# CSCI3180 – Principles of Programming Languages – Spring 2019

**Assignment 2 — Your First Date with Python and Duck Typing**

**Deadline: Mar 15, 2020 (Sunday) 23:59**

## 1 Introduction

The purpose of this assignment is to offer you the first experience with Python, which supports the Object-Oriented programming paradigm. Our main focuses are Dynamic Typing and Duck Typing. Please use Python **3.6** to finish this assignment.

The assignment consists of 4 tasks.

- You need to implement a famous board game called Six Men's Morris in Python.
- You are asked to demonstrate the advantages/disadvantages of dynamic typing through some example code, which could be taken from any code you write for this assignment.
- We give you the Java source code of a game called "Save The Tribe". You need to re-implement the game in Python to make the code cleaner with the help of Duck Typing.
- Additional features and mechanisms are introduced to the "Save The Tribe" game. You need to implement the enhanced game in both Java and Python.

After completing the 4 tasks, you need to write a report elaborating on Dynamic Typing and Duck Typing.

**NOTES:** all your codes will be graded on the Linux machines in the Department. You are welcome to develop your codes on any platform, but please remember to test them on Department machines.

## 2 Task 1: Six Men's Morris

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 OO design and the game rules for the 2-player "Six Men's Morris" game stated in this section.

Six Men's Morris is a popular game in Italy, France and England during the Middle Ages but was obsolete by 1600. You can have a try to play this game in `https://www.novelgames.com/en/sixmensmorris`.

For simplicity, we want you to implement a watered-down version of Six Men's Morris. The detailed specification or game rules are described as follows.

### 2.1 Description

The game board consists of a grid with 16 intersections or **points** as shown in Figure 1. Each player has **six pieces**, or "men", coloured **black** or **white**. Players try to form "**mills**"— **three of their men lined up horizontally or vertically**—allowing a player to remove the opponent's man from the game board. For simplicity, Player 1 always uses the **white** pieces and Player 2 uses the **black** ones.

The game proceeds in the following two phases.

1. **Placing the pieces.** The game begins with an empty board. Player 1 first starts to play and the two players take turns placing their men, one per turn on empty points. This phase will last for 6 rounds since each player has only 6 men. Note that a player forming a mill can remove one of the opponent's pieces from the board. However, **no one can win during Phase 1**. After all men are placed, Phase 2 begins.

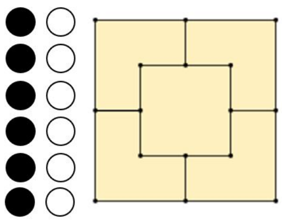


Figure 1: Game board for Six Men's Morris.

2. **Moving the pieces.** Players again alternate moves, this time moving a man to an **adjacent and unoccupied point**. A piece may not "jump over" another piece. Players continue to try to form mills and remove their opponent's pieces as in Phase 1. Note that a player can "break" a mill by moving one of his pieces out of an existing mill, then moving it back to form the same mill a second time (or any number of times), each time removing one of his opponent's men.

**Game goal or winning conditions.** A player wins by reducing the opponent to **two men** (where the opponent could no longer form mills and thus be unable to win), or by **leaving the opponent with no legal moves**. Note that **this game will never end in a tie**.

As shown in Figure 2, on the left-hand side, Player 1 loses the game because (s)he has only two pieces left. On the right-hand side, Player 1 also loses the game because (s)he cannot move any one of the white pieces.

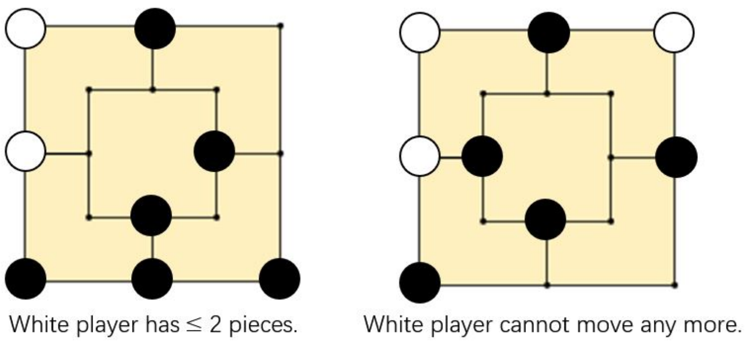

White player has ≤ 2 pieces.
White player cannot move any more.

