Objectives

* Introduction to Object Oriented Program (OOP)
* Practice Test-Driven Development (TDD)

Due Date

This assignment is due on Tuesday, 30 January, 2024, by 11:00 pm. Refer to the syllabus on how late day tokens are to be used.

**Remarks:**

* It is expected you develop test cases for your code.
* When you are asked to hand in code, you cut-and-paste the definition(s) from your code and paste into this Word document immediately following the activity.
* Include with your code several test case examples and your results.
* This homework is to be individual work.

Value

This assignment is worth 20 points.

Activities

# OOP with Animals

1. Create classes to represent the following hierarchy using inheritance in a module called Animals.py. Note the follow-on instructions on characteristics of each class.

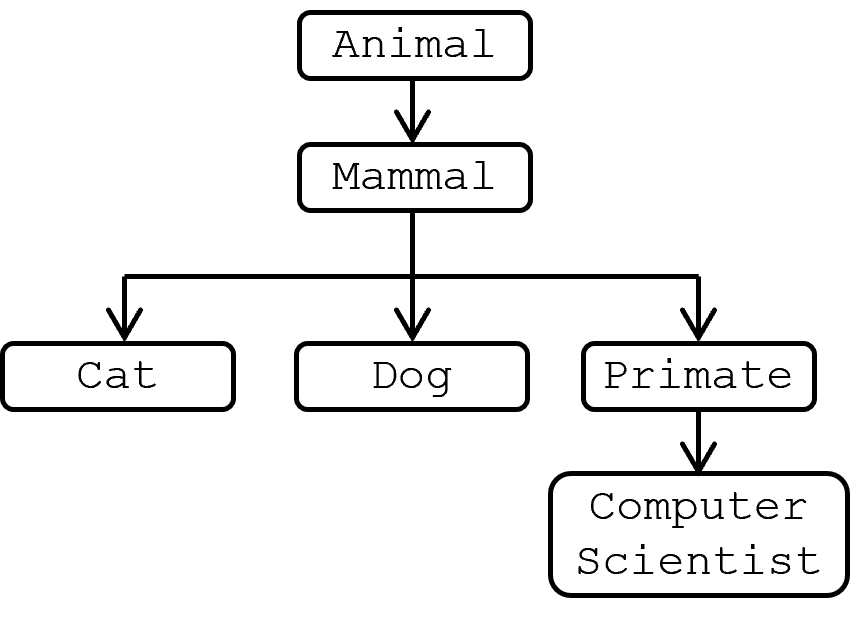


Figure 1: Class Hierarchy to Implement

1. Every class should be initialized with an attribue name. Because all objects share this behavior, it should be defined in the superclass constructor (Which is the “superclass”? What is a “constructor”?). It should have the default value of “It” (This is not recommended, you should give your pets a name. I named my first dog “Stay” ;).
2. Every class except ComputerScientist should define its own speak() method, which returns a string giving the name and an appropriate sound for that class. For example:

c1 = Cat('Babs')

c1.speak()

>>> 'Babs says Meow!'

1. Do not implement speak in the ComputerScientist subclass - it should default to whatever method you define for Primate.
2. The Animal superclass should define a method called reply(), which just calls the relevant speak:

c1 = Cat('Babs')

c1.speak() # should call Cat.speak()

c1.reply() # should call Animal.reply()

>>> 'Babs says Meow!'

>>> 'Babs says Meow!'

1. Provide a listing of your code for Animals.py below this bullet. Include the test code for your program (following the TDD approach as described in the text).

class Animal:  
 *"""Represents a non-specific animal"""* def \_\_init\_\_(self, name="It"):  
 *"""Instantiates a new animal with a name"""* self.name = name  
  
 def speak(self):  
 *"""Allows for an error to be caught if speak is not defined elsewhere"""* raise NotImplementedError("Subclasses must implement this method")  
  
 def reply(self):  
 *"""Calls the speak method"""* return self.speak()  
  
class Mammal(Animal):  
 *"""Represents a Mammal in animal kingdom"""* def speak(self):  
 return f"{self.name} says HuH!"  
  
  
class Primate(Mammal):  
 *"""Represents subclass of Mammals that are Primates"""* def speak(self):  
 return f"{self.name} says EE-EE-OOH-OOH"  
  
class ComputerScientist(Primate):  
 *""""Represents a subclass of Primates that are Computer Scientists"""* # speak is inherited from primate  
 pass  
  
class Cat(Mammal):  
 *"""Represents a subclass of Mammals that are Cats."""* def speak(self):  
 return f"{self.name} says Meow!"  
   
