CSCI3180 – Principles of Programming Languages – Spring 2022

Assignment 2 — Dynamic and Duck Typing

Deadline: Mar 6, 2022 (Sunday) 23:59

1 Introduction

The purpose of this assignment is to offer you a first-hand experience with Python, which supports the object-oriented design and programming paradigm. Our main focuses are Dynamic Typing and Duck Typing.

The assignment consists of two main parts. First, you need to implement the rules of a single-player game named *Strange Planet* in Python to brush up on your Python skills based on the given Java implementation. Second, we will extend the game and you need to implement more classes for the game in Python. In the report, you need to demonstrate the flexibility of Dynamic Typing and Duck Typing. *All Python implementations in this assignment should be built using Python 3.8.*.

2 Task 1: Strange Planet

Because of a wrong operation, your spacecraft crash-landed on an unknown planet in space, and you get lost wandering around. The planet is full of dangers and unknown creatures. Try to find your way to your spaceship and get back to the Earth again. This is a programming exercise for you to get familiar with Python, which is a dynamically typed language. In this task, you have to *strictly follow* our proposed object-oriented design and the game rules for Strange Planet stated in this section. *You have to follow the prototypes of the given functions exactly.* The Java implementation of the game is also given to you for reference. Another purpose of this exercise is for you to appreciate our object-oriented design. Think about how you would design the software and compare it to our design. You need to implement this game in *Python*. Note that if you need to access a private variable (whose name starts with an underscore) of a class from another class, you must use the getter and the setter. You are not allowed to access a private variable directly using a dot operator from another class.

2.1 Programming Environment Preparation

Before you can build and run the Java program we provide, you may have to prepare your development environment first. We will provide a virtual machine containing everything you need in the development. If you prefer programming on your machine, please follow the instructions below to install the development kits and some necessary packages. If you cannot be bothered to configure all these things, please use our virtual machine instead.

1. **Build Tools**. We provide a `Makefile` so that you can easily build the Java application via a `make` command. On an Ubuntu machine with root privilege, install build tools by executing the following command in the terminal

   ```
   sudo apt-get install build-essential
   ```

   On macOS, if the `make` command is not found, you should install the build tools via XCode. On Windows, you should consider downloading an IDE and configuring it by yourself, too.

2. **Java Development Kit (JDK)**. To build the Java source files, you have to install the JDK first. There are different JDKs available for a variety of platforms. To build this Java application, feel free to install any kind of JDK, like OpenJDK, Oracle JDK, IBM Java SDK, Apache Harmony, *etc.* Please refer to their official websites for the installation guide in detail.

   We recommend you install the open-source OpenJDK. On a Ubuntu machine with root privilege, open your terminal and execute the following command to find all available OpenJDK versions.

```
apt search ^openjdk-[0-9]*-jdk$
```

Install any version you like via the following commands. Replace `<version>` with specific version number. For example, you can install any one of `openjdk-8-jdk`, `openjdk-11-jdk`, `openjdk-13-jdk` and `openjdk-14-jdk` on Ubuntu 20.04 LTS.

```
sudo apt-get update
# Replace <version> with a specific version number.
# For example, install openjdk-14-jdk on Ubuntu 20.04 LTS.
sudo apt-get install openjdk-<version>-jdk
```

After executing this command in the terminal, you should have already installed OpenJDK correctly. Check the Java version and Java compiler version by `java --version` and `javac --version`. You will get a *similar* output as follows if everything goes fine:

```
$ java --version
OpenJDK version "14.0.2" 2020-07-14
OpenJDK Runtime Environment (build 14.0.2+12-Ubuntu-120.04)
OpenJDK 64-Bit Server VM (build 14.0.2+12-Ubuntu-120.04, mixed mode,
    sharing)
$ javac --version
javac 14.0.2
```

Now you have successfully installed the JDK. Similarly, on any MacOS machine, simply open your terminal and execute the following command:

```
# Replace <version> with a specific version number.
brew cask install adoptopenjdk<version>
```

On Windows, you may have to refer to the official websites for the installation guide in detail.

Again, if you find it tedious to set up the environment on your machine, use our *Ubuntu* virtual machine instead. Everything mentioned above in this section will be pre-installed so that you do not need to worry about wasting time on environment preparation.

2.2 Description

Strange Planet is a single-player game. The player needs to get to the spaceship over a complicated terrain, represented by an $\mathbf{n} \times \mathbf{m}$ map consisting of $\mathbf{n}$ rows and $\mathbf{m}$ columns. Two cells are *one step away from each other* if they are horizontally or vertically adjacent, and they are *neighbors* if they are adjacent to each other either vertically, horizontally, or diagonally. Except for cells along the boundaries of the map, all cells have exactly eight neighbors. See Figure 1 for an example. The cells that are one step away from cell $(3, 3)$, i.e., the cell at row 3 and column 3, are highlighted by red boxes, and all eight neighbors of cell $(3, 3)$ are highlighted by either red or blue boxes.

The player starts in the bottom-left corner $(\mathbf{n} - 1, 0)$ and needs to reach the top-right corner $(0, \mathbf{m} - 1)$ where the spaceship is located. Due to the limited oxygen, the player must reach the spaceship within $\mathbf{h}$ hours and has $\mathbf{p}$ health points (HPs) initially. The cells on the map have different terrain types, such as plain, mountain, and swamp. According to the terrain of a cell, the player may either cannot enter the cell or spends several hours traversing it.