Figure 2: Winning conditions of Six Men's Morris. A player wins by reducing the opponent to **two pieces** or by **leaving them without a legal move**.

**Mill examples.** As shown in Figure 3, the left board forms a mill since three black pieces are **lined up horizontally**. The right board forms no mill because there are no lines between the lower two pieces.

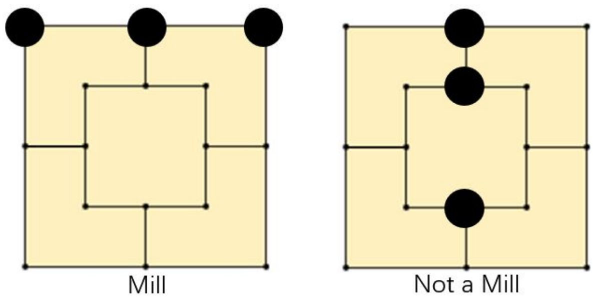


Figure 3: Visualization of a mill. Mill is formed when three men of the same color are lined up horizontally or vertically.

## 2.2 Problem Specification

This section describes the requirements, program design, function decomposition, game flow, etc.

### 2.2.1 Game Board Representation

There are 16 possible points on a board. Therefore, we use letters $a, b, c, ..., p$ to denote these board positions, as illustrated in Figure 4.

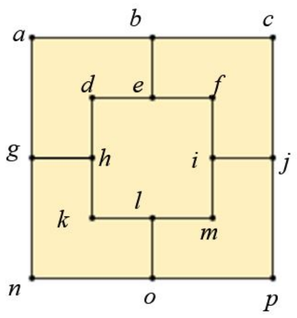
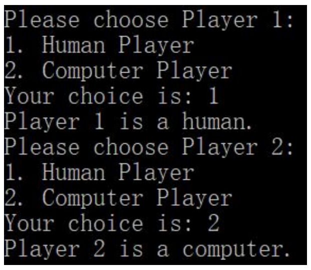


Figure 4: Board positions specification.

To encode the whole game board, we use an array of 16 characters. Each character in the array is either "." (Empty point), "#" (Player 1), or "@" (Player 2). Using this representation, the initial board is encoded as an array of 16 "." characters.

### 2.2.2 Input/Output Specification

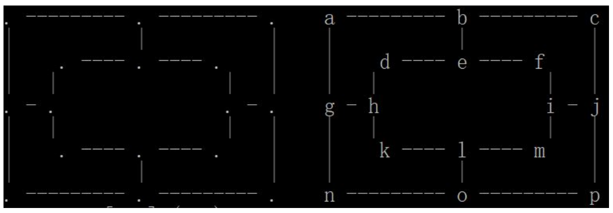
**Types of Player.** Users can choose Player 1 and Player 2 to be either human- or computer-controlled. You have to use the command line to request the types of the two players before starting the game (as shown in Figure 5).



```
Please choose Player 1:
1. Human Player
2. Computer Player
Your choice is: 1
Player 1 is a human.
Please choose Player 2:
1. Human Player
2. Computer Player
Your choice is: 2
Player 2 is a computer.
```

Figure 5: Setting for the type of players.

**Board.** Once the players are chosen, the empty board is outputted first. Note that we always output two views of the board at the same time. The left view represents the current board state and the right one shows the board positions map (where the letters $a, b, c, ..., p$ denote the positions on the board) as shown in Figure 6 because the board is not rectangular. You do not need to worry about the board's visualization because we have implemented the printing function in the provided code. Do not modify the printing function, or you will lose marks.



```
. ---------- . ---------- .      a ---------- b ---------- c
|            |            |      |            |            |
|     . ---- . ---- .     |      |     d ---- e ---- f     |
|     |             |     |      |     |             |     |
. - . .             . - .        g - h               i - j
|     |             |     |      |     |             |     |
|     . ---- . ---- .     |      |     k ---- l ---- m     |
|            |            |      |            |            |
. ---------- . ---------- .      n ---------- o ---------- p
```

Figure 6: Empty board.