# Test Cat Initialization and speak method  
cat = Cat("Wilbur")  
assert cat.name == "Wilbur", "Cat name initialization failed"  
assert cat.speak() == "Wilbur says Meow!", "Cat speak method failed with the name Wilbur"  
  
# Test Animal reply method   
assert cat.reply() == "Wilbur says Meow!", "Animal reply method failed using Cat instance with the name Wilbur"  
  
# Testing Mammal class  
mammal = Mammal("Brittney")  
assert mammal.name == "Brittney", "Mammal name initialization failed"  
assert mammal.speak() == "Brittney says HuH!", "Mammal speak method failed"  
  
# Testing Primate class  
primate = Primate("George")  
assert primate.name == "George", "Primate name initialization failed"  
assert primate.speak() == "George says EE-EE-OOH-OOH", "Primate speak method failed"  
  
# Testing ComputerScientist class  
cs = ComputerScientist("Alice")  
assert cs.name == "Alice", "ComputerScientist name initialization failed"  
assert cs.speak() == "Alice says EE-EE-OOH-OOH", "ComputerScientist speak method failed"

# OOP & TDD with Cards

This is where the bulk of your work will be for this assignment. Anticipate spending at least a few hours here if you are new to OOP and TDD, and consider breaking it up over a few days if your schedule allows.

Card games are a classic application of OOP. They let us use composition (decks of cards contain several card objects) and inheritance (a hand of cards can be treated as a specialized deck). The diagram below shows the inheritance model and the specific instance variables and methods we’ll implement here. Descriptions of attributes and methods are provided later in this document.

