Role-Playing Game (RPG)

psp-09-08

1 Overview

This lab assignment will provide a (hopefully slightly amusing) context for getting more practice with defining classes based on natural language specifications (the instructions given), inheritance mechanics, and creating and using objects. It's important to note that this wouldn't necessarily be exactly how a game would be designed, as there are some sub-optimal design choices expressed in the instructions and template code to make things more interesting from an object-oriented programming perspective.

2 Learning Outcomes

By the end of this project students should be able to:

- read and write programs that define multiple objects;
- read and write programs that utilize inheritance;
- read and write programs that utilize composition relationships;
- work effectively with a partner using pair-programming;
- write an effective report that describes the students' problem solving process.

3 Pre - Lab Instructions

Do this part before you come to lab:

- Read Problem Space Chapter 9: Classes and Objects, paying special attention to the inheritance section (9.5).
- Read the lab instructions and given code, and then sketch the UML diagram for the RPG program.
- Write down any questions you have about the code and how the finished program should work
- Be prepared to show your UML to the lab aide at the beginning of lab.

4 Lab Instructions

The definition of character classes lie at the heart of role-playing games: Character classes define the attributes of a character and that character's abilities. Most role-playing games are a simulation of the interaction of these attributes and abilities with the world or other characters, with randomness (expressed through rolls of a die with the following convention: "3d8" means rolls a 8-sided die 3 times, and add up the total) being a core part of the uncertainty in the simulation. The object-oriented paradigm was originally developed to support simulations, so it's a pretty nice fit for use in designing games. In this lab assignment, we'll use object-oriented programming to develop the foundation for a (very simple; we only have one lab session) game that simulates combat between two characters of different classes.

