.3.6.
- `Game` must store a list of instances of the `Level` class in 2.3.8.
- Moved raising of `InvalidPositionError` from `start_next_level` to `load_file` in 2.3.8.
- Clarified various methods in 2.3.8.