- The color of a mountain cell is grey, and the player cannot enter a mountain cell.
- The color of a swamp cell is blue, and the player spends 2 hours to travel out of it.
- The color of a plain cell is green, and the player spends 1 hour to travel out of it.

See Figure 1 for example cells with different terrain types. Note that the player's oxygen is updated after the player moved from one cell to another, and the amount (in terms of hours) of oxygen to be deducted is based on the type of cell out of which the player is traveling.

There are $\mathbf{g}$ Goblins scattered around the map. In other words, a cell can either be empty or occupied by the player or a Goblin. At each round, the player takes action first, followed by the

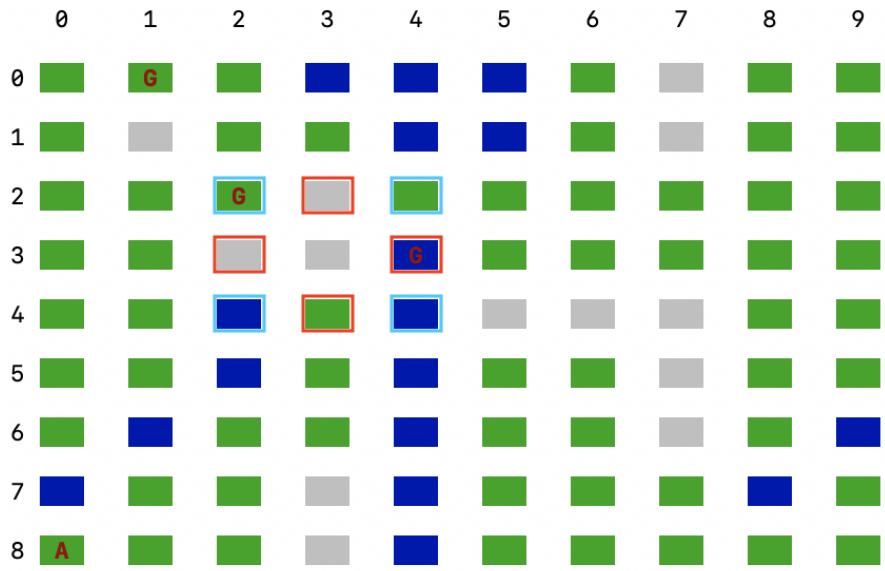

Figure 1: Example map of Strange Planet.

Goblins' actions in turns. The player can move to a cell that is one step away from the player's current position by specifying the direction: "`L`" (Left), "`R`" (Right), "`U`" (Up), and "`D`" (Down). When the player attempts to move to a new cell, one of the following cases will happen:

- The action is invalid if the target cell is a mountain cell or the movement is out of the map boundary.
- If the target cell is an empty swamp or plain cell, the player enters the target cell.
- If the target cell is occupied by a Goblin, the Goblin disappears. The player enters the target cell, and the player's HP is reduced by 1.

All Goblins patrol on the map according to a predetermined sequence of actions, and they move a cell that is one step away from the current cell at each round. When a Goblin attempts to move to a new cell, one of the following cases will happen:

- If the target cell is a mountain cell, the Goblin stays at the original cell.
- If a Goblin enters a cell occupied by the player, the Goblin disappears. The player's HP is reduced by 1.

The moving routes of every pair of Goblins are always non-overlapping. In other words, a Goblin never attempts to move to a cell occupied by another Goblin.

After the player and all Goblins take action, one of the three cases will happen:

- The player loses if either the player's remaining oxygen (in hours) or HPs is non-positive.
- The player wins if the player reaches the goal cell $(0, \mathbf{m} - 1)$, and the player's remaining oxygen (in hours) and HPs are both positive.
- Otherwise, the game continues until either the player wins or loses.

2.3 Input/Output Specification

In this exercise, you are required to read the map information from a file and use the command line to get inputs about the player's action. All outputs should also be printed to the command line window. Please follow the exact wording, spacing, and format of the output attachment.

```
9 10 30 5 3
P P P S S S P M P P
P M P P S S P M P P
P P P M P P P P P P
P P M M S P P P P P
P P S P S M M M P P
P P S P S P P M P P
P S P P S P P M P S
S P P M S P P P S P
P P P M S P P P P P
0 1 L R R L
2 2 L R U R L D
3 4 U U U D D D
```

Figure 2: Example map file of Strange Planet.

Input File The input file consists of all configurations of the game. An example file is given in Figure 2, which describes the map in Figure 1. The first line of an input file consists of **n** (the number of rows), **m** (the number of columns), **h** (the initial amount of oxygen), **p** (the initial number of HPs) and **g** (the number of Goblins). In each of the following **n** lines, there are **m** characters indicating the terrain type of each cell in that row ("P" for plains, "M" for mountains, and "S" for swamps). The final **g** lines contain information about the Goblins. Each line starts with two integers representing the initial row and column of a Goblin, followed by a sequence of characters representing the Goblin's repeated actions. For example, the first Goblin is initially at cell $(0, 1)$ and repeats the action sequence of moving left, moving right, moving right, and moving left. You can assume that after taking the specified sequence of actions, a Goblin will always return to its original position. At each round, Goblins should take action with order according to the appearance of their description in the file. In the example file, the Goblin with initial position $(0, 1)$ should move first at each round, and then the Goblin initially at $(2, 2)$ moves second, followed by the Goblin initially at $(3, 4)$.

Displaying Game Information After each round of actions by the player and the Goblins, the updated information, which consists of the current map state and the players' remaining oxygen and HPs should be printed.

