Outline

Overview

Programs manipulating large data sets

When we start considering programs that manipulate large data sets, we need to use more efficient algorithms.

Often the key to an efficient algorithm lies in how the data is organized, the so-called **data structures**.

OO-programs often use **container classes** to manage collections of objects.

Examples: lists, dictionaries in Python

We will design special-purpose container classes that are not built-in to Python (or C++), whose methods will be carefully implemented based on efficiency issues.

Overview

Programs manipulating large data sets

When we start considering programs that manipulate large data sets, we need to use more efficient algorithms.

Often the key to an efficient algorithm lies in how the data is organized, the so-called **data structures**.

OO-programs often use **container classes** to manage collections of objects.

Examples: lists, dictionaries in Python

We will design special-purpose container classes that are not built-in to Python (or C++), whose methods will be carefully implemented based on efficiency issues.

Overview

Programs manipulating large data sets

When we start considering programs that manipulate large data sets, we need to use more efficient algorithms.

Often the key to an efficient algorithm lies in how the data is organized, the so-called **data structures**.

OO-programs often use **container classes** to manage collections of objects.

Examples: lists, dictionaries in Python

We will design special-purpose container classes that are not built-in to Python (or C++), whose methods will be carefully implemented based on efficiency issues.

Overview

Programs manipulating large data sets

When we start considering programs that manipulate large data sets, we need to use more efficient algorithms.

Often the key to an efficient algorithm lies in how the data is organized, the so-called **data structures**.

OO-programs often use **container classes** to manage collections of objects.

Examples: lists, dictionaries in Python

We will design special-purpose container classes that are not built-in to Python (or C++), whose methods will be carefully implemented based on efficiency issues.

Interface for the list Class

Lists are Containers

A container class provides objects which contain collections of other objects.

Usually, containers are homogeneous—the data is all of one type. But a Python list can contain string, int, and float values at the same time.

Interface for the list Class

Python list Method Specifications

- Concatenation `list1 + list2`
- Repetition `list1 * int1` or `int1 * list1`
- Length `len(list1)`
- Index `list1[i]`
- Slice `list1[start:stop:step]`
- Membership `item in list1`
- Append `list1.append(obj1)`
- Insert `list1.insert(int1, obj1)`
- Delete index `list1.pop(i)`
- Remove object `list1.remove(obj1)` see more at https://docs.python.org/3/tutorial/datastructures.html

Interface for the list Class

Slice `list1[start:stop:step]`

Parameter Values for slicing

- The `step` parameter in the slice operation can be negative (it means step backwards).
- If the `step` parameter is missing, its default value is assumed to be 1.

  (The book only shows start and stop. This will work to step through each value, since the default step is 1. But to skip the odd indices, say, you would use step = 2.)

Specifying the Deck Class

Sequential Collections

- A **sequential collection** is a container which allows one to traverse its objects sequentially.
- If the collection is not empty, it will have a first item.
- Each item (except the last) will have a next item after it.
- Starting from the first item, the entire collection is traversed by going to the next item until the last item is reached.
- A deck of cards is a sequential collection: the top card is the first, when the current card is removed, the next card is now at the top of the deck. The last card is at the bottom of the deck.

Specifying the Deck Class

Sorted Lists

- A *sorted list* is a homogeneous list where the items are *increasing* (each item is less than the next item) or *decreasing* (each item is greater than the next item).
- The items must be comparable: there is a binary operator (<) returning a boolean value.
- A deck of cards can be sorted, but for games, they are unsorted by shuffling.

Specifying the Deck Class

Objective: To Simulate a Deck of Cards

Let's create a Deck of Cards ADT, using Python's class:

```
class Deck(object):
 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."""
```

Specifying the Deck Class

Deck of cards: some thoughts

- A `Deck` object is a container class for `Card` objects, which have rank and suit attributes.
- We need to add 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."""
```

- update preconditions in `deal` method to `self.size()>0`

Implementing the Deck Class

Concrete Representation

Attributes of Deck class:

- `cards`, a Python's list of `Card` objects, as defined in Chapter 2.

  Remark: An Abstract Data Type, when implemented, should only have attributes which are private, that is, with an underscore (_) as first character.

  The book does not do that here, which is unsafe. If a function outside the class has access to the concrete representation, then it will become broken if that representation changes, which is exactly what we want to avoid.

Implementing the Deck Class

Concrete Representation

Methods:

- __init__(self) creates a 52 Card deck.
- shuffle(self) prepares for random dealing by putting the list of Cards in random order.
- deal(self), returns a `Card` object, while removing it from the list `cards`.
- size(self) returns the number of cards remaining in the list. (See Deck.py in Chapter 3 or on our web-page)

Implementing the Deck Class

Concrete Representation of Deck class

```
def __init__(self):
    cards = []
    for suit in Card.SUITS:
        for rank in Card.RANKS:
            cards.append(Card(rank,suit))
    self.cards = cards
def shuffle(self):
    n = self.size()
    cards = self.cards
    for i,card in enumerate(cards):
        pos = randrange(i,n)
        cards[i] = cards[pos]
        cards[pos] = card
```

Implementing the Deck Class

Concrete Representation of Deck class

```
def __init__(self):
    cards = []
    for suit in Card.SUITS: 4 iterations
        for rank in Card.RANKS: 13 iterations
            cards.append(Card(rank,suit)) 2 operations
    self.cards = cards 1 operation
def shuffle(self):
    n = self.size()
    cards = self.cards
    for i,card in enumerate(cards):
        pos = randrange(i,n)
        cards[i] = cards[pos]
        cards[pos] = card
```

Implementing the Deck Class

Concrete Representation of Deck class

```
def __init__(self):    running time is O(n), n is # of cards
    cards = []
    for suit in Card.SUITS: 4 iterations
        for rank in Card.RANKS: 13 iterations
            cards.append(Card(rank,suit)) 2 operations
    self.cards = cards 1 operation
def shuffle(self):
    n = self.size()
    cards = self.cards
    for i,card in enumerate(cards):
        pos = randrange(i,n)
        cards[i] = cards[pos]
        cards[pos] = card
```

Implementing the Deck Class

Concrete Representation of Deck class

```
def __init__(self):    running time is O(n), n is # of cards
    cards = []
    for suit in Card.SUITS:
        for rank in Card.RANKS:
            cards.append(Card(rank,suit))
    self.cards = cards
def shuffle(self):
    n = self.size()
    cards = self.cards
    for i,card in enumerate(cards):  n operations, n iterations
        pos = randrange(i,n) 1 operation
        cards[i] = cards[pos] 1 operation
        cards[pos] = card 1 operation
```

Implementing the Deck Class

Concrete Representation of Deck class

```
def __init__(self):    running time is O(n), n is # of cards
    cards = []
    for suit in Card.SUITS:
        for rank in Card