**SI 507 Fall 2019**

**Homework 2:** Unit Testing

## Homework Objective:

Demonstrate the ability to:

* Write effective unit tests
* Use tests to verify that new code works as specified

## Deliverables and Submission Process:

For the main assignment, submit a modified version of the file cards.py via Canvas. The code must be executable! Code that does not run in Python3 will be given a score of 0. You can receive partial credit for working programs.

For the Extra Credit problems, submission details are included with the instructions.

## Supporting Material:

Files: cards.py

## Background:

In order to complete this assignment you will need to familiarize yourself with the Card class covered in lecture. You will also want to review any readings on Unit Testing in Python.

Then you will include tests for the cases described below. There are a few notes though:

* You may create as many or few unittest.TestCase subclasses as you like.
* You must arrange your code so that if you were to fail a test, that would not cause any other tests to fail as a result of that test failing.
* IF YOU INVOKE THE play\_war\_game FUNCTION IN A TEST CASE, YOU MUST INVOKE IT WITH THE PARAMETER testing=True, like this:

play\_war\_game(testing=True) AS OPPOSED TO play\_war\_game()

If you do not do that, we will not be able to grade your homework properly!

* You may assume that other programmers will NOT invoke these functions with unacceptable inputs (e.g. no one will try to create a card with rank 0). You just need to ensure that the code works as intended.
* If you find any errors through your testing, DO NOT fix the code.

### Part 1 Basic Unit Testing (20 points)

1. Test that if you create a card with rank 12, its rank will be "Queen"
2. Test that if you create a card instance with suit 1, it will be suit "Clubs"
3. Test that if you invoke the \_\_str\_\_ method of a card instance that is created with suit=3, rank=13, it returns the string "King of Spades"
4. Test that if you create a deck instance, it will have 52 cards in its cards instance variable
5. Test that if you invoke the pop\_card method on a deck, it will return a card instance.
6. Test that if you invoke the pop\_card method on a deck, the deck has one fewer cards in it afterwards.
7. Test that if you invoke the replace\_card method, the deck has one more card in it afterwards. (Please note that you want to use pop\_card function first to remove a card from the deck and then replace the same card back in)
8. Test that if you invoke the replace\_card method with a card that is already in the deck, the deck size is not affected.(The function must silently ignore it if you try to add a card that’s already in the deck)
9. Test that the return value of the play\_war\_game function is a tuple with three elements, the first of which is a string. (This will probably require multiple test methods!)
10. Write 1 additional test. Make sure to include a descriptive message in it so we can easily see what you are testing!

### Extra Credit 1: Writing your own class/tests (2 points)

In this part, you will implement the class Hand and create tests to verify that it works as specified. You will need to create a new testing class called TestHand that subclasses unittest.TestCase, and implement 3 test functions.

class Hand:

# create the Hand with an initial set of cards

# param: a list of cards

def \_\_init\_\_(self, init\_cards):

pass

# add a card to the hand

# silently fails if the card is already in the hand

# param: the card to add

# returns: nothing

def add\_card(self, card):

pass

# remove a card from the hand

# param: the card to remove

# returns: the card, or None if the card was not in the Hand

def remove\_card(self, card):

pass

# draw a card from a deck and add it to the hand

# side effect: the deck will be depleted by one card

# param: the deck from which to draw

# returns: nothing

def draw(self, deck):

pass

These tests are less specified than those in part 1, so you will have to think about writing good tests for each of these functions.

You do not need to worry about testing for invalid inputs. For example, you can assume that Hand will be initialized with a valid list of valid Cards, and that draw( ) will be called with a valid Deck.

1. Test that a hand is initialized properly.
2. Test that add\_card( ) and remove\_card( ) behave as specified (you can write one test for this, called testAddAndRemove.
3. Test that draw( ) works as specified. Be sure to test side effects as well.

***What to turn in:***

Create a *new* file called cards2.py and turn it in. This needs to be submitted separately from the cards.py you turn in for Parts 1, even though it will be based on it.

### Extra Credit 2 (2 points):

Implement two new pieces of functionality:

1. Add a function “remove\_pairs” to Hand that looks for pairs of cards in a hand and removes them. Note that if there are three of a kind, only two should be removed (it doesn’t matter which two). Write tests to verify that this works.
2. Add a function “deal” to Deck that takes two parameters representing the number of hands and the number of cards per hand and returns a list of Hands. If the number of cards per hand is set to -1, *all* of the cards should be dealt, even if this results in an uneven number of cards per hand. Write tests to verify that this works.

***What to turn in:***

Create a *new* file called cards3.py and turn it in. This needs to be submitted separately from the cards.py you turn in for Parts 1, even though it will be based on it.