Map Format The map should include all cells and objects as required in Section 2.2, together with row number and column number. As is shown in Figure 3, we aim to print a map containing nine rows ten columns. The first row starts with three spaces, followed by numbers from 0 to 9, each separated from the subsequent one by five spaces. Each row of the map consists of a row number, one space, and a list of colored cells starting from the second row. Each cell has three characters to display the terrain type and the occupant in the cell by the corresponding background color and characters for the occupants:

- When the player occupies a cell, the character in the middle is "`A`";
- When a Goblin occupies a cell, the character in the middle is "`G`";
- Otherwise, the cell is empty, and the character in the middle is a space character.

Every two cells in the same row are three spaces away from each other. The ANSI escape codes for colors of different terrain types are given in the skeleton code, and you may follow the description in https://www.lihaoyi.com/post/BuildyourownCommandLinewithANSIescapecodes.html to print colored text.

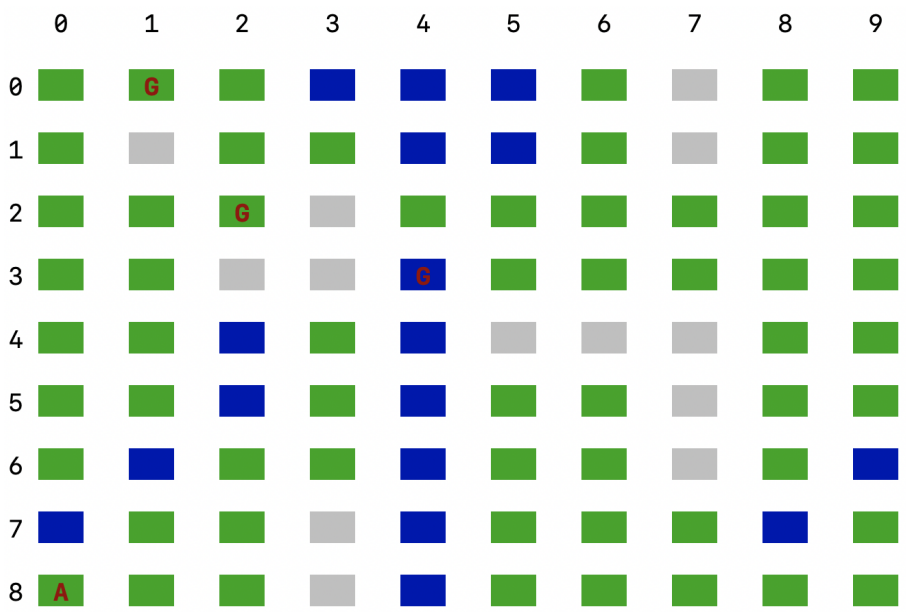

Figure 3: Example map file of Strange Planet.

Player's Status The player's status should be in a new line following the cell matrix. The string should be "`Oxygen: <oxy>, HP: <hp>.`" where `<oxy>` and `<hp>` are the player's remaining oxygen in hour(s) and HPs.

Human-Controlled Player When the player takes action at each round, your program should prompt a message "`Next move (U, D, R, L):`, and the input should be a character of either "`L`" (Left), "`R`" (Right), "`U`" (Up) or "`D`" (Down). See Figure 3. If the input is invalid, you should output a message "`Invalid command. Please enter one of {U, D, R, L}.`"

Other Information There are other messages such as the movements of non-player game characters and the interaction among game characters when one game character attempts to enter an occupied cell. The messages are given in the skeleton code, and *you are not allowed to change any given print statements.*

Game End When the game is finished, a string `<msg>` should be printed to indicate the result of the game. The string "`<msg>`" can be either "Congrats! You win!" or "Bad Luck! You lose.".

2.4 Class Design

Please follow the classes defined below in your implementation. *You are not allowed to add any variables, methods, or classes.* We will start the program with `python3 StrangePlanet.py basic`. The following class design uses the naming convention of Python, and the class design for Java will be given in the appendix. In particular, underscored variables are "private". You are also required to use properties for setter and getter method used in the given Java implementation. You are supposed to follow strictly the Python naming convention to name classes, methods and variables. Please accomplish all tasks marked with **TODO** in the provided Python template.

1. **Class** `Cell`: an abstract super class representing a cell object.

- **Instance Variable(s)**
  - `_row`, `_col`: integers representing the row and column of the cell.
  - `_hours`: the number of hours required to travel away from the cell.
  - `_occupant`: an object which is occupying the cell.
  - `_color`: a string for the ANSI escape codes for the background color.
- **Instance Method(s)**
  - `__init__(row, col)`: initialize a cell and set the position to `(row, col)`.
  - `occupant()`: the getter method of `_occupant` using property decorator.
  - `hours()`: the getter method of `_hours` using property decorator.
  - `set_occupant(obj)`: set the occupant for the cell. If `_occupant` is not empty, call the `interact_with` method of the `_occupant`. Return `True` to indicate if setting `obj` to be the new occupant is successful; otherwise return `False`.
  - `remove_occupant()`: set `_occupant` to `None`.
  - `display()`: print a string to display the cell and the occupant according to the terrain type and the output format.