**Every time** a player does a **legal** movement on the board, the program will output the current state of the board.

**Movements.** This program requires decisions of the following three operations from the human or computer players.

- PUT(*pos*): In Phase 1 of this game, the players put pieces in turn on the board. It takes an input of one letter as the position of the piece. If it is an illegal movement or input, the program will output a warning message "Invalid Put-Movement." and request for input again. The input message's format for players is like "Player 1 [Put] (pos): " and it takes a input of one letter.
- MOVE(*s, t*): In Phase 2 of this game, the players move the existing pieces on the board and the input takes 2 letters: source position and target position. If it is an illegal movement or input, the program will output a warning message "Invalid Move-Movement." and request for input again. The input message's format for players is like "Player 1 [Move] (from to): " and it takes an input of two letters separated by a space.
- REMOVE(*pos*): In either phase of the game, once a player forms a mill on the board, the player can do an **extra** movement, which removes an opponent's piece. In this movement, it takes an input of one letter position. If it is an illegal removal or input, the program will output a warning message "Invalid Remove-Movement." and request for input again. The input message's format for players is like "Player 1 [Remove] (pos): " and it takes a input of one letter.

A computer player always makes valid operations. The program should print out the input message of the corresponding operations and the decision by the computer player same as the message for human player.

**Overall Interface.** As shown in Figure 7, this is basically the whole interface of our game. The board will be displayed after **every** movement or play and each player has to make his decision for the requested movement. Although a computer player does not request for input, it outputs the decision in the same format as the human player does.

After the game ends, it will output a winning message for the winner like "Congratulation to the winner: Player 1!".

## 2.3 Required Classes and Functions

To let you get familar with OO-design, we provide an OO class design framework for this game. You should follow this OO-design and must not modify the prototypes of any of these classes and functions. You can design extra functions if you find it necessary.

- class **SixMensMorris**

  This class controls the overall game process, including phases 1 and 2 in Six Men's Morris, and the starting and ending of the game.

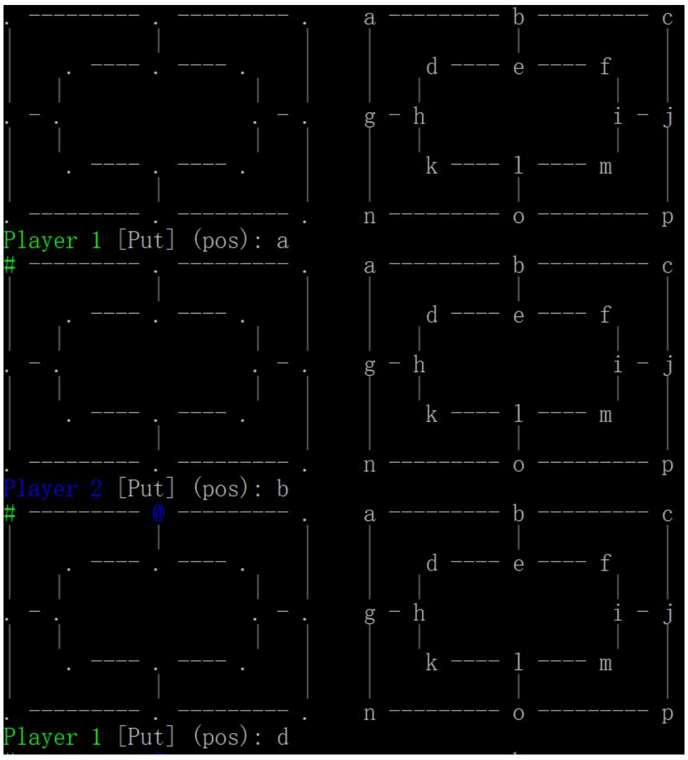
  - board = Board()

    This is a variable representing the game board.
  - players = [Player(1), Player(2)]

    This is an array with two variables representing the players.
  - num_play = 0

    This is a variable representing the number of total plays (one play denotes one movement by a player.). Each time a player does a movement, this variable is incremented by 1. We note that 1 round is equivalent to 2 plays.
  - next_player(self)