The code below will give you the structure for an RPG-style combat program and some example implementations, just to make your lives a bit easier and make the lab go quicker. Your job will be to fill-in the missing code for class and method definitions, as guided by the docstrings and instructions found throughout the code. Places where you have to add code are indicated with "TODO".

```
1  import random
2
3
4  class DiceRoller:
5      # A utility class for dice rolling.
6
7      def roll(self, times, sides):
8          # Rolls times number of sides-sided dice; returns the total.
9          total = 0
10         for i in range(times):
11             roll = random.randint(1, sides)
12             total += roll
13         return total
14
15 r = DiceRoller()
16
17
18 class Attack:
19     # Encapsulates the concept of an attack.
20
21     def __init__(self, name, number_of_die, sides_of_die, damage_type):
22         """Creates an attack with privates attributes.
23
24             Adds _name, _sides, _number, and _type
```

```
25          to self, with values provided by the arguments.
26
27          """
28          self._name = name
29          self._sides = sides_of_die
30          self._number = number_of_die
31          self._type = damage_type
32
33      def get_attack_type(self):
34          """Returns the type of attack, as a string."""
35          return self._type
36
37      def get_damage(self):
38          """Returns a damage value for this attack.
39
40          Rolls _number number of _sides sided dice, using
41          DiceRoller r, and returns the resulting value.
42
43          """
44
45          # TODO
46
47      # Write getters and setters for the rest of the Attack class variables.
48
49      # TODO
50
51
52
53
54  class Adventurer:
55      # Encapsulates the concept of an adventurer.
56
57      def __init__(self, name, hit_points, defense, magic_defense, initiative):
58          """Creates an adventurer with private attributes.
59
60          Adds _name, _hit_points, _defense, _magic_defense, and _initiative to
61          self, with values provided by the arguments.
62
63          """
64
65          # TODO
66
67      def is_alive(self):
68          """Returns True if this object has more than 0 hit points."""
69
70          # TODO
```

```
71
72      def roll_initiative(self):
73          """Returns a random integer between 0 and this object's _initiative value."""
74
75          # TODO
76
77      def take_damage(self, amount, damage_type):
78          """Applies damage to this object's attributes.
79
80          If the damage type is "physical", reduce this object's hit points by
81          what is left after reducing amount by the object's defense. If the damage
82          type is "magic", reduce the object's hit points by what is left after
83          reducing amount by the object's magic_defense. (For instance, let's say an
84          Adventurer with 10 defense takes 18 as the amount of damage and "physical"
85          as the damage_type. We would subtract the defense from the amount, so that
86          the Adventurer really only takes 8 damage. Then, we subtract 8 from the
87          Adventurer's hit points. An Adventurer's defense and magic defense never
88          decrease.)
89
90          Of course, make sure that the damage applied can never be less than 0
91          (so if an adventurer with 4 defense takes 3 physical damage, they really
92          take 0 damage overall, not −1), and that an adventurer can never have
93          negative hit points (so if an adventurer with 3 hit points remaining takes
94          5 damage, they now have 0 hit points, not −2).
95
96          Also prints out information about the damage application process,
97          something like 'Gandalf suffers 1 damage after 4 defense and has
98          12 hit points left.'
99
100         """
101
102         # TODO
103
104     # Write getters and setters for all class variables
105
106     # TODO
107
108
109 class Fighter(Adventurer):
110     """Encapsulates a Fighter class, inheriting from Adventurer.
111
112     Fighters are defined by 40 health point (_HP), defense of 10 (_DEF),
113     and magic defense (_MAG_DEF) of 4, which are stored as class variables.
114
115     Note: It is convention to capitalize the names of variables that are to
116     remain constant, and note the use of an underscore to indicate
```

```
117         that the variables are private.
118
119         """
120         _HP = 40
121         _DEF = 10
122         _MAG_DEF = 4
123
124         def __init__(self, name, initiative):
125             """Creates a Fighter object.
126
127             Calls the superclass __init__ method with name, _HP, _DEF,
128             _MAG_DEF, and initiative.
129
130             Also adds a variable _melee that references an instance of the
131             Attack class. For our simple game, this lets us create fixed
132             attacks such as Attack("Slash", 1, 8, "physical").
133
134             """
135             super().__init__(name, Fighter._HP, Fighter._DEF, Fighter._MAG_DEF, initiative)
136             self._melee = Attack("Slash", 1, 8, "physical")
137
138         def strike(self):
139             """Calculates and returns information about a physical strike from this object.
140
141             Returns a tuple, consisting of the damage value and the damage type. The
142             damage value is obtained by calling the get_damage() method on
143             _melee, while the type is obtained by calling get_attack_type() on
144             _melee.
145
146             Also prints out information about the strike, something like
147             'Aragorn attacks with Slash for 5 physical damage.'
148
149             """
150
151             # TODO
152
153         def __str__(self):
154             """Returns a string representation of this object, something like
155             'Aragorn with 40 hit points and a Slash attack (1d8).' The abbreviation
156             in the parentheses represents the number and type of dice used for the
157             attack, so 1d8 means 1 die with 8 sides is rolled.
158
159             """
160
161             # TODO
162
```

```
163
164 class Wizard(Adventurer):
165     """Encapsulates a Wizard class, inheriting from Adventurer.
166
167     Wizards have 20 HP, defense of 4, and magic defense of 10, all
168     stored as class variables using the same naming conventions as Fighter.
169
170     """
171
172     # TODO
173
174     def __init__(self, name, initiative):
175         """Creates a Wizard object. This is done in exactly the same way as
176         it was done in the Fighter class.
177
178         Calls the superclass __init__ method with name, _HP,
179         _DEF, _MAG_DEF, and initiative.
180
181         Also adds a variable _spell that references an instance of the
182         Attack class. For our simple game, this lets us create fixed
183         attacks such as Attack("Fireball", 3, 6, "magic").
184
185         """
186
187         # TODO
188
189     def cast(self):
190         """Calculates and returns information about a spell from this object.
191
192         Returns a tuple, consisting of a damage value and a damage type. The
193         values are obtained by calling get_damage() and get_attack_type() on
194         _spell.
195
196         Also prints out information about the attack, much like in Fighter's
197         attack method.
198
199         """
200
201         # TODO
202
203     def __str__(self):
204         """Returns a string representation of this object, much like
205         in Fighter's __str__ method.
206
207         """
208
```

```
209             # TODO
210
211
212 if __name__ == "__main__":
213     # Create a Fighter object by providing a name and
214     # initiative value to the Fighter constructor, and print the object out
215     a = Fighter("Aragorn", 20)
216     print("Created: ", a)
217
218     # Create a Wizard object, using the Wizard constructor with similar
219     # information as that provided for Fighter, and print the object out
220
221     # TODO
222
223     # Create a variable to keep track of the rounds of combat
224
225     # TODO
226
227     # As long as both combatants are alive, keep fighting rounds
228
229     # TODO
230
231         # Roll initiative for both combatants; whichever one has the highest
232         # initiative gets to attack this round. The other one... sits there
233         # If the initiative values are equal, roll them again until one is higher.
234         # An attack works by getting a damage value and type by using the cast() or strike()
235         # method from whatever object won initiative. Then, call take_damage() on the
236         # losing object using the values that cast()/strike() returned
237
238         # TODO
239
240         # Don't forget to increment your round counter
241
242         # TODO
243
244     # Print out a message indicating which combatant is still alive and their
245     # remaining hit points
246
247     # TODO
```