2. **Class** `Plain`: a class extending the `Cell` class to represent a cell with the terrain type being "plain". The `Plain` class has the following additional instance method(s):
   - **Additional Instance Method(s)**
     - `__init__(row, col)`: initialize a plain cell and set its position to `(row, col)`. Initialize instance variables `_hours` and `_color`.
3. **Class** `Mountain`: a class extending the `Cell` class to represent a cell with the terrain type being "mountain". The `Mountain` class has the following additional instance method(s):
   - **Additional Instance Method(s)**
     - `__init__(row, col)`: initialize a mountain cell and set its position to `(row, col)`. Initialize instance variable `_color`.
     - `set_occupant(obj)`: return `False`.
4. **Class** `Swamp`: a class extending the `Cell` class to represent a cell with the terrain type being "swamp". The `Swamp` class has the following additional instance method(s):
   - **Additional Instance Method(s)**
     - `__init__(row, col)`: initialize a swamp cell and set its position to `(row, col)`. Initialize instance variables `_hours` and `_color`.
5. **Class** `GameCharacter`: an abstract class representing a game character.
   - **Instance Variable(s)**
     - `_row`, `_col`: integers representing the current position of the game character.
     - `_occupying`: an object of the `Cell` class which is occupied by the game character.
     - `_name`: a string representing the type of the game character.
     - `_active`: a Boolean variable to indicate whether the game character is active.
     - `_character`: a character for displaying the game character on the map.
     - `_color`: a string for the ANSI escape codes for the text color.
   - **Instance Method(s)**
     - `__init__(row, col)`: initialize a game character and set the position to be `(row, col)`.
     - `name()`: the getter method of `_name` using property decorator.
     - `row()`: the getter method of `_row` using the property decorator.
     - `col()`: the getter method of `_col` using the property decorator.
     - `active()`: the getter method of `_active` using the property decorator.
     - `occupying()`: the getter method of `_occupying` using the property decorator.

- `active(active)`: the setter method of `_active` using the property decorator.
- `occupying(cell)`: the setter method of `_occupying` using the property decorator.
- `act(map)`: an abstract method for the game character to take action at each round.
- `interact_with(comer)`: an abstract method which is called when `comer`, which is an object of the `GameCharacter` class, attempts to enter the cell occupied by the game character.
- `cmd_to_pos(char)`: a helper function which returns the next position according to the input character `char`.
- `display()`: return the value of `_character`.

6. **Class** `Player`: a class extending the `GameCharacter` class to represent the player. The `Player` class has the following additional instance variable(s) and method(s):

   - **Additional Instance Variable(s)**
     - `_hp`, `_oxygen`: integers representing the remaining HP and oxygen of the player.
     - `_valid_actions`: a list of characters representing the valid actions for the player.
   - **Additional Instance Method(s)**
     - `__init__(row, col, h, o)`: initialize the position to `(row, col)` and the initial `_hp` and `_oxygen` to `h` and `o` respectively.
     - `hp()`: the getter method of `_hp` using property decorator.
     - `hp(h)`: the setter method of `_hp` using the property decorator.
     - `oxygen()`: the getter method of `_oxygen` using property decorator.
     - `oxygen(ox)`: the setter method of `_oxygen` using the property decorator.
     - `act(map)`: the player takes action with the following iteration:
       (a) Prompt a message and get the user input for the moving direction and check whether the input is valid. If the input is invalid, print an error message and repeat step (a).
       (b) Get the position of the next cell by the user input and call the `get_cell` method of the `_map` object. If the returned cell is `None`, go to step (a).
       (c) Call the `set_occupant` method to update the occupant of the returned cell. If the returned value is `False`, go to step (a). Otherwise, the move is successful, call `remove_occupant` of `_occupying` and update the variables `_col`, `_row`, `_oxygen` and `_occupying` according to the game rules.
       (d) Check whether `_active` is true. If not, remove the player from the cell `_occupying` and update the `_occupying` according to the game rules.
     - `interact_with(comer)`: check whether `comer` is a Goblin. If so, print a message to indicate that the player meets a Goblin. Update `_hp` and `_active` of `comer` according to the game rules. Return `False` to indicate that `comer` does not occupy the cell.

7. **Class** `Goblin`: a class extending the super class `GameCharacter` to represent a Goblin. The `Goblin` class has the following additional instance variable(s) and method(s):

   - **Instance Variable(s)**
     - `_actions`: a list of characters representing the cyclic action sequence.
     - `_cur_act`: an integer indicating the current action of the Goblin in `_actions`.
     - `_damage`: an integer indicating the damage that can be caused by the Goblin.
   - **Instance Method(s)**
     - `__init__(row, col, actions)`: initialize a Goblin with a list `actions` of cyclic action sequence, and set the position to `(row, col)`
     - `damage()`: the getter method of `_damage` using property decorator.
     - `act(map)`: a Goblin takes action with the following steps:

(a) Get the position of the next cell from `actions` and `cur_act`, and get the next cell by calling the `get_cell` of the `map` object. Increment `cur_act` by one.

(b) If the next cell is not `None`, call the `set_occupant` method to update the occupant of the returned cell. If the move is successful, call `remove_occupant` of `_occupying` and update the variables `_col`, `_row`, and `_occupying` according to the game rules. Print a message to indicate the Goblin enters a new cell.

(c) Check whether `_active` is true. If not, remove the Goblin from the cell `_occupying` and update the `_occupying` according to the game rules.

- `interact_with(comer)`: check whether `comer` is the player. If so, print a message to indicate that the Goblin meets the player at position `(_row, _col)`. Update `_active` and the `_hp` of `comer` according to the game rules. Return `True` to indicate that `comer` occupies the cell.

8. **Class** `Map`: an object which contains basic information of the map and methods for inquiring the relevant information.