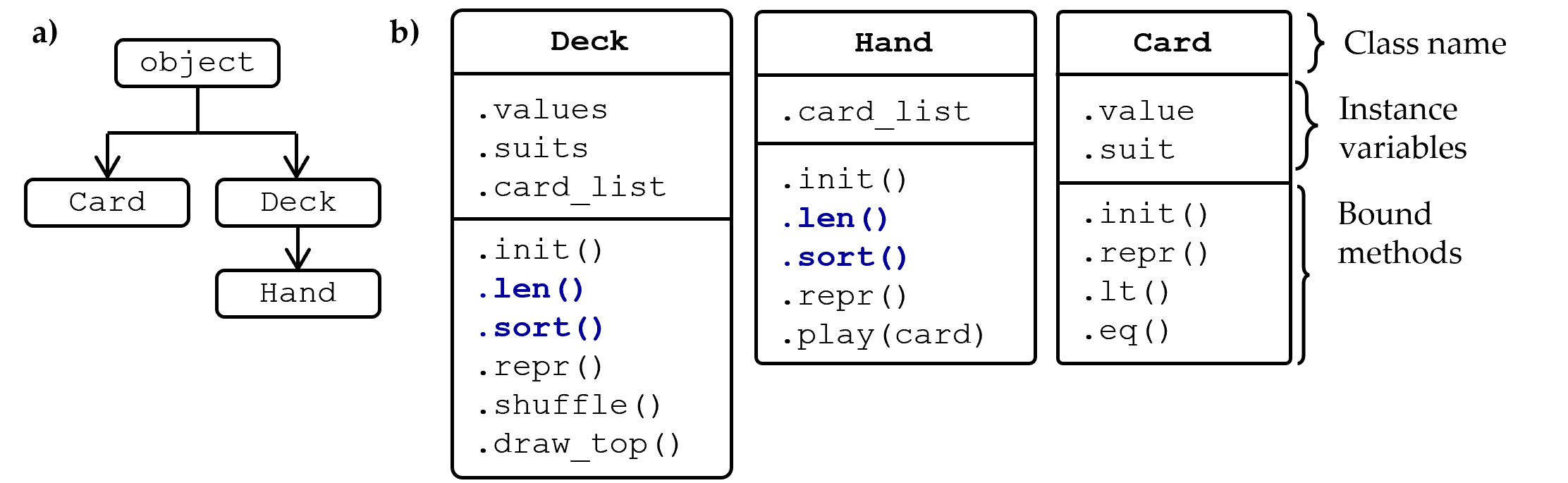
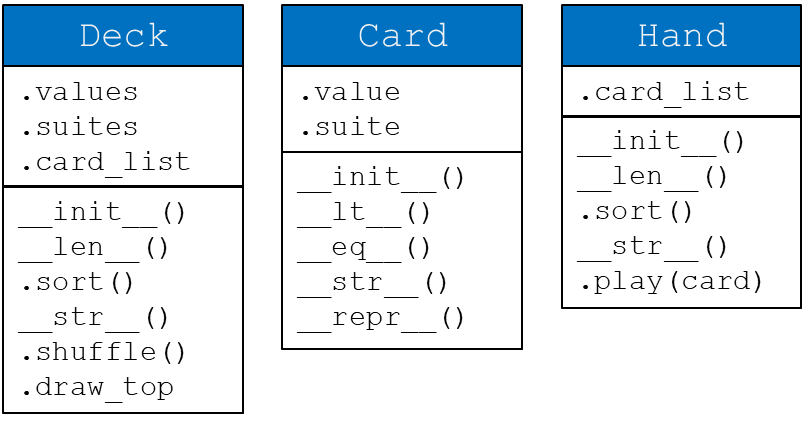
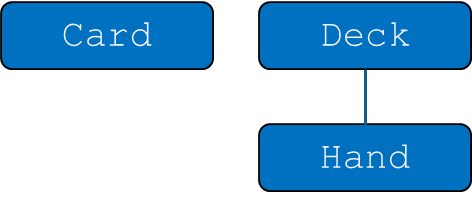


Figure 2: (a) Class diagram and (b) attributes for each class. Inherited attributes are in blue.

Create the classes above in a file called Cards.py. Using the TDD approach, write unittests BEFORE you write your other code and place your unittests in a file called TestCards.py. This includes any exceptions you should raise. See the unittest basic example below for an illustration of how to test that an error is raised and read more about unittest ([link)](https://docs.python.org/3/library/unittest.html#basic-example). Your final TestCards.py class should include 3 classes - one for each class you are trying to test:

|  |
| --- |
| from Cards import Card, Deck, Hand  import unittest  **class** TestCard(unittest.TestCase):  *"Test cases specific to the Card class"* **def** test\_init(self):  *"Add a docstring here"*  *# other tests* |
| **class** TestDeck(unittest.TestCase):  *# your tests here* **class**  **class** TestHand(unittest.TestCase):  **if** \_\_name\_\_ == '\_\_main\_\_':  unittest.main() |

Structure your code based on OOP principles. Hand should inherit from Deck, and should not overload any methods unless it needs to.

Every method (including your unittests) should have a docstring.

When writing tests, create your own examples. Do not use any of the examples shown below.

Some OOP relationships:

* Composition - Deck comprises a list of Card objects
* Inheritance - Hand is inherited from Deck so .len() and .sort() do not need to be recoded.

