Practice Activity: Testing Solitaire Mancala

One emphasis in the class will be testing your code as you write it. In some cases, this testing code will be provided by us as a `poc_simpletest` test suite or an Owltest machine test. For the first few mini-projects, you are also welcome to use test suites created by your peers in testing your code. However, you will eventually need to become proficient in writing your own test suites.

In this tutorial, we will walk you through the process of implementing a test suite for Solitaire Mancala as you implement the first few methods of the `SolitaireMancala` class. Before we begin, we suggest that you review this class page that explains the basics of creating a test suite using `poc_simpletest`.

Read and think about the mini-project requirements

To begin your development process, you should start with the Solitaire Mancala practice mini-project description. Prior to any coding attempts, you should read the description and think about how you might implement the mini-project. One crucial mistake that many beginner/intermediate programmers make is trying to write code too soon. Take your time and make sure that you thoroughly understand the project requirements. Ask questions in the forums if you don't understand some aspect of the requirements.

Sketch the outlines of the class definition

If you are not provided with a program template, your next goal should be to sketch an outline of your program. This outline should include a docstring at the top of the program that describes the behavior of the program as well as any necessary class definitions with their various method implemented using a `pass` statement. The class definitions and their methods should include docstrings that describe their behavior.

For Solitaire Mancala, we did not provide you with a program template. However, the mini-project description includes a detailed description of the required `SolitaireMancala` class. Working from that description, you should have produced a sketch for Solitaire Mancala. Once you have attempted this task, feel free to compare your attempt to our example program.

At this point, you should also begin the process of creating a test suite for your implementation. This version of the test suite represents a simple starting point for your program. Note that the template imports the `poc_simpletest` module and defines a single function `run_suite(GameClass)` which takes the `SolitaireMancala` class as input. Inside `run_suite`, we have created an instance of a `TestSuite()` object from the `poc_simpletest` module and add a final call to the method `report_results` for this object to print a summary of the test results in the console.

Code and test your implementation of the class methods

At this point, you are ready to begin the process of implementing and testing your code. The first two methods that you should implement are the `__init__` method and the `__str__` method. You should begin with the `__init__` method since it initializes a `SolitaireMancala` object and the `__str__` method since it provides the ability to examine a `SolitaireMancala` object.

The mini-project description states `__init__` initializes a `SolitaireMancala` object whose initial board is `[0]`. Although it is extremely tempting to implement the `__init__` method immediately, you should instead add a test to the your test suite that tests whether your impending implementations of `__init__` and `__str__` work correctly. **Creating test data for your methods before you implement them is one of the hallmarks of a good programmer.** The process of coming up with good test data before implementing forces you to think more deeply about what each method should do. In the long run, the extra time spent on building test examples will save you substantially more time in debugging.

In this version of the test suite, we have a added a single test that checks whether a new board has the initial configuration `[0]`. At this point, we implement both methods and are ready to run our test code. To do this, we add two statements that import the our testing code and call `run_suite` with a `SolitaireMancala` object of the form:

```
1  import poc_mancala_testsuite_v1 as poc_mancala_testsuite
2  poc_mancala_testsuite.run_suite(SolitaireMancala)
3
```

Note that when you are creating and using a test suite, the name `poc_mancala_testsuite_v1` can be replaced with the name of your saved test suite of the form `userXX_XXXXXX_X`. The second statement passes the `SolitaireMancala` class to your imported test code and runs all of the specified tests on instances of `SolitaireMancala` objects.

Running our example program yields "`Ran 1 tests. 0 failures.`" Our code has passed its first test!

Find errors in our implementation

At this point, we proceed to the methods `set_board` and `get_number_seeds`. Again, our first task is to add some tests to our test suite that check the expected results from these two methods. In this version of the test suite, we have added more calls to `run_suite` that set the board's configuration and then check the resulting configuration using the `__str__` method and the `get_num_seeds` methods. Having built these tests, we then implemented the two methods. Running this example program yields the message
"`Test #1a: str Computed: [0, 0, 1, 1, 3, 5] Expected: [5, 3, 1, 1, 0, 0]`".

This failure of one of our test cases alerts us that something is amiss. The `__str__` displays the board configuration with the store on the left. The expected result from the test data correctly indicates that the store should be displayed on the right. This example program corrects the error and passes all tests.

At this point, the iterative process of coding and testing should start to become apparent to you. While building test examples prior to implementing a method may seem laborious, slow and steady wins the race when coding at the limits of your abilities.

✓ Terminer