   - **Instance Variable(s)**
     - `_rows`, `_cols`: integers representing the number of rows and columns of the map.
     - `_cells`: a two-dimensional array where each element is an object of the `Cell` class.
   - **Instance Method(s)**
     - `__init__(height, width)`: initialize a `height` $\times$ `width` map, where all elements are `Cell` objects.
     - `rows()`: the getter method of `_rows` using property decorator.
     - `cols()`: the getter method of `_cols` using property decorator.
     - `get_cell(row, col)`: check whether the position is out of boundary. If so, print a message to indicate the position is out of boundary and return `None`. Otherwise, return cell at `(row, col)`.
     - `build_cell(row, col, cell)`: check whether the position is out of boundary. If so, print a message to indicate the position is out of boundary and return `False`. Otherwise, add `cell` to `_cells` at the position `(row, col)` and return `True`.
     - `get_neighbours(row, col)`: return a list of neighboring cells of the cell at `(row, col)`.
     - `display()`: display the map by calling the `display` method of each cell in `_cells`.

9. **Class** `Engine`: a game engine object which holds the map, coordinates the actions of different objects and controls the screen outputs.

   - **Instance Variable(s)**
     - `_map`: an object of the `map` class.
     - `_player`: an object of the `player` class.
     - `_actors`: a list of all objects that can take actions at each round.
   - **Instance Method(s)**
     - `__init__(data_file)`: initialize the objects in the game according to the input file with the name `data_file`.
     - `run()`: run the game until the game ends.
     - `clean_up()`: clean up all objects that are not active in `_actors`.
     - `state()`: return 1 (win) and -1 (lose) when the game ends. Otherwise, return 0.
     - `print_info()`: print out the map and the relevant information of the game for each round according to the output format.
     - `print_result()`: print a string for the final result of the game.

3 Task 2: Demonstrating Advantages of Dynamic Typing

There are commonly-claimed advantages of Dynamic Typing:

1. More generic code can be written. In other words, functions can be defined to apply to arguments of different types.

2. Possibilities of mixed type collection data structures.

Please provide a scenario and concise example codes to demonstrate the advantages of Dynamic Typing mentioned above. You are welcome to provide other advantages and disadvantages along with code segment for bonus points.

Dynamic Typing makes coding more flexible and convenient, but you should bear in mind that type checking can only be carried out at runtime, incurring time overhead and reliability issues.

4 Task 3: Duck Typing

The strange planet is more dangerous than expected! In this task, you are required to implement more classes including `Trap` and `Volcano`. After accomplishing this task, you will have experienced a special feature of Python called *Duck Typing* that is available only in dynamically typed programming languages. Again, *you have to follow the prototypes of the given functions exactly.* Additionally, you must not use the *type* function to check the type of a variable in your program.

4.1 Duck Typing

The following synopsis of Duck Typing is summarized from:

> http://en.wikipedia.org/wiki/Ducktyping
> http://www.sitepoint.com/making-ruby-quack-why-we-love-duck-typing

In standard statically-typed object-oriented languages, objects' classes (which determine an object's characteristics and behavior) are essentially interpreted as the objects' types. Duck Typing is a new typing paradigm in dynamically-typed (late binding) languages that allow us to dissociate typing from objects' characteristics. It can be summarized by the following motto by the late poet James Whitcombe Riley:

> *When I see a bird that walks like a duck and swims like a duck and quacks like a duck,*
> *I call that bird a duck.*

The basic premise of Duck Typing is simple. If an entity looks, walks, and quacks like a duck, for all intents and purposes it is fair to assume that one is dealing with a member of the species anas platyrhynchos. In practical Python terms, this means that it is possible to try calling any method on any object, regardless of its class. An important advantage of Duck Typing is that we can enjoy *polymorphism without inheritance*. An immediate consequence is that we can write more generic codes which are cleaner and more precise.

4.2 Description

On the strange planet, Goblins also make use of **t** traps for hunting. A trap is an *invisible* object so that a cell occupied by a trap is represented by and displays only a space character. When a game character enters a cell occupied by a trap, one of the following cases happens:

- If the game character is the player, then the player will lose one HP and one hour of oxygen.
- If the game character is a Goblin, the Goblin will become inactive and disappear from the game.

No matter what type of game character enters a cell containing a trap, the trap will become exposed and disappear from the game. Figure 5 gives an example map. It is shown that there is one trap in one of the eight neighboring cells of cell $(7, 1)$ where the player is currently occupying.

