Outline

ADT Deck

Objective: To Simulate a Deck of Cards

Deck of Cards ADT, using Python's class:

```
class Deck:
 def __init__ (self):
  """post:  create a 52-card deck, standard order"""

 def shuffle(self):
  """post:  randomizes the order of cards."""

 def deal(self):
  """deal a single card
  pre:  the deck is not empty
  post:  returns the next card and removes it."""
```

ADT Deck

Deck of cards: some thoughts

- A `Deck` object is a container class for `Card` objects, which have rank and suit attributes.
- We added a method that allows the client program to check if any cards are left in a deck:

```
 def size(self):
  """ Cards left
  """ post:  returns the number of cards left in
the deck."""
```

- note the precondition in `deal` method to `self.size()>0`

Specifying the Hand Class

Bridge

The Bridge card game has four players or hands. Each player gets 13 cards and it is better to put them in some order.

You can find information about its rules online, for example here: https://www.acbl.org/learn/

You can also find some video-instructions on how to play it, say on YouTube, for example
here: https://www.youtube.com/watch?v=Tyd7KlsRY04
or here: https://www.youtube.com/watch?v=9yzS_26fICk (part 1)

Specifying the Hand Class

Problem: To Simulate a Bridge Hand

If we want to write a program to play the card game bridge, we need a new class to represent a legal hand for bridge.

We will use the `Card` and `Deck` abstractions.

A hand need to be able to:

- deal: Deal a shuffled deck into 4 13-card bridge hands.
- sort: Sort the suits of each hand (Ace is highest), and
- dump: print out the contents of each hand.

  Other methods may need to be defined during the implementation of the listed ones.

Specifying the Hand Class

Specification for Hand class

```
class Hand:
 """A labeled collection of cards that can be sorted """

 def __init__(self, label=""):
    """Create an empty collection with the given label """

 def add(self, card):
    """Add a card to the hand """

 def sort(self):
    """Arrange the cards in descending bridge order """

 def dump(self):
    """Print out the contents of the Hand """
```

Specifying the Hand Class

Creating a Bridge Hand

```
class Hand:
 def __init__(self, label=""):
    self.label = label
    self.cards = []
 def add(self, card):
    self.cards.append(card)
 def sort(self):
    """put code for sort here"""
 def dump(self):
    print(self.label + "'s Cards:")
    for c in self.cards:
       print(" ", c)
```

Specifying the Hand Class

Comparing Cards - this code is at the end of the Card class

```
def __eq__(self, other):
    return (self.suit_char == other.suit_char and
        self.rank_num == other.rank_num)
def __lt__(self, other):
    if self.suit_char == other.suit_char:
        return self.rank_num < other.rank_num
    else:
        return self.suit_char < other.suit_char
def __ne__(self, other):
    return not(self == other)
def __le__(self, other):
    return self < other or self == other
```

Specifying the Hand Class

Sorting A Hand Manually With Selection Sort

```
def sort(self):
    cards0 = self.cards
    cards1 = []
    while cards0 != []:
        next_card = max(cards0)
        cards0.remove(next_card)
        cards1.append(next_card)
    self.cards = cards1
```

Running time: $\Theta(n^2)$

Specifying the Hand Class

Sorting A Hand Using Python's sort

```
def sort(self):
    self.cards.sort()
    self.cards.reverse()
```

Running time: $\Theta(n \log n)$

Incremental Development and Unit Testing

Unit testing

Once the development is broken into separate classes, it is nice to be able to test each class once it's developed.

Moreover, it is very convenient to test the class *as* it's being developed!

Recall the tests that we have for cardADT.py and for Card.py.

How about testing just one of the behaviors/operations? Like rank, or rankName, ...

Testing a component in isolation is knows as unit testing.

Incremental Development and Unit Testing

Benefits of writing unit tests

- tests can be run again when we go back and modify the code

- running a modified program against the previously successful tests is called regression testing

- writing unit tests while writing the class helps to work out the design of a class
- test-driven-development is when tests are written before any actual production code is added to the system. This way as each function/method is added, it is immediately testable.

Incremental Development and Unit Testing

Test-driven-development

-write the original class with each method containing just **pass** statement

- write the test code for a method, and then

- implement enough of the class to get the test to pass

-keep repeating the process of writing a test and modifying the class until the class is complete and passes all the tests.

Unit Testing Card's rank() method

Let's test the rank() method of our Card class.

```
import unittest        a framework for unit testing
from Card import *
class RankTest(unittest.TestCase):
 """ Tests Rank methods:  rank() and rankName() """
 def testRanks(self):
  """ creates cards of rank 2 through 14 of clubs and
verifies that the created card's rank is equal to the
rank it was created with """
  for i in range(2,15):
  myCard = Card(i,'c') # create i of clubs
  self.assertEqual(myCard.rank(),i)
```

Unit Testing Card's rank() method

TestCase

`TestCase` class defines a number of useful methods for unit tests.
Two commonly used:
* `assertEqual` (also known as `failUnlessEqual`)
* `assertNotEqual` (also known as `failIfEqual`)

Each method takes two parameters that are tested for equality.

Within our class RankTest, every method that starts with `test` will be called automatically by the `unittest` framework.

Unit Testing Card's rank() method

```
def main():
 unittest.main()
main()
```