The expected input/output of each method is defined below. Most do what you would think intuitively. Before you start implementing code, review these guidelines:

## class Card

* \_\_init\_\_ - initialize a new card based on the specified parameters:
  + .value - the value of a card (i.e. the 3 in 3 of hearts)
  + .suit - the suit of a card (i.e. the hearts in 3 of hearts)
* \_\_repr\_\_ - return a “code-perspective” string of the object “Card(3, ‘hearts’)”. This is *very* similar to the \_\_str\_\_ magic method. Read this article to get insight into the different uses of each (i.e., they both “do” the same thing. But, convention is that a class definition “uses” them differently).
* \_\_str\_\_ - NOTE: This is not visible in the architecture picture. Return a “user-perspective” view of the card object “3 of hearts”.
* \_\_lt\_\_ (that is a small “el” in front of the “t”) - Implement as a magic method (\_\_lt\_\_) so it can be called with the standard operator (<). Sort by suit first, then increasing value (suits are sorted alphabetically, so clubs < diamonds)
* \_\_eq\_\_ - Implement as a magic method so it can be called with the standard operator (=). Sort by suit first, then value (suits are sorted alphabetically, so clubs < diamonds)

## Examples

**>>>** c1 **=** Card(3, 'hearts')

**>>>** repr(c1)

*'Card(3, ‘hearts’)'*

**>>>** print(c1)

*'3 of hearts’'*

**>>>** c2 **=** card(3, 'spades')

**>>>** c1 **<** c2

True

**>>>** c3 **=** Card(4, 'hearts')

**>>>** c3 **<** c2

True

## class Deck

* \_\_init\_\_ - initialize a Deck based on the collection of values and suits passed - make one card for each value/suit combination.
  + .values - collection of values stored in deck. This should be a parameter with a default of the numbers 1 through 13
  + .suits - collection of suits stored in deck. This should be a parameter with a default of ('clubs', 'diamonds', 'hearts', 'spades')
  + .card\_list - list of Card objects, containing all cards in the deck
* \_\_len\_\_ - override the magic method for length and return the number of cards in the deck.
* sort - sorts the cards in the deck
* \_\_str\_\_ - returns a “user perspective” string representation of the deck.
* shuffle - randomize the order of the deck. Import the random module and use random.shuffle() method here (look up in your chosen reference what this method is.
* draw\_top - remove and return the top card of the deck.
  + Treat the last item in card\_list as the “top” of the deck.
  + Raise a RuntimeError with the string “Cannot draw from empty deck” if someone tries to draw from an empty deck

## Examples

|  |
| --- |
| **>>>** d1 **=** Deck() *# use default values and suits*  **>>>** len(d1)  52  **>>>** d2 **=** Deck([2, 1], ['triangles', 'dots'])  **>>>** print(d2)  *'Deck: [Card(1 of dots), Card(2 of dots), Card(1 of triangles), Card(2 of triangles)]'*  **>>>** d2.shuffle()  **>>>** print(d2)  *'Deck: [Card(2 of triangles), Card(1 of dots), Card(2 of dots), Card(1 of triangles)]'*  **>>>** d2.draw\_top()  1 of triangles  **>>>** d2.draw\_top()  2 of dots  **>>>** d2.draw\_top()  1 of dots  **>>>** d2.draw\_top()  2 of triangles  **>>>** d2.draw\_top()  Traceback (most recent call last): ...  *RuntimeError*: Cannot draw from empty deck |

## class Hand

* \_\_init\_\_ - create a Hand with a passed in list of cards
* \_\_str\_\_ - returns a string representation of the Hand. Note that the class definition for Hand inherits all of the methods of Deck which already has a \_\_str\_\_ method. This will override that \_\_str\_\_ method and provide a new \_\_str\_\_ magic method for a Hand.
* play(card)
  + removes and returns card from hand
  + raises a RuntimeError with the string “Attempted to play Card(1 of clubs) that is not in Hand: [Card(5 of clubs),...*<and then list the whole hand>*” if card is not in hand

## Examples

|  |
| --- |
| **>>>** h\_clubs **=** Hand([Card(value, 'clubs') **for** value **in** range(5, 0, **-**1)])  **>>>** print(h\_clubs)  *'Hand: [Card(5 of clubs), Card(4 of clubs), Card(3 of clubs), Card(2 of clubs), ...'*  **>>>** h\_clubs.sort() *# inherited from Deck*  **>>>** print(h\_clubs)  *'Hand: [Card(1 of clubs), Card(2 of clubs), Card(3 of clubs), Card(4 of clubs), ...'*  **>>>** len(h\_clubs) |
| 5  **>>>** h\_clubs.play(Card(1, 'clubs'))  `Card(1 of clubs)`  **>>>** h\_clubs.play(Card(1, 'clubs')) Traceback (most recent call last): ...  *RuntimeError*: Attempted to play Card(1 of clubs) that is not in Hand: [Card(5 of clubs),... |