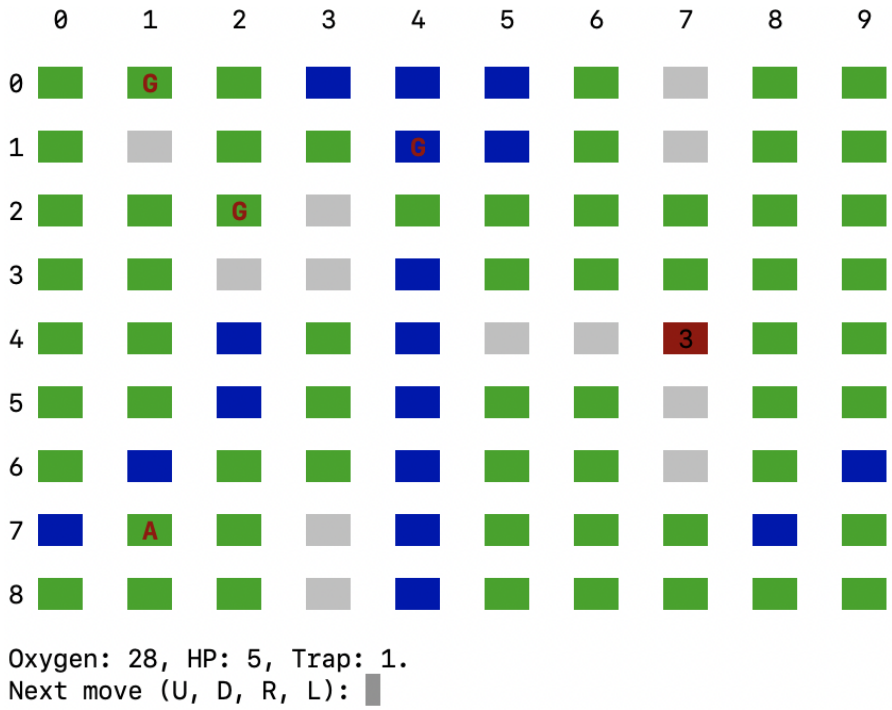

Figure 4: Example map of the game with extensions.

There are also **v** volcanoes on the map. A volcano is a special mountain, which also "takes action" in each round: it erupts with a certain frequency. The color of a volcano cell is red, and a volcano will display a number representing the number of rounds before its next eruption. For example, there is one volcano on the map in Figure 5 at position `(4,7)`, and the volcano will erupt in three rounds. When a volcano erupts, the player will get one HP damage if the player is at one of the neighboring cells of the volcano cell. Also, all Goblins at neighboring cells of an erupted volcano will become inactive and disappear from the game.

4.3 Input/Output Specification

Since there are additional elements on the planet, we update the input and output formats as follows. Please follow the exact wording, spacing, and format of the output attachment.

Input File Figure 5 shows an example map file with traps and volcanoes. The additional parts are highlighted by red boxes. There are two additional numbers in the first line: **t** (the number of traps) and **v** (the number of volcanoes). The map for the file in Figure 5 describes a map with two traps and one volcano. There are three additional lines at the end of the file, where the first two lines describe the information for the two traps and the last line is about the volcano. Each line for a trap consists of two integers for the row and the column number of the trap. Each line for a volcano consists of an integer `row` for row number, an integer `col` column number and an integer `freq` for the frequency of eruption, indicating that the `Mountain` at `(row, col)` is a volcano and erupts for every `freq` rounds. You can assume that *there are no traps and volcanoes at the bottom-left and the top-right cells.*

Displaying Game Information After each round of actions by the player, all Goblins, and all volcanoes, the updated information should be printed in the command line. The additional messages for traps and volcanoes are specified as follows.

Traps The information of traps should be printed after displaying the map. The status message shall become "`Oxygen: <oxy>, HP: <hp>, Trap: <trap>`", where `<oxy>` and `<hp>` are the player's remaining oxygen and HPs, and `<trap>` is the number of traps surrounding the player.

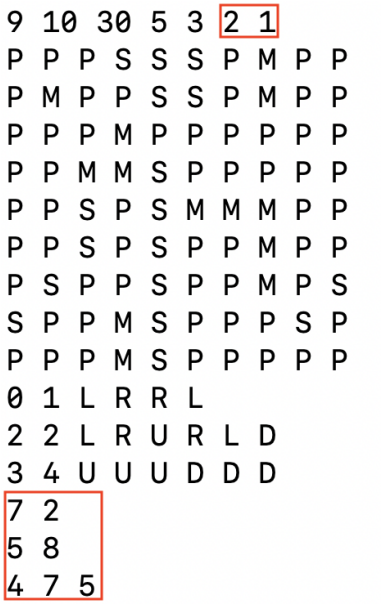

Figure 5: Example map file of Strange Planet with extensions

Volcanoes A volcano is displayed as a cell with three characters, where the first and the last ones are spaces. The middle character is the number of rounds before the volcano erupts.

Other Information There is other information such as the "actions" of volcanoes and the interaction between traps and game characters when one game character attempts to enter an occupied cell. The messages are given in the skeleton code, and *you are not allowed to change any given print statements.*

4.4 Class Design

Please follow the classes defined below in your implementation. *You are not allowed to add any variables, methods, or classes.* We will start your Python program using `python3 StrangePlanet.py extension`. The following class design uses the naming convention of Python, and the class design for Java will be given in the appendix. You are supposed to follow strictly the Python naming convention to name classes, methods, and variables. Please accomplish all tasks marked with **TODO** in the provided Python template.

1. **Class `Trap`**: a class representing a trap with the following components:
   - **Instance Variable(s)**
     - `_row`, `_col`: integers representing the current position of the trap.
     - `_name`: a string representing the trap.
     - `_occupying`: an object of the `Cell` class which is occupied by the trap.
   - **Instance Method(s)**
     - `__init__(row, col)`: initialize a cell and set the position to `(row, col)`.
     - `name()`: the getter method of `_name` using property decorator.
     - `occupying()`: the getter method of `_occupying` using property decorator.
     - `occupying(cell)`: the setter method of `_occupying` using property decorator.
     - `interact_with(comer)`: Remove the trap from the cell `_occupying` and update `_occupying` according to the game rule. Update the properties of `comer` as follows:
       (a) If `comer` is the player, update the player's oxygen and HPs and print a message to indicate the player has fallen into a trap. Return `True` to indicate the player occupies the cell successfully.

(b) If `comer` is a Goblin, update the active status of `comer` and print a message to indicate a Goblin has fallen into a trap. Return `False` to indicate the Goblin did not occupy the cell successfully.

– `display()`: return a space character.

