CS 120 (Fall 21): Introduction to Computer Programming II

Long Project #6

due at 5pm, Tue 5 Oct 2021

1 Schedule Note

The Midterm is coming up! This project is to help you get some experience with classes, before you have to take the Midterm (which will include them).

2 Overview

In this project, you will be practicing by creating a set of simple classes. You will create three different Python files; `classes_prob1.py` will have three small classes (`Simplest, Rotate, Band`), while `classes_prob2.py` and `classes_prob3.py` will each have one class each (`Color`, and `Room`, respectively). Our test code will import your files, and then use your classes, testing them to see if they work properly.

3 `class Simplest`

`Simplest` is a trivial object, with three public fields: `a,b,c` . It has a constructor which takes those three as parameters (in the order `a,b,c`), and sets the public fields in the object. It has no other methods.

Define this class inside `classes_prob1.py` .

Usage Example

```
obj = Simplest(10,20,30)
print(obj.a)      # should be 10
```

4 `class Rotate`

`Rotate` stores three different values, but rotates them, round-robin fashion, when you ask it to. It has the methods defined below.

All of the data fields must be private fields.

Define this class inside `classes_prob1.py` .

- `__init__(self, first,second,third)`

  Constructor. It should store the three values in private fields.

- `get_first(self)`
  `get_second(self)`
  `get_third(self)`

  Getters, for the three values, as they currently stand. This class **does not have any setter methods.**

- `rotate(self)`

  Rotates the first, second, and third values The first should move to the end (the third position), and the second and third should move up.

Usage Example

```
obj = Rotate("foo", "bar", "baz")
print(obj.get_first())      # should be "foo"
obj.rotate()
print(obj.get_first())      # should be "bar"
```

5 class Band

`Band` represents a musical group. It can contain one singer, one drummer, and any number of guitar players. When the object is created, it will only contain a singer; use getters and setters to update the various band members. (It has the members defined below.)

All of the data fields must be private fields.

Define this class inside `classes_prob1.py` .

- `__init__(self, singer)`

  Constructor. It should store the singer it is passed, set the drummer to None, and record that the band does not (yet) have any guitar players.

- `get_singer(self)`
  `set_singer(self, new_singer)`
  `get_drummer(self)`
  `set_drummer(self, new_drummer)`

  These are the getters and setters for the singer and the drummer.

- `add_guitar_player(self, new_guitar_player)`
  `fire_all_guitar_players(self)`

  The `Band` can have any number of guitar players: from 0 to as many as you like. The class begins with 0, when it is created; to add a player, you can call `add_guitar_player()` . You cannot change players once they are added, or remove them one at a time; the only way to get rid of the guitar players is to fire **all** of them, all at once.

- `get_guitar_players(self)`

  When you call `get_guitar_players()`, the `Band` class must return a list of guitar players - and the players must be listed in the order that they were added to the band. However, the list **must not** be a reference to some internal list that you are using to store the guitar players - you must **duplicate** that information into a new list.

  Why would you do this? Because if you didn't, the caller can modify the list without asking you - which changes what the `Band` thinks is its list of guitar players.

  Should you do this in the Real World? Sometimes! In some programs, isolating the two code components is critical - so you would duplicate the array, for safety. But in other programs, performance might be more important - so you might return the array (and take the risk) because it's faster (and easier).

- `play_music(self)`

  The band plays some music, based on the members of the band. We represent this by printing various lines of output.

  First, the singer sings. The class knows of two special singers, Frank Sinatra and Kurt Cobain. All other singers sound exactly the same. If the singer is `''Frank Sinatra''` (`https://www.youtube.com/watch?v=957t48-rU9E`), then print the following:

  ```
  Do be do be do
  ```

  If the singer is `''Kurt Cobain''` (`https://www.youtube.com/watch?v=FklUAoZ6KxY`), then print the following:

  ```
  bargle nawdle zouss
  ```

  Otherwise, print the following:

  ```
  La la la
  ```

  Next, the drummer plays the drums. If the drummer is `None`, then print nothing; however, if the drummer is not `None`, then print the following:

  ```
  Bang bang bang!
  ```

  Finally, the guitars play. For each guitar player in the band, print the following

  ```
  Strum!
  ```

  (Yes, you may print several lines of the same thing.)

Usage Example

```
obj = Band("Elvis Presley")
obj.set_drummer("Chad Smith")
obj.play_music()
```

6 class Color

The class `Color` represents an RGB (`https://en.wikipedia.org/wiki/RGB_color_model`) color. It stores the red, green, and blue components of the color as integers; each one is in the range [0,255] (inclusive). It has the methods defined below.

All of the data fields must be private fields.

Define this class inside `classes_prob2.py` .

- `__init__(self, r,g,b)`

  Constructor. This class should accept values that are too large or too small; but should bound them; that is, any value less than 0 should be set to 0, and any number larger than 255 should be set to 255.