# Submitting

At a minimum, submit the following files:

* This Word document with your all of your code pasted in order after these bullets. Each code file should start on a new page in this document (CTRL+ENTER). Have this file on the top of your submission (so that it opens by default). And then, in the same submission but under this Word document, provide your three code files:
* Cards.py
* TestCards.py

import random  
class Card:  
 def \_\_init\_\_(self, value, suit):  
 self.value = value  
 self.suit = suit  
  
 def \_\_repr\_\_(self):  
 *'''Return the machine representation of the card'''* return f"Card({self.value!r}, {self.suit!r}"  
  
 def \_\_str\_\_(self):  
 *'''Return string representation of the card'''* return f"{self.value} of {self.suit}"  
  
 def \_\_lt\_\_(self, other):  
 *'''Compare Cards first by suit then by value'''* if self.suit == other.suit:  
 return self.value < other.value  
 return self.suit < other.suit  
  
 def \_\_eq\_\_(self, other):  
 *'''Check if two cards are equal first by suit then by value'''* return self.value == other.value and self.suit == other.suit  
  
class Deck:  
 def \_\_init\_\_(self, values=range(1, 14), suits=('clubs', 'diamonds', 'hearts', 'spades')):  
 self.card\_list = []  
 for suit in suits:  
 for value in values:  
 self.card\_list.append(Card(value, suit))  
  
 def \_\_len\_\_(self):  
 *"""Return the number of cards in the deck."""* return len(self.card\_list)  
  
 def sort(self):  
 *"""Sort the cards in the deck."""* self.card\_list.sort()  
  
 def \_\_str\_\_(self):  
 *"""Return a string representation of the deck."""* return ', '.join(str(card) for card in self.card\_list)  
  
 def shuffle(self):  
 *"""Shuffle the cards in the deck."""* random.shuffle(self.card\_list)  
  
 def draw\_top(self):  
 *"""Remove and return the top card of the deck."""* if not self.card\_list:  
 raise RuntimeError("Cannot draw from an empty deck")  
 return self.card\_list.pop()  
  
class Hand(Deck):  
 def \_\_init\_\_(self, cards=None):  
 *"""Initialize a hand with an optional list of cards."""* if cards is not None:  
 self.card\_list = cards  
 else:  
 self.card\_list = []  
  
 def \_\_str\_\_(self):  
 *"""Return a string representation of the hand."""* return 'Hand: ' + ', '.join(str(card) for card in self.card\_list)  
  
 def play(self, card):  
 *"""Play a card from the hand."""* if card not in self.card\_list:  
 raise RuntimeError(f"Attempted to play {card} that is not in hand: {self}")  
 self.card\_list.remove(card)  
 return card

from Cards import Card, Deck, Hand  
  
import unittest  
  
class TestCard(unittest.TestCase):  
 *"""Class to evaluate the performance of Card class"""* def test\_card\_init(self):  
 *"""Test the instantiation of a Card."""* card = Card('3', 'hearts')  
 self.assertEqual(card.value, '3')  
 self.assertEqual(card.suit, 'hearts')  
  
 def test\_card\_repr(self):  
 *"""Test the machine representation of a Card."""* card = Card('3', 'hearts')  
 self.assertEqual(repr(card), "Card('3', 'hearts'")  
  
 def test\_card\_str(self):  
 *"""Test the human-readable representation of a Card."""* card = Card('3', 'hearts')  
 self.assertEqual(str(card), "3 of hearts")  
  
 def test\_card\_lt(self):  
 *"""Test the less-than comparison of two Cards."""* card1 = Card('3', 'hearts')  
 card2 = Card('4', 'hearts')  
 self.assertTrue(card1 < card2)  
  