2. **Class** `Volcano`: a class extending the class `Mountain` with the following additional components:

- **Instance Variable(s)**
  - `_frequency`: an integer representing the frequency for volcanic eruption.
  - `_countdown`: an integer representing the number of rounds before the next eruption.
  - `_active`: a Boolean variable to indicate whether the volcano is active or not.
- **Instance Method(s)**
  - `__init__(row, col, freq)`: initialize a volcano cell with the eruption frequency being `freq`, and set the position `(row, col)`.
  - `active()`: the getter method of `_active` using property decorator.
  - `act(map)`: a method which is called when the volcano "takes action". Reduce the `_countdown` by one. When `_countdown` becomes 0:
    (a) Print a message to indicate that the volcano at the position `(_row, _col)` erupts and reset `_countdown` to be `_frequency`.
    (b) Get all neighboring cells using the `get_neighbours` of `map`. Update the properties of all occupants at the neighboring cells according to the game rules.
  - `display()`: print a string to display the volcano and the countdown according to the output format.

In the Python implementation, you also need to adapt the `Engine` class to become the `NewEngine` class by changing the following components:

- `__init__(data_file)`: initialize all objects including objects of the `Trap` class and the `Volcano` class.
- `print_info()`: print out the map and the relevant information including the number of traps surrounding the player on the map.

5 Task 4: Demonstrating Advantages of Duck Typing

There are at least two places where we use duck typing in Python. Can you identify them? Please provide the scenarios and concise example codes to compare your Python and Java implementations and explain how duck typing makes coding more flexible and convenient.

6 Report

Your report should answer the following questions within **FOUR** A4 pages.

1. Provide example code and necessary elaborations for demonstrating advantages of Dynamic Typing as specified in Task 2.
2. Provide example code and necessary elaborations for demonstrating advantages of Duck Typing as specified in Task 4.

7 Submission Guidelines

Please read the guidelines **CAREFULLY**. If you fail to meet the deadline because of submission problems on your side, marks will still be deducted. So please start your work early!

1. In the following, **SUPPOSE**

your name is Chan Tai Man,
your student ID is 1155234567,
your username is tmchan, and
your email address is tmchan@cse.cuhk.edu.hk.

2. In your source files, insert the following header. **REMEMBER** to insert the header according to the comment rule of Python.

```
/*
 * CSCI3180 Principles of Programming Languages
 *
 * --- Declaration ---
 *
 * I declare that the assignment here submitted is original except for source
 * material explicitly acknowledged.  I also acknowledge that I am aware of
 * University policy and regulations on honesty in academic work, and of the
 * disciplinary guidelines and procedures applicable to breaches of such policy
 * and regulations, as contained in the website
 * http://www.cuhk.edu.hk/policy/academichonesty/
 *
 * Assignment 2
 * Name : Chan Tai Man
 * Student ID : 1155234567
 * Email Addr : tmchan@cse.cuhk.edu.hk
 */
```

3. Late submission policy: less 20% for 1 day late and less 50% for 2 days late. We shall not accept submissions more than 2 days after the deadline.

4. The report should be submitted to VeriGuide, which will generate a submission receipt. The report should be named `report.pdf`. The VeriGuide receipt of the report should be named `report.pdf`. The report and receipt should be submitted together with codes in the same ZIP archive.

5. `Tar` your source files to `username.tar` by

```
tar cvf tmchan.tar python.zip report.pdf receipt.pdf
```

6. `Gzip` the `tarred` file to `username.tar.gz` by

```
gzip tmchan.tar
```

7. `Uuencode` the `gzipped` file and send it to the course account with the email title "HW2 *studentID yourName*" by

```
uuencode tmchan.tar.gz tmchan.tar.gz \
| mailx -s "HW2 1155234567 Chan Tai Man" csci3180@cse.cuhk.edu.hk
```

8. Please submit your assignment using your Linux accounts.

9. An acknowledgment email will be sent to you if your assignment is received. Please **DO NOT** delete or modify the acknowledgement email. You should contact your TAs for help if you do not receive the acknowledgment email within 5 minutes after your submission. **DO NOT** re-submit just because you do not receive the acknowledgment email.

10. You can check your submission status at

    http://course.cse.cuhk.edu.hk/~csci3180/submit/hw2.html.

11. You can re-submit your assignment, but we will only grade the latest submission.

12. Enjoy your work :>

A Class Designs for Java Implementations

In this section, the instance variables and the instance methods of Java implementation are compared with those of Python implementation.

