Practice Exercises for Classes (part 2)

Solve each of the practice exercises below. Each problem includes three CodeSkulptor links: one for a template that you should use as a starting point for your solution, one to our solution to the exercise, and one to a tool that automatically checks your solution.

1. Before proceeding to the task of implementing Memory using the `Tile` class that we developed in Week 6a, your job is to complete couple of pairs of exercises in which we implement and then use some simple classes. As your first task, implement a `Person` class which has the fields `first_name`, `last_name` and `birth_year`. This class should include the methods: `__init__` which takes strings for the two name fields and an integer for the year of birth, `full_name` returns the full name for a person as a string, which is the first name followed by a space, followed by the last name, `age` which takes the current year as input and returns the age in years of the person (Don't worry about days and months here, just return the difference of the two years.), and `__str__` returns a string that includes the first name and last name of the person as well as their year of birth. Definition of `Person` class template --- Definition of `Person` class solution --- Definition of `Person` class (Checker)
2. Write a function `average_age` that takes a list of `Person` objects along with the current year and returns the average age of the people in the list. Remember that `average_age` should only use the methods defined in the `Person` class. (The body of `average_age` should not access the fields in a `Person` object directly.)Application of `Person` class template --- Application of `Person` class solution --- Application of `Person` class (Checker)
3. Implement a `Student` class which has the fields `person` (Person object), `password` (string), and `projects`(list of strings). (Note that class uses another class, just as in Blackjack.) This class should include the following methods:, `__init__` which takes a person (specified as Person object) and a password (specified as a string) and creates a `Student` object (the list of projects should be empty to start), `get_name` which returns the student's full name, `check_password` which take a supplied password and returns a Boolean indicating whether the supplied password matches the student's created password, `get_projects` which returns the list of the student's projects and, `add_project(project_name)` which adds the specified project to the student's list of projects. Note that this last method does not check whether

the project already exists in the list. Definition of `Student` class template --- Definition of `Student` class solution --- Definition of `Student` class (Checker)

4. Write a function `assign` that takes a list of `Student` objects, a student full name, a password, and a project as parameters. This function should search the list of students for students whose name and password match the supplied information. When a match is found, the function checks the student's current list of projects for the supplied project. If the project does not already exist in the list, the function adds the project to the list. Remember to use only methods for the `Student` class to manipulate `Student` objects. Application of `Student` class template --- Application of `Student` class solution --- Application of `Student` class (Checker)
5. We now return to Memory. For this problem, your task is to initialize a game of Memory using the `Tile` class. Starting from the provided template, complete the implementation of the function `new_game()` that initializes our version of Memory. In particular, create a list with two copies of the numbers in `range(0, DISTINCT_TILES)` and use `random.shuffle` to shuffle the list. Then, use the initializer for the `Tile` class to create a horizontal row of `2 * DISTINCT_TILES` tiles whose numbers are hidden. Finally, implement a draw handler using the `draw` method for the `Tile` class that draw all 16 tiles on the canvas. Initial version of Memory template --- Initial version of Memory solution --- Initial version of Memory (Checker)
6. Your next task is to build a simple version of Memory that exposes tiles in response to mouse clicks. Using the provided template, add code to the mouse handler `mouseclick()` that exposes a tile in response to a mouse click on that tile. This code should interact with a tile using only `Tile` class methods. Clickable version of Memory template --- Clickable version of Memory solution --- Clickable version of Memory (Checker)
7. **Challenge**: To finish our object-oriented implementation of Memory, complete the implementation of the function `mouseclick()` in the provided template based on the state logic described in the Memory video. Again, your code should interact with the tiles only via `Tile` class methods. When done correctly, the code should express the logic of this function in surprisingly readable manner. Final version of Memory template ---Final version of Memory solution --- Final version of Memory (Checker)
8. **Challenge**: Your object-oriented implementation of Memory is remarkably flexible. Experiment with building an extended version that involves a 2D layout of the tiles. If you are particularly ambitious, you can also add an `update_position()` method to the `Tile` class that modifies the position of a tile when it is called. If you add a loop in your draw handler that calls

`update_position()` on each tile, you can easily build a variant of Memory in which the tiles move dynamically.