D&D Dice Roller

CSCI 315, Programming Languages

Aaron Pfister

Spring 2018

Ruby

This lab is designed to use the Ruby programming language. Directions for downloading and installing ruby: https://www.ruby-lang.org/en/documentation/installation/

Overview

The goal of this lab is to create a 2 player (Human vs. Machine, Human vs. Human, Machine vs. Machine), turn based dice rolling game. The object of the game is to knock your opponent down more times than they knock you down. My inspiration for this lab comes from turn based video games and a dungeons and dragons video game I used to play. I also believe the turn based style is easier to think about, plan out and implement in code. Since Ruby is object-oriented, this lab will utilize classes and inheritance.

Attack Module

First you will need to create the module for Attack, which will hold methods for using a weapon. Attack will hold 2 methods: **try_hit** and **damage**. The **try_hit** method will take an integer, hit_chance, as its parameter and use the rand() function to pick a number between 1 and 100. The method will return true if the random number is greater than hit_chance and false if it is less. The **damage** method will take two integers for its parameters, damage_modifier and dice. The method uses the rand function to pick a number between 1 and dice and multiplies this number by the damage modifier, and returns the result.

Weapon Class

Next you will start creating the class for Weapon. The Weapon class has four instance variables: **name, hit_chance, damage_modifier, dice**. **hit_chance** stores the lowest number a player can roll and still hit their opponent. **dice** is the size of the dice and **damage_modifier** is the number multiplied by the dice to determine overall damage of a swing. The constructor will take a string and 3 integers to be set to the corresponding member variables.

For the Weapon class, you will need to create the **swing** method that returns the total damage when **try_hit** returns true, and 0 when it returns false.
You can use the method **attr_accessor** in order to create getter and setter functions for member variables of your class. For example:

```
class Class
  attr_accessor :m_var #single variable
    attr_accessor :m_var2, :m_var3 #multiple variables

    def initialize(m_var, m_var2, m_var3)
      @m_var, @m_var2, @m_var3 = m_var, m_var2, m_var3
    end
end

c1 = Class.new(1, 2, 3)
puts "#{c1.m_var}" # => 1
c1.m_var = 2
puts "#{c1.m_var}" # => 2
```

Player Class

The Player class contains 2 subclasses: Human and Machine.
Player will have 3 instance variables: **health, weapons_list, wins**. **weapons_list** holds the 3 different weapons available to the player at the beginning of each round. Both**health** and **wins** will be updated and maintained throughout the game. The constructor will take an integer and an array of weapons to be set to **health** and **weapons_list**, and **wins** will be set to 0.

Human Class

Human contains 1 more instance variable: **name**. The constructor will take and integer on top of the Player class constructor.
The **set_weapon** method will be necessary for this class. This method will prompt the user to choose a weapon and then set the Players weapon to that choice.

Tip: Using attr_accessor :weapon will provide functions available to the Weapon class.

Machine Class

The Machine class constructor takes the same parameters as the Player class, but, upon instantiation, will set a random weapon from **weapons_list** and **name** to Machine.
The **set_weapon** method will choose a random weapon from **weapons_list** and set it as the new weapon.

Game Layout

1. Each game will start with a picture using the Catpix gem. Instructions and information about the gem can be found here. It is as simple as:

```
require 'catpix'

Catpix::print_image "dice.jpg",
  :limit_x => 0.5,
  :limit_y => 0,
  :center_x => true,
  :center_y => true,
  :bg => "white",
  :bg_fill => false,
  :resolution => "high"
```

   Note: I was not able to find a better gem to use. Unfortunately the picture quality is not great, but increasing terminal window size will result in better quality.

2. Ask the user to input 3 **weapon**s for the Machine **weapons_list**, then 3 weapons for the Human weapons list. You can assume that there will always be three weapons and there will be no bad input.

3. Ask the user to enter the starting **health** and **name** for each Player. After this, Player objects can be declared. If the user enters "Machine", the type of Player will be Machine. Next, ask the user how many rounds they would like to play.

4. After the initial information is recorded, the game may begin. For each round:

(a) Set weapon of each player.

(b) Print out picture signifying the start of a round.

(c) Make each player swing their weapon.

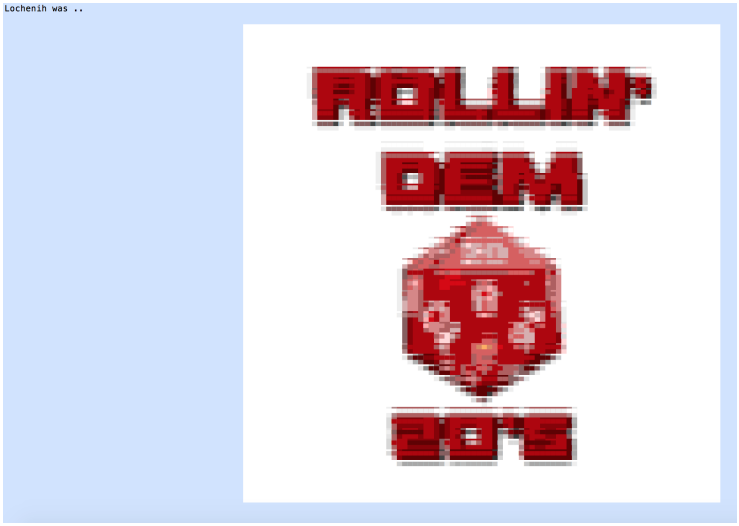
(d) Update/maintain **health** and **wins** of each player.

(e) Repeat c and d until the **health** of one of the Players reaches 0.

5. Once each round has been played, a summary of each round should be printed to the screen. For example:

```
Lochenih won round 1 using a Longsword.
Lochenih won round 2 using a Mace.
Lochenih won round 3 using a Rapier.
```

6. Lastly, the winner should be announced, and the program will end.



Testing and Troubleshooting

Since the outcome of each test may change from time to time, you should make sure your program is printing the correct strings at the correct times. For each test, your output should match up to the beginning of each round. I recommend using the Internet to become familiar with Ruby before starting the lab.
You can use the piping method in order to run test inputs:

ruby game.rb < tests/t01.in

Files Provided

Each student will be given a starter pack containing 2 files (starter_weapon.rb and starter_player.rb), a tests folder, and images to use in the program.

Submission

Each student will work on the project individually and submit your program (**weapon.rb player.rb game.rb**) via Blackboard or email. Programs will be graded manually based on use of classes, Catpix, and other features of the Ruby language, along with commenting and cleanliness of code.