1. **Class** `Cell`.
   - **Instance Variable(s)**
     - `int row`, `int col`: corresponding to `_row` and `_col`.
     - `int hours`: corresponding to `_hours`.
     - `Object occupant`: corresponding to `_occupant`.
     - `String color`: corresponding to `_color`.
   - **Instance Method(s)**
     - `Cell(int row, int col)`: corresponding to `__init__(row, col)`.
     - `Object getOccupant()`: corresponding to `occupant()`.
     - `int getHours()`: corresponding to `hours()`.
     - `boolean setOccupant(Object obj)`: corresponding to `set_occupant(obj)`.
     - `void removeOccupant()`: corresponding to `remove_occupant()`.
     - `void display()`: corresponding to `display()`.
2. **Class** `Plain`.
   - **Additional Instance Method(s)**
     - `Plain(int row, int col)`: corresponding to `__init__(row, col)`.
3. **Class** `Mountain`.
   - **Additional Instance Method(s)**
     - `Mountain(int row, int col)`: corresponding to `__init__(row, col)`.
     - `boolean setOccupant(Object obj)`: corresponding to `set_occupant(obj)`.
4. **Class** `Swamp`.
   - **Additional Instance Method(s)**
     - `Swamp(int row, int col)`: corresponding to `__init__(row, col)`.
5. **Class** `GameCharacter`.
   - **Instance Variable(s)**
     - `int row`, `int col`: corresponding to `_row` and `_col`.
     - `Cell occupying`: corresponding to `_occupying`.
     - `String name`: corresponding to `_name`.
     - `boolean active`: corresponding to `_active`.
     - `char character`: corresponding to `_character`.
     - `String color`: corresponding to `_color`.
   - **Instance Method(s)**
     - `GameCharacter(int row, int col)`: corresponding to `__init__(row, col)`.
     - `String getName()`: corresponding to `name()`.
     - `int getRow()`: corresponding to `row()`.
     - `int getCol()`: corresponding to `col()`.
     - `boolean getActive()`: corresponding to `active()`.
     - `Cell getOccupying()`: corresponding to `occupying()`.
     - `void setActive(boolean active)`: corresponding to `active(active)`.
     - `void setOccupying(cell)`: corresponding to `occupying(cell)`.

- `void act(Map map)`: corresponding to `act(map)`.
- `boolean interactWith(Object comer)`: corresponding to `interact_with(comer)`.
- `String display()`: corresponding to `display()`.
- `int[] cmd2Pos(char c)`: corresponding to `cmd_to_pos(char)`.

6. **Class** `Player`.
   - **Instance Variable(s)**
     - `int hp`, `int oxygen`: corresponding to `_hp` and `_oxygen`.
     - `char[] validActions`: corresponding to `_valid_actions`.
   - **Instance Method(s)**
     - `Player(int row, int col, int hp, int oxygen)`: corresponding to `__init__(row, col, h, o)`.
     - `int getHp()`: corresponding to `hp()`.
     - `void setHp(int h)`: corresponding to `hp(h)`.
     - `int getOxygen()`: corresponding to `oxygen()`.
     - `void setOxygen(int o)`: corresponding to `oxygen(ox)`.
     - `boolean interactWith(Object comer)`: corresponding to `interact_with(comer)`.
     - `void act(Map map)`: corresponding to `act(map)`.
7. **Class** `Goblin`.
   - **Instance Variable(s)**
     - `char[] actions`: corresponding to `_actions`.
     - `int curAct`: corresponding to `_cur_act`.
     - `int damage`: corresponding to `_damage`.
   - **Instance Method(s)**
     - `Goblin(int row, int col, char[] actions)`: corresponding to `__init__(row, col, actions)`.
     - `void act(Map map)`: corresponding to `act(map)`.
     - `int getDamage()`: corresponding to `damage()`.
     - `boolean interactWith(Object comer)`: corresponding to `interact_with(comer)`.
8. **Class** `Map`.
   - **Instance Variable(s)**
     - `int rows`, `int cols`: corresponding to `_rows` and `_cols`.
     - `Cell[][] cells`: corresponding to `_cells`.
   - **Instance Method(s)**
     - `Map(int rows, int cols)`: corresponding to `__init__(height, width)`.
     - `int getRows()`: corresponding to `rows()`.
     - `int getCols()`: corresponding to `cols()`.
     - `Cell getCell(int row, int col)`: corresponding to `get_cell(row, col)`.
     - `boolean buildCell(int row, int col, Cell cell)`: corresponding to `build_cell(row, col, cell)`.
     - `ArrayList<Cell> getNeighbours(int row, int col)`: corresponding to `get_neighbours(row, col)`.
     - `void display()`: corresponding to `display()`.
9. **Class** `Engine`.
   - **Instance Variable(s)**
     - `Map map`: corresponding to `_map`.

- `Player player`: corresponding to `_player`.
- `ArrayList<Object> actors`: corresponding to `_actors`.

- **Instance Method(s)**
  - `Engine(String dataFile)`: corresponding to `__init__(data_file)`.
  - `void run()`: corresponding to `run()`.
  - `void cleanUp()`: corresponding to `clean_up()`.
  - `int state()`: corresponding to `state()`.
  - `void printInfo()`: corresponding to `print_info()`.
  - `void printResult()`: corresponding to `print_result()`.

10. **Class** `Trap`.

- **Instance Variable(s)**
  - `int row`, `int col`: corresponding to `_row` and `_col`.
  - `Cell occupying`: corresponding to `_occupying`.
  - `String name`: corresponding to `_name`.
- **Instance Method(s)**
  - `Trap(int row, int col)`: corresponding to `__init__(row, col)`.
  - `String getName()`: corresponding to `name()`.
  - `Cell getOccupying()`: corresponding to `occupying()`.
  - `void setOccupying(Cell cell)`: corresponding to `occupying(cell)`.
  - `boolean interactWith(Object comer)`: corresponding to `interact_with(comer)`.
  - `String display()`: corresponding to `display()`.

11. **Class** `Volcano`.

- **Instance Variable(s)**
  - `int frequency`: corresponding to `_frequency`.
  - `int countdown`: corresponding to `_countdown`.
  - `boolean active`: corresponding to `_active`.
- **Instance Method(s)**
  - `Volcano(int row, int col, int freq)`: corresponding to `__init__(row, col, freq)`.
  - `boolean getActive()`: corresponding to `active()`.
  - `void act(Map map)`: corresponding to `act(map)`.
  - `void display()`: corresponding to `display()`.