  (Since you're doing exactly the same thing three times, you are **required** to use a function here, to contain the common code.)

- `__str__(self)`

  Instead of using getters for the individual colors, this class has three methods that allow you to view the color in different ways.

  The `__str__()` method, (which, of course, is the method that Python will call if you call `str(object)`) returns a string that has red, green, and blue components (in that order); the complete string looks like this:

  ```
  rgb(10,20,30)
  ```

  (Don't include any whitespace, including any newline character.)

  Did you realize that Python's f-strings (which we've used for formatted output) work to build a string for any purpose - not just for printing?

- `html_hex_color(self)`

  `html_hex_color()` returns the same information as `__str__()` , but encoded the way that colors are often specified on web pages (`https://www.w3schools.com/colors/colors_hex.asp`). These start with a hash character (#) and then have 6 hexdecimal characters (0 through 9, A through F). The first two represent the red component of the color, the next two represent the green component, and the last two represent the blue component. For example, if the color is (0,255,64), the proper return value would be:

```
#00FF40
```

But you don't need to know how hexdecimal works! (Until you take 252.) If you are using f-strings, you can add a "format specifier" to show Python how you want the variable be printed out. To format an integer `foo` as exactly two uppercase hex characters, use format string

```
f"foo={foo:02X}"
```

- `get_rgb(self)`

  Finally, the function `get_rgb()` returns the red, green, blue components as a tuple.

- `set_standard_color(self, name)`

  This class does not let you change the colors individually. Instead, you can set the color to one of four standard colors. The new color must be given as a string. Perform a **case-insensitive** check; if it is red, yellow, white, or black, set the r,g,b components of the object to the proper values. (You can find the proper RGB values for each color at the link above.)

- `remove_red(self)`

  Set the red component of the current color to zero. Leave the green and blue components unchanged.

Usage Example

```
obj = Color(0,500,0)           # will get bounded to (0,255,0)
x = str(obj)                   # returns "rgb(0,255,0)"
obj.set_standard_color("WHITE")
```

7 class Room

The class `Room` represents a room in some underground maze; it might be used, perhaps as part of a MUD (`https://en.wikipedia.org/wiki/MUD`). Each room has four exits, which often (not always) lead to other rooms. The exits are `n,s,w,e` - representing the four points of the compass.

In a real `Room` object that was part of a game, there would be many different properties (like a description, a list of objects on the floor, etc.). But for our little class, you will only need to support two types of fields: a name (which must be a string) and the four exits.

The name must be a private field; thus, you must store it in a variable whose name starts with a single underscore. You must provide a getter and setter for the name.

The four exits, however, should be public fields; they should simply have the names `n,s,w,e` . Each exit is either a reference to another `Room` object, or else `None` (meaning that the wall has no exit in that direction).

All exits must be symmetric. That is, if you have an East exit that points to another `Room`, then that `Room` must have a West exit which comes back to the current room.

Next, you will write one non-trivial method: `collapse_room()`, which represents a cave-in. In this method, which is called on a certain `Room`, you must set all of the exits of that room to `None`. In addition, in each adjacent `Room`, you must set the exit coming **back** to `None` as well - so that it's impossible to reach this room anymore, from anywhere else.

Define this class inside `classes_prob3.py` .

7.1 `Room`: Required Methods

- `__init__(self, ????)`

  Your class must include a constructor. However, my testcases will never create any `Room` objects directly (only through calling `build_grid()`), so you have total freedom about what you want the parameter(s) to be.

- `get_name(self)`
  `set_name(self, name)`

  Getter and setter for the room name. The name must always be a string.

- `collapse_room(self)`

  Breaks all exits to and from this room. All of the exits leaving this room are set to `None`; in addition, in each adjacent room, the link to this room is set to `None`.

7.2 Factory Function: `build_grid()`

In addition to the class, your file `classes_prob3.py` must also define a function (not a method of any class) named `build_grid(wid,hei)`. This will build a two-dimensional grid of `Room` objects, where the width (that is, the West-East size) and the height (that is, the North-South size) are as given. You may assume (without checking) that the height and width are both ≥ 1.

You must allocate all of the necessary `Room` objects, and link them together with their exits. `Rooms` on the edge of the grid will have one exit which is `None` (those in the corner will of course have two `None`s).

You must return the `Room` object representing the Southwest corner. **Do not return an array or other data structure;** you must return a reference to a single object.

While I don't care what names you give to the various rooms, you must give the rooms different names. You can come up with any scheme that you want (given them grid coordinate names), randomly select names from a pool, etc. However, you must (somehow) guarantee that (a) every name is a string

of nonzero length; and that (b) no two `Room`s in the grid you return have the same name.

8 Turning in Your Solution

You must turn in your code using GradeScope.