Practice Exercises for Classes (part 1)

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. For this set of practice exercises, we will walk you through the creation of a `Tile` class suitable for use in your week five mini-project, Memory. This class will model the logical behavior of cards/tiles used in Memory. Our goal in walking through this design will be to understand the syntactic structure of a Python class in detail as well as the logic that goes into designing a useful class. In Week 6b, we will use this `Tile` class to re-implement Memory in an object-oriented style. If, at some point, you feel confused, you may also want to watch this video on the Basics of OOP by Julie. Many students in previous sessions found it to be helpful.

To begin, your first task is to define a `Tile` class using two lines of Python. For now, the body of this class should just be a single `pass` statement to keep Python from throwing an error. As usual in Python, the body of the class definition should be indented. Definition of `Tile` class template --- Definition of `Tile` class solution --- Definition of `Tile` class (Checker)

2. Your next task is to create two instances (versions) of a `Tile` object. Note this is possible even though the body of the class definition is currently empty. Of course, the objects that you create at this point will contain no data. We'll add some data to `Tile` objects in the next problem. Remember that to create an instance of a class `ClassName` , you can use the expression `ClassName()`. Create a `Tile` object template --- Create a `Tile` object solution --- Create a `Tile` object (Checker)

3. As we noted in the last problem, our current definition of the `Tile` class produces objects that contain no data. This design is not going to be much help in re-implementing Memory. At this point, we should start considering what kind of data a `Tile` object should contain. One piece of data that `Tile` objects should certainly contain is the number associated with the tile. To create a `Tile` object that contains the number associated with the tile, we need to define a special function known as an *initializer* in the body of the class definition. (Functions defined in the body of the class definition are referred to as *methods*.)

In Python, the class initializer always has the special name `__init__`. The parameters to this function `__init__` provide the information necessary to create the data stored in the object. By convention, the first parameter to `__init__` has the name `self` and serves as a reference to the object generated by the initializer. The remaining parameters, if any, contain the information used in creating the object. The body of the initializer consists of sequence of Python statements that use this information to compute and add data to the object. Each piece of data stored in the object is named by a *field*. If name of a field is `field_name`, we can reference this piece of data via the expression `self.field_name`.

Your next task is to implement an initializer for the `Tile` class and create two tiles; one called `my_tile` whose `number` field has the value `3` and another called `your_tile` whose number field has value `4`. The definition of the initializer should include the required parameter `self` and a parameter `num` that is the number associated with the tile. The body of the initializer should store `num` in the field `number`. To create a `Tile` object, you will need to include the number associated with the tile as parameter when you call `Tile(....)` to create a `Tile` object. (The reference corresponding to the parameter `self` is generated by Python during the creation of the `Tile`object and is not included in the call to the initializer.)

Initializer for `Tile` class template --- Initializer for `Tile` class solution --- Initializer for `Tile` class (Checker)

4. In the previous problem, we accessed the one piece of data in the `Tile` object `my_tile` via the expression `my_tile.number` in the provided testing code. More generally, given an object `my_object`, Python can access the contents of the field `field_name` via the expression `my_object.field_name`. In the Python community, accessing the content of an object using its field names is common practice. In this class, we will follow the practice of accessing the contents of objects using methods known as *getters* and *setters*. While not required by Python, this practice encourages the user of the class to manipulates class objects solely via class methods. The advantage of following this practice is that the implementer of the class definition (often someone other than the user of the class) may restructure the organization of the data fields associated with the object while avoiding the need to rewrite code that uses the class.

For this problem, your task is to implement a method `get_number` that returns the number associated with a tile. As usual, the first parameter for this method will be `self`. Then, call this method on the `Tile` object `my_tile`and assign this value to the variable `tile_number`. Following the convention of other object-oriented programming languages, Python's syntax for calling methods on class objects is `object_name.method_name(....)`. In evaluating this call, `object_name` is bound to the first parameter `self` in the definition of `method_name`. Note that we have already seen this syntax when calling list methods in Python. For example, the statement `my_list.append(5)` appends the number `5` to the end of the list `my_list`. Example method for the `Tile` class template ---- Example method for the `Tile` class solution --- Example method for the `Tile` class (Checker)

5. At this point, our `Tile` class is still not so useful. Your next task is to add a field called `exposed` to a `Tile` class definition and implement three methods that manipulate this field. Logically, this field will be `True` when the tile's number is exposed to the player and `False` when the tile's number is hidden from the player. To add this field to a `Tile` object, you will need to add another parameter `exp` to the `__init__` method and initialize the `exposed`field with this value. Once you have done this, implement three methods described below that manipulate this field. You will use all of these methods when we re-implement Memory.