    This function returns the current player in this round.



```
. ---------- . ---------- .     a ---------- b ---------- c
|            |            |     |            |            |
|     . ---- . ---- .     |     |     d ---- e ---- f     |
|     |             |     |     |     |             |     |
.  -  .             .  -  .     g  -  h             i  -  j
|     |             |     |     |     |             |     |
|     . ---- . ---- .     |     |     k ---- l ---- m     |
|            |            |     |            |            |
. ---------- . ---------- .     n ---------- o ---------- p
Player 1 [Put] (pos): a
# ---------- . ---------- .     a ---------- b ---------- c
|            |            |     |            |            |
|     . ---- . ---- .     |     |     d ---- e ---- f     |
|     |             |     |     |     |             |     |
.  -  .             .  -  .     g  -  h             i  -  j
|     |             |     |     |     |             |     |
|     . ---- . ---- .     |     |     k ---- l ---- m     |
|            |            |     |            |            |
. ---------- . ---------- .     n ---------- o ---------- p
Player 2 [Put] (pos): b
# ---------- @ ---------- .     a ---------- b ---------- c
|            |            |     |            |            |
|     . ---- . ---- .     |     |     d ---- e ---- f     |
|     |             |     |     |     |             |     |
.  -  .             .  -  .     g  -  h             i  -  j
|     |             |     |     |     |             |     |
|     . ---- . ---- .     |     |     k ---- l ---- m     |
|            |            |     |            |            |
. ---------- . ---------- .     n ---------- o ---------- p
Player 1 [Put] (pos): d
```

Figure 7: The interface.

- opponent(self, player)
  This function returns the opponent of the player. If the argument is Player 1, then it will return Player 2, and vice versa.
- check_win(self, player)
  This function returns a Boolean variable representing whether the player wins the game. It calls board.check_win() in fact.
- start_game(self)
  This function controls the overall logic of the game.

• class **Board**

This class defines the game board and its properties such as the connected edges and all possible mills on the board, as well as a series of movements and legal movement checks on the board. The state of the board is controlled by the players.

- state
  This is the representation of the state of the game board, which stores an array of size 16.
- mills
  This variable pre-defines all possible mill configurations, each of which is represented as a 3-tuple on the board.
- edges
  This variable pre-defines all possible edges, each of which is represented as a 2-tuple on the board.
- print_board(self)
  This function prints the current stored board state. It would print two views – one shows the board state and the other one shows the reference board positions for the players. Please refer to the visualization of the board in Figure 7.

  - check_put(self, pos)
    This function checks whether the PUT-movement is legal. The argument is the putting position. The output is a Boolean variable.
  - check_move(self, s, t, player)
    This function checks whether the MOVE-movement is legal. The arguments are the source position, the target position, and the player. The output is a Boolean variable.
  - check_remove(self, pos, player)
    This function checks whether the REMOVE-movement is legal for the player. The arguments are the position on which the piece is expected to be removed and the player. The output is a Boolean variable.
  - put_piece(self, pos, player)
    This function does the PUT-movement on the board without checking.
  - move_piece(self, s, t, player)
    This function does the MOVE-movement on the board without checking.
  - remove_piece(self, pos, player)
    This function does the REMOVE-movement on the board without checking.
  - form_mill(self, pos, player)
    This function returns a Boolean variable representing whether it forms a mill at this position for the player.
  - check_win(self, player, opponent)
    This function returns a Boolean variable representing whether the current player wins. If the current player wins the game, then it returns True. Otherwise, it returns False. Please refer to the winning conditions.

- class **Player**

  This class defines the behaviors of players. The player can put pieces, move pieces and remove pieces on the board.

  - id
    Player's id. It is an integer 0 or 1.
  - symbol
    Player's symbol. Player 1 uses "#" but Player 2 uses "@".
  - board
    This is the game board for the player.
  - next_put(self)
    This function defines a PUT-movement for the player. It should be implemented by the **Human** and **Computer** classes. **It returns the position to which the piece is put because it might trigger the formation of a mill.**
  - next_move(self)
    This function defines a MOVE-movement for the player. It should be implemented by the **Human** and **Computer** classes. **It returns the position to which the piece moves because it might trigger the formation of a mill.**
  - next_remove(self, opponent) This function defines a REMOVE-movement for the player. It should be implemented by the **Human** and **Computer** classes.

