

CS3010-3 Python, Fall 2012: Problem Set 3

Daniel Bauer

Total points: 25

Due date: Feb 16th, 6pm

Submission instructions:

Place the files for all problems in a directory named `[your_uni]_week[X]`, where X is the number of the problem set. For instance if your uni is xy1234 and you are submitting the problem set for the first week, the directory should be called xy1234_week1. Either zip or tar and gzip the directory (using `tar -c xy1234_week1 | gzip > xy1234_week1.tgz`) and upload it to the directory for problem set X on the Courseworks page for this class.

Please pay attention to the general guidelines/homework policy on the course website.

Part1 (15 points) - Classes, Methods, Attributes, Instances

Create a module 'restaurant' which contains the following classes:

(a) Create a class `Guest` that represents a hungry guest at a restaurant.

- Guest instances have an attribute `hunger`, which indicates how hungry the guest is (how many portions of food he can eat).
- Guest instances have an attribute `name`, which stores the name of the guest.
- `name` and the initial `hunger` value are passed as parameters when a `Guest` object is created, e.g.

```
>>> g = Guest('John', 5)
```

creates a `Guest` with a `hunger` value of 5 (the guest can eat 5 portions) .

- Guest instances have a method `eat(self)`. When this method is called, the guest eats one portion of food, decrementing his `hunger` attribute. When the `hunger` value becomes 0, the `Guest` burps. The `eat(self)` method also returns the `hunger` value after eating.

```
>>> g.eat() # g.hunger was 1
John: Burp!
0
```

(b) Create a class `Restaurant` with base class `list`. Each element in a `Restaurant` represents a table.

Each `Restaurant` instance has the following attributes:

- The attribute `size` stores the number of tables in a `Restaurant`. When the `Restaurant` instance is created, a `size` parameter is passed and `size` elements of the `Restaurant` are initialized to `None` to indicated that the corresponding table is not occupied.
- A method `seat(self, guest)` that seats an arriving guest (an instance of class `Guest`) at the first empty table (i.e. inserts the object `guest` as an element of the `Restaurant`). If there is no available table, the method prints 'No free table.' and returns `False`. Otherwise it prints a message indicating which guest is seated to which table and returns `True`.

- A method `serve(self)`. Restaurants are sushi restaurants and can only serve one portion to one guest at a time. The `serve(self)` method finds the first table with a hungry guest, prints a message indicating which guest is being served, and calls the guest's `eat(self)` method. If the guest is not hungry any more she is removed from the table (the corresponding element is reset to `None`). If there is no guest to serve, the `serve(self)` method prints 'No guest to serve'.

You can test your `Guest` and `Restaurant` classes by running `test_restaurant.py`. The program creates a restaurant of size 2 and three guests Mary (hunger 3), John (hunger 2) and Alice (hunger 1). It then repeatedly tries to seat a guest and serve until all guests have been seated. The script should produce the output:

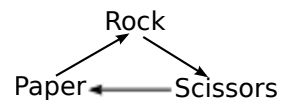
```
Seating guest Mary at table 0.  
Serving guest Mary.  
Seating guest John at table 1.  
Serving guest Mary.  
No free table.  
Serving guest Mary.  
Mary: Burp!  
Seating guest Alice at table 0.  
Serving guest Alice.  
Alice: Burp!
```

- (c) The restaurant in part (b) has a problem that's rather annoying for the guests. Create a class `FancyRestaurant` with base class `Restaurant`, which overloads the `serve(self)` method and fixes this problem.

Part 2 (10 points) - Overloading Special Methods

In this problem you will implement a very basic version of the popular game Rock-Paper-Scissors using object oriented programming concepts. In this game two players show one of three hand gestures (called 'rock', 'paper', or 'scissors') on the count of three. The player whose gesture 'defeats' the gesture of his opponent wins. Each gesture defeats one other gesture and is defeated by the third. Specifically the following 'defeats' relation holds between the three gestures.

- Rock blunts scissors.
- Scissors cut paper.
- Paper covers rock.



- (a) Create a module 'gestures' that defines three classes Rock, Scissors, and Paper.

Make all Rock, Scissors, and Paper instances comparable by overloading the classes' `__cmp__(self, other)` methods.

$x > y$ should return True if x defeats y . $x < y$ should return True if y defeats x . $x == y$ should return True if x is the same gesture as y .

For instance, your program should support the following comparisons:

```
>>> Rock() > Scissors()
True
>>> Scissors() > Paper()
True
>>> Rock() < Paper()
True
>>> Paper() == Paper()
True
```

- (b) Create a class Player, with a method `play(self)` that returns an instance object of either Rock, Paper, or Scissors. Your method can randomly select one of the gestures (you may want to have a look at the choice function in the random module ¹).
- (c) Implement a class HumanPlayer with base Player, that overloads `play(self)` to read in the user's choice of a gesture from the keyboard instead of randomly selecting a move.
- (d) Write a main function that creates a HumanPlayer and a computer Player, repeatedly requests a gesture from each player instance, compares the result and keeps track of the score. The score and chosen gestures should be printed after each turn (hint: implement the gesture classes' `__str__(self)` method).

¹<http://docs.python.org/library/random.html>