- `is_exposed` which takes a tile and returns the value of the `exposed` field,
- `expose_tile` which takes a tile and sets the value of its `exposed` field to be `True`, and
- `hide_tile` which take a tile and set the value of its `exposed` field to be `False`.

Note how this design of the `Tile` class logically binds together the data associated with a tile into a single object. In our previous implementation of Memory, the data associated with a single tile was stored in two separate lists. This design required us to keep track of where the data for the same object was in both lists. For Memory, this task wasn't so hard. However, as the complexity of your programs increase, grouping all of the data corresponding to a single logical entity together is a powerful way to reduce the complexity of your program. Methods for exposing and hiding a `Tile` object template --- Methods for exposing and hiding a `Tile` object solution --- Methods for exposing and hiding a `Tile` object (Checker)

6. In the previous problem, we used the method `is_exposed` to determine whether a tile's number was exposed. Sometimes, especially during debugging, it is useful to know the value of all of the object data fields. As the first few exercises showed, trying to print a `Tile` object in Python yields a message of the form `"<__main__.Tile object>"` This message indicates the object is a tile, but doesn't provide any helpful information about the data stored in the object. In Python, we can define a special string method named `__str__`that Python will automatically call when printing an object or converting the object into a string. Based on the values of the object's fields, the body of the `__str__` method should construct and return a string that indicates the state of the object.

For this problem, implement a string method for the `Tile` class that return a string of the form `"Number is 3, exposed is True"`. Remember that you will need to use the `str` operation to convert the data in an object into a string. __str__ method for `Tile` class template --- __str__ method for `Tile` class solution ---__str__ method for `Tile` class (Checker)

7. Challenge: At this point, our `Tile` class has captured much of the behavior of tiles in Memory. However, the problem of how to draw tiles still remains. One possibility would be to maintain a list of `Tile` objects and draw them using a for-loop where the location of the tile varies as function of the loop's iteration as done in our suggestion implementation of Memory for week five. This choice, while not terrible, implicitly models the location of a tile by its position in a list. Instead, we suggest that the we store the location of the tile as part of the data associated with a `Tile` object. This choice groups all of the relevant properties of a tile into one object. Following the convention of `draw_text`, the tile's location will be specified by the lower left corner of the tile with the width and height of the tile being specified by the global constants `TILE_WIDTH` and `TILE_HEIGHT`, respectively.

With this design, we can implement a `draw_tile` method for the `Tile` class that draws the specified tile object at its corresponding location. This `draw_tile` method will use SimpleGUI's `draw_text` or `draw_polygon`methods to draw the tile. Since both of these methods require the canvas which itself is passed to the draw handler, we must call the `draw_tile` method inside the draw handler and pass the current canvas as a parameter to `draw_tile`. This design moves most of the complexity of drawing a tile out of the draw handler into the `Tile` class and makes the logic of the draw handler much more transparent.

For this problem, your task is to add the location of the tile to the `Tile` class and implement a `draw_tile`method for the Tile class. The tile should display the tile's number as text at the tile's location if the tile's number is exposed. Otherwise, it should draw the unexposed tile at the its location as a green polygon. If your implementation of the `draw_tile` method is correct, our implementation of the draw handler should display a pair of tiles on the canvas, one whose number is exposed and one whose number is not. draw_tile method for the `Tile` class template --- draw_tile method for the `Tile` class solution --- draw_tile method for the `Tile` class (Checker)

8. Challenge: To complete our implementation of the `Tile` class, we must determine whether a tile has been clicked. In our week five implementation of Memory, this computation was done in the draw handler. For our object-oriented version, we will implement a method `is_selected` for the `Tile` class that returns `True` if a tile contains a specified point. To determine if a tile has been clicked, the mouse handler will then simply call `is_selected` using the position of the mouse. Again, this design moves the determination of whether a point lies on a tile out of the mouse handler into the `Tile` class, reducing the complexity of the mouse handler.

For the final problem, your task is to implement the `is_selected` method. This method take a point `pos` as a parameter and return `True` if the point lies inside the tile. If your implementation of this method is correct, our provided template will allow you to flip over two cards whose numbers are hidden. is_selected method for `Tile`class template --- is_selected method for `Tile` class solution --- is_selected method for `Tile` class (Checker)