- class **Human**

  This class extends the superclass **Player** to play the game, where the player's movements are from the input of the user.

- next_put(self)
  This function outputs a prompt and takes a single location input for a PUT-movement. Specifically, the input message's format is like "Player 1 [Put] (pos): " and it takes an input of one letter. It keeps checking whether it is a legal PUT-movement until the input is legal. After getting the correct input, it then does the movement on the board (by calling board.put_piece()). **It returns the position where the piece to put because it might trigger the formation of a mill.** When there is an error input, it will output a warning message "Invalid Put-Movement.".
- next_move(self)
  This function outputs a prompt and takes two location inputs (s, t) for a MOVE-movement. Specifically, the input message's format for players is like "Player 1 [Move] (from to): " and it takes an input of two letters separated by a space. It keeps checking whether it is a legal MOVE-movement until the input is legal. After getting the correct input, it then does the movement on the board (by calling board.move_piece()). **It returns the position to which the piece moves because it might trigger the formation of a mill.** When there is an error input, it will output a warning message "Invalid Move-Movement.".
- next_remove(self, opponent)
  This function outputs a prompt and takes a location input for a REMOVE-movement. Specifically, The input message's format for players is like "Player 1 [Remove] (pos): " and it takes a input of one letter. It keeps checking whether it is a legal REMOVE-movement until the input is legal. After getting the correct input, it then does REMOVE-movement on the board (by calling board.remove_piece()). It returns nothing. When there is an error input, it will output a warning message "Invalid Remove-Movement.".

- class **Computer**

  This class extends the superclass **Player** to play the game, where the player's movement is generated by the computer.

  - next_put(self)
    This function randomly generates a legal PUT-movement and does the movement on the board (by calling board.put_piece()). It outputs the same message like "Player 1 [Put] (pos): " and one letter representing his decision. **It returns the position where the piece locates because it might trigger the formation of a mill.** Note that the computer only generates a legal movement and thus print no warning message.
  - next_move(self)
    This function randomly generates a legal MOVE-movement (s,t) and then does the movement on the board (by calling board.move_piece()). It outputs the same message as the human player: "Player 1 [Move] (from to): " and two letters representing his decision. **It returns the position to which the piece moves because it might trigger the formation of a mill.** Note that the computer only generates a legal movement and thus print no warning message.
  - next_remove(self, opponent)
    This function randomly generates a legal REMOVE-movement and does REMOVE-movement on the board (by calling board.remove_piece()). It outputs the same message like "Player 1 [REMOVE] (pos): " and one letter representing his decision. It returns nothing. Note that the computer only generates a legal movement and thus print no warning message.

## 2.4 Your programming task

Overall there are six source files, one for each defined class and additional tools file: Board.py, Computer.py, Human.py, Player.py, SixMensMorris.py, utils.py. The files are templates. Your task is to **complete the missing parts of the code which are marked by "TODO"**. The programming environment is Python3 (Python3.5+) only.

1. Complete your name and student ID in every file.
2. Based on the above problem specification, complete the game program by implementing the above member functions.
3. The program starts by running "python3 SixMensMorris.py". You should design enough corner cases to test your program.

Things to note:

1. Apart from completing these member functions, **Do Not** change any other parts of our provided source code.
2. Please strictly follow the Input/Output specification in Subsection 2.2.2. You might easily get the I/O details by directly going through the provided code. Otherwise, your marks can be deducted.
3. For better visualization, we added color in the output and you can find the color functions in the file "utils.py". You should not change the color or its uses in the source code. Otherwise, it might cause errors when we are comparing the output content by machines.
4. In Python, there is a library that you can use to debug the code, *i.e.*, ipython. You can simply insert "`from IPython import embed; embed()`" and run the program. Then you can debug your program like checking the variable values or debugging the code you write while the code is running.