Below is an example of what a run of the final product might look like:

```
Created: Aragorn with 40 hit points and a Slash attack (1d8)
Created: Gandalf with 20 hit points and a Fireball spell (3d6)

** ROUND 1
Aragorn wins initiative!
Aragorn attacks with Slash for 5 physical damage
Gandalf suffers 1 damage after 4 defense and has 19 hit points left

** ROUND 2
Gandalf wins initiative!
Gandalf attacks with Fireball for 11 magic damage
Aragorn suffers 7 damage after 4 magic defense and has 33 hit points left

** ROUND 3
Gandalf wins initiative!
Gandalf attacks with Fireball for 9 magic damage
Aragorn suffers 5 damage after 4 magic defense and has 28 hit points left

** ROUND 4
Gandalf wins initiative!
Gandalf attacks with Fireball for 9 magic damage
Aragorn suffers 5 damage after 4 magic defense and has 23 hit points left

** ROUND 5
Aragorn wins initiative!
Aragorn attacks with Slash for 4 physical damage
Gandalf resists all damage!

** ROUND 6
Aragorn wins initiative!
Aragorn attacks with Slash for 8 physical damage
Gandalf suffers 4 damage after 4 defense and has 15 hit points left

** ROUND 7
Aragorn wins initiative!
Aragorn attacks with Slash for 7 physical damage
Gandalf suffers 3 damage after 4 defense and has 12 hit points left

** ROUND 8
Aragorn wins initiative!
Aragorn attacks with Slash for 8 physical damage
Gandalf suffers 4 damage after 4 defense and has 8 hit points left

** ROUND 9
Gandalf wins initiative!
Gandalf attacks with Fireball for 17 magic damage
Aragorn suffers 13 damage after 4 magic defense and has 10 hit points left

** ROUND 10
Gandalf wins initiative!
Gandalf attacks with Fireball for 16 magic damage
Aragorn suffers 12 damage after 4 magic defense and has 0 hit points left

Gandalf has won with 8 hit points left!
```

When you have completed the lab run pep8 against your code until all formatting errors have been corrected and your code is PEP 8 compliant. See the Getting Started lab if you need instructions on running the program, or the pep8 documentation found here.

5 Lab Report

Each pair of students will write a single lab report together and each student will turn in that same lab report on BBLearn. Submissions from each student on a pair should be identical.

Your lab report should begin with a preamble that contains:

- The lab assignment number and name
- Your name(s)
- The date
- The lab section

It should then be followed by four numbered sections:

1. Problem Statement

In this section you should describe the problem in **your** own words. The problem statement should answer questions like:

- What are the important features of the problem?
- What are the problem requirements?

This section should also include a reasonably complete list of requirements in the assignment. Following your description of the problem, include a bulleted list of specific features to implement. If there are any specific funtions, classes or numeric requirements given to you, they should be represented in this bulleted list.

2. Planning

In the second section you should describe what planning you did in order to solve the problem. You should include planning artifacts like sketches, diagrams, or pseudocode you may have used. You should also describe your planning process. List the specific data structures or techniques you plan on using, and why.

3. Implementation and Testing

In the third section you should describe how you implemented your plan. As directed by the lab instructor you should (as appropriate) include:

- a copy of your source code (Submitted in BBLearn as a .py file)
- a screen shot of your running application / solution
- results from testing

4. Reflection

In the last section you should reflect on the project. Consider different things you could have done to make your solution better. This might include code organization improvements, design improvements, etc.

You should also ask yourself what were the key insights or features of your solution? Were there alternative approaches or techniques you could have employed? How would these alternatives have impacted a different solution?

5. Partner Rating

Every assignment you are required to rate your partner with a score -1, 0 or 1. This should be submitted in the comment section of the BBLearn submission, and not in the report document. You do not have to tell your partner the rating you assign them. A rating of 1 indicates that your partner was particularly helpful or contributed exceptional effort. A rating of 0 indcates that your partner met the class expectations of them. Rating your partner at -1 means that they refused contribute to the project, failed to put in a resonable effort or actively blocked you from participating. If a student recieves three ratings of -1 they must attend a mandatory meeting with the instructor to dicuss the situation, and recieving additional -1 ratings beyond that, the student risks losing a letter grade, or even failing the course.

Colophon