 def test\_card\_eq(self):  
 *"""Test the equality comparison of two Cards."""* card1 = Card('3', 'hearts')  
 card2 = Card('3', 'hearts')  
 self.assertTrue(card1 == card2)  
  
class TestDeck(unittest.TestCase):  
 *"""Class to evaluate the performance of Deck class"""* def setUp(self):  
 *"""Set up a deck of cards for each test."""* self.deck = Deck()  
  
 def test\_deck\_init(self):  
 *"""Test the initialization of a Deck."""* self.assertEqual(len(self.deck.card\_list), 52)  
  
 def test\_deck\_len(self):  
 *"""Test the length of the Deck."""* self.assertEqual(len(self.deck), 52)  
  
 def test\_deck\_sort(self):  
 *"""Test the sorting of the Deck."""* self.deck.shuffle() # Shuffle first  
 self.deck.sort() # Sort it in order  
 self.assertEqual(self.deck.card\_list[0].suit, 'clubs') # Assume clubs  
 self.assertEqual(self.deck.card\_list[0].value, 1) # Assuming clubs value 1  
  
 def test\_deck\_str(self):  
 *"""Test the string representation of the Deck."""* deck\_str = str(self.deck)  
 self.assertIsInstance(deck\_str, str)  
 self.assertIn('of', deck\_str)  
  
 def test\_deck\_shuffle(self):  
 *"""Test the shuffle method."""* original\_deck = self.deck.card\_list.copy()  
 self.deck.shuffle()  
 self.assertNotEqual(original\_deck, self.deck.card\_list)  
  
 def test\_deck\_draw\_top(self):  
 *"""Test drawing the top card from the Deck."""* top\_card = self.deck.card\_list[-1]  
 drawn\_card = self.deck.draw\_top()  
 self.assertEqual(top\_card, drawn\_card)  
 self.assertEqual(len(self.deck), 51)  
  
  
class TestHand(unittest.TestCase):  
 *"""Class to evaluate the performance of Hand class"""* def setUp(self):  
 *"""Create a Hand instance with some Cards for testing."""* self.hand = Hand([Card(3, 'hearts'), Card(4, 'diamonds'), Card(2, 'spades')])  
  
 def test\_hand\_init(self):  
 *"""Test the initialization of Hand with a list of Cards."""* self.assertEqual(len(self.hand.card\_list), 3, "Hand should be initialized with 3 cards")  
  
 def test\_hand\_len(self):  
 *"""Test the \_\_len\_\_ method to ensure it returns the correct number of cards."""* self.assertEqual(len(self.hand), 3, "Length of hand should be 3")  
  
 def test\_hand\_sort(self):  
 *"""Test the sort method to ensure the Hand is sorted correctly."""* self.hand.sort()  
 self.assertEqual(self.hand.card\_list[0].suit, 'diamonds', "First card should be of suit 'diamonds'")  
 self.assertEqual(self.hand.card\_list[0].value, 4, "First card should be 4 of diamonds")  
  
 def test\_hand\_str(self):  
 *"""Test the \_\_str\_\_ method for a correct string representation of the Hand."""* hand\_str = str(self.hand)  
 self.assertIsInstance(hand\_str, str)  
 self.assertIn("4 of diamonds", hand\_str, "String representation of hand should contain '4 of diamonds'")  
  
 def test\_hand\_play(self):  
 *"""Test the play method by playing a card from the Hand."""* card\_to\_play = Card(4, 'diamonds')  
 played\_card = self.hand.play(card\_to\_play)  
 self.assertEqual(played\_card, card\_to\_play, "The played card should be the same as the card requested to play")  
 self.assertEqual(len(self.hand), 2, "The hand should have one less card after playing")  
  
 def test\_play\_card\_not\_in\_hand(self):  
 *"""Test the play method raises an error if the card is not in the Hand."""* card\_not\_in\_hand = Card(5, 'hearts')  
 with self.assertRaises(RuntimeError):  
 self.hand.play(card\_not\_in\_hand)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()