## 3 Task 2: Demonstrating Advantages and Disadvantages of Dynamic Typing

These are commonly-claimed advantages and disadvantages of Dynamic Typing:

1. Advantages
   - More generic code can be written. In other words, functions can be defined to apply on arguments of different types.
   - Possibilities of mixed type collection data structures.
2. Disadvantage
   - Loss of type checking at compile time.

Please provide concise example codes *designed by yourself or extracted from any code you write for this assignment* to demonstrate the advantages and disadvantage of Dynamic Typing mentioned above. You are welcome to provide other advantages or disadvantages along with code fragment for extra bonus points.

## 4 Task 3: Save The Tribe

This task is to build a game in Python. A Java implementation of the game is given. You need to understand its behavior and re-implement it with Python. After finishing this task, you will have experienced a special feature of Python called Duck Typing, which is possible only in a dynamically typed language. And you will see a difference between the Java and Python implementations, the former of which does not support Duck Typing.

## 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 characteristics and behavior) are essentially interpreted as the objects' types. Duck Typing is a new typing paradigm in dynamically-typed (late binding) languages that allows 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 concise.

## 4.2 Background

In Ancient Rome, a tribe is attacked by alien invaders. The only way to wipe out these invaders is to seek an artifact from a desert nearby. However, the desert is very dangerous with many monsters residing there.

With the mission to save the tribe, a soldier (the player) comes to the desert and starts his journey to seek the artifact. Each monster in the desert lives in its own cave and the artifact is taken by one of the monsters. The soldier has to enter the cave and kill the monster in a topological order so that (s)he could get the keys to enter other caves. Figure 8 illustrates the pre-defined topological order, where each node represents a cave. If the soldier wants to enter cave 2, (s)he has to kill the monster inside cave 5 or cave 1 first to get the key for cave 2. The monster inside cave 7 keeps the artifact that the soldier wants to collect.

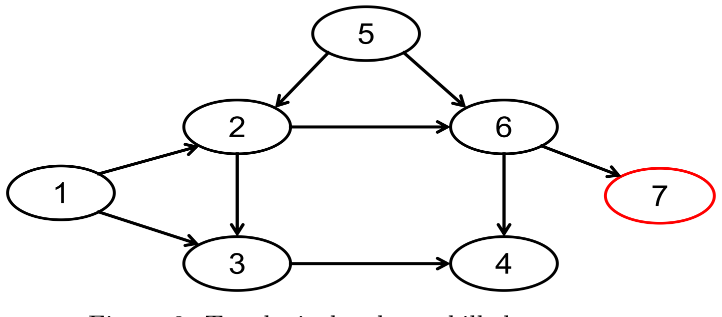
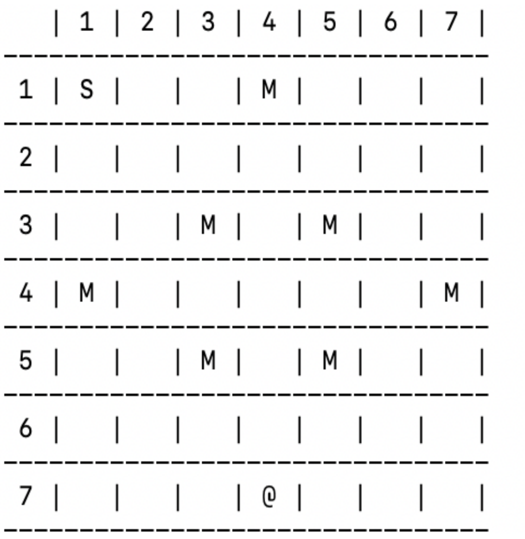


Figure 8: Topological order to kill the monsters.

## 4.3 Task Description

At the beginning, there are 7 monsters and a spring in the desert. All of them are distributed in different positions, as illustrated in Figure 9. The health capacity of the soldier is 100, the health capacity of the monsters is a random number in the range of 30–70, which must be in multiples of 10. The soldier can drink the spring water to recover fully his health. Before exploring the desert, the soldier is equipped with 2 elixirs to increase his health during fight, as well as the keys to enter cave 5 and cave 1.



```
  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
--------------------------------
1 | S |   |   | M |   |   |   |
--------------------------------
2 |   |   |   |   |   |   |   |
--------------------------------
3 |   |   | M |   | M |   |   |
--------------------------------
4 | M |   |   |   |   |   | M |
--------------------------------
5 |   |   | M |   | M |   |   |
--------------------------------
6 |   |   |   |   |   |   |   |
--------------------------------
7 |   |   |   | @ |   |   |   |
--------------------------------
```

Figure 9: Map of the desert. Symbols of $M$, $S$ and @ represent the monsters, soldier and spring respectively.

- At each iteration, the soldier can move to the next grid cell in the desert.
  - When meeting a cave, the monster inside will check whether the soldier has the key to enter or not.
    * If yes, the soldier will fight with the monster in a round-based manner. In each round, the soldier has three options:
      1. Attack: The health of the monster will be damaged by 10. If the monster is still alive, the health of the soldier will also be damaged by 10. Enter next round. *Note that the soldier always attacks first.*
      2. Escape: The soldier escapes and the health of monster will be recovered to its health capacity automatically. The fight is over.
      3. Use Elixir: The soldier uses elixir, which could increase her/his health by 15 to 20 randomly. Enter next round.

      If the soldier successfully kills the monster (health of monster is reduced to 0 or less), (s)he will get the dropped item(s) from the monster. Otherwise, when the health of the soldier is reduced to 0 or less, the soldier is dead and the game is over.
    * If no, the monster will tell the soldier where (s)he can collect the key to enter this cave.
  - When meeting the spring, the soldier will drink the water and recover his health to 100. The soldier can only the water for one time.
- When the soldier gets the artifact from the monsters, the game will be over.

You should read and execute the given Java program to understand its behavior and design. Then please replicate all the behavior of the given Java Program in Python with Duck Typing, following the same class design. You should not introduce extra instance variables or instance methods. You program should run by calling `python3 SaveTheTribe.py`. You will also be evaluated on your programming style.

## 5 Task 4: Strengthen Yourself

A merchant comes to the desert. The soldier could buy shield or elixir with coins to strengthen himself, which enables him to kill the monster more easily. After killing each monster, the soldier could get one coin.

## 5.1 New Game Elements

At the beginning, there are 7 monsters ($M$), a merchant ($), and a spring (@) in the desert. The merchant is located at (7,7) on the map. Note that the soldier has no coins initially.

- At each iteration, the soldier can move to the next grid cell in the desert.
  - When meeting a cave, the monster inside will check whether the soldier has the key to enter or not.
    * If yes, the soldier will fight with the monster in a round-based manner. In each round, the soldier has three options:
      1. Attack: The health of the monster will be damaged by 10. If the monster is still alive, the health of the soldier will also be damaged by 10 - his defence value, but the damage value is no less than zero. Enter next round. *Note that the soldier always attacks first.*
      2. Escape: The soldier escapes and the health of monster will be recovered to its health capacity automatically. The fight is over.
      3. Use Elixir: The soldier uses elixir, which could increase her/his health by 15 to 20 randomly. Enter next round.

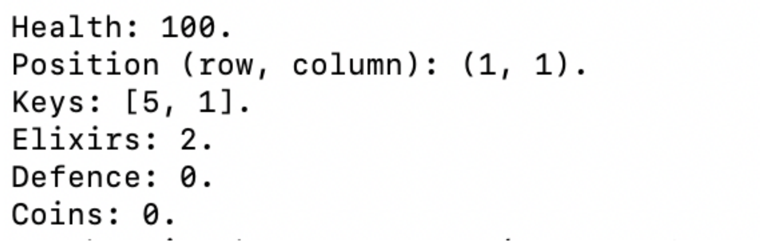
      If the soldier successfully kills the monster (health of monster is reduced to 0 or less), he will get the dropped item(s) and one coin from the monster. Otherwise, when the health of the soldier is reduced to 0 or less, the soldier is dead and the game is over.
    * If no, the monster will tell the soldier where (s)he can collect the key to enter this cave.
  - When meeting the spring, the soldier will drink the water and recover his health to 100.
  - When meeting the merchant, the soldier has three options:
    1. The soldier can buy an elixir with one coin.
    2. The soldier can buy a shield with two coins. Each shield can improve the defence value of the soldier by 5.
    3. Leave. (Note that the soldier will also automatically leave if (s)he doesn't have enough money to buy what (s)he chooses to buy).
- When the soldier gets the artifact from the monsters, the game will be over.

## 5.2 Program Enhancement

You are now required to enhance both the Java and Python implementations of Task 3, according to the requirements below:

1. You need to add a **Merchant** class. A Java template is provided for your reference.
2. You need to make appropriate modifications to the **Soldier** and **Monster** classes, for which you need to use the power of inheritance as much as possible. For example, for the modification of the **Soldier** class, you need to *add a subclass* of the **Soldier** class, and name it as **Task4Soldier** class, instead of modifying the **Soldier** class directly. But please note that you are allowed to modify the instance variable "health" in **Soldier** class into a protected variable.
3. You need to make appropriate modifications to the **SaveTheTribe** and **Map** classes in Java, but only need to modify **SaveTheTribe** class in Python, due to the usage of Duck Typing. For the changes on these two classes, you are allowed to modify the original class design of Task 3.
4. You are required to name each class that you have modified or added for Task 4 as **Task4xxx**. For example, if you modify the **SaveTheTribe** class, please name it as **Task4SaveTheTribe**, and name the corresponding file as Task4SaveTheTribe.py or Task4SaveTheTribe.java.

5. You are required to display all the attributes of the soldier in each iteration, as illustrated in Figure 10.
6. Your program should be run by calling `python3 Task4SaveTheTribe.py` and `javac Task4SaveTheTribe.java`.



```
Health: 100.
Position (row, column): (1, 1).
Keys: [5, 1].
Elixirs: 2.
Defence: 0.
Coins: 0.
```

Figure 10: Displayed information in each iteration.

## 6 Report

Your simple report should answer the following questions within TWO A4 pages.

1. Providing example code and necessary elaborations for demonstrating advantages and disadvantages of Dynamic Typing as specified in Task 2.
2. Using codes for Task 3, give two scenarios in which the Python implementation is better than the Java implementation. Given reasons.
3. Using codes for Task 4, illustrate further advantages of Duck Typing.

## 7 Submission Guidelines

Please read the guidelines CAREFULLY. If you fail to meet the deadline because of submission problem 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 and Java.

```
/*
 * 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
 */
```

The sample file header is available at

```
http://course.cse.cuhk.edu.hk/~csci3180/resource/header.txt
```

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 report should be named "receipt.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 task1.zip task3_python.zip task4_java.zip \
   task4_python.zip report.pdf receipt.pdf
   ```

6. Your `task1.zip` should include

   ```
   Board.py Computer.py Human.py Player.py SixMensMorris.py utils.py
   ```

7. Your `task3_python.zip` should include

   ```
   Pos.py Cell.py Map.py Spring.py SaveTheTribe.py Soldier.py Monster.py
   ```

8. Your `task4_python.zip` should include

   ```
   Pos.py Cell.py Map.py Spring.py SaveTheTribe.py Soldier.py Monster.py
   Task4SaveTheTribe.py Task4Merchant.py Task4Soldier.py Task4Monster.py
   ```

9. Your `task4_java.zip` should include

   ```
   Pos.java Cell.java Map.java Spring.java SaveTheTribe.java Soldier.java
   Monster.java Task4Map.java Task4SaveTheTribe.java Task4Merchant.java
   Task4Soldier.java Task4Monster.java
   ```

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

    ```
    gzip tmchan.tar
    ```

11. `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
    ```

12. Please submit your assignment using your Unix accounts.
13. An acknowledgement email will be sent to you if your assignment is received. **DO NOT** delete or modify the acknowledgement email. You should contact your TAs for help if you do not receive the acknowledgement email within 5 minutes after your submission. **DO NOT** re-submit just because you do not receive the acknowledgement email.
14. You can check your submission status at

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

15. You can re-submit your assignment, but we will only grade the latest submission.
16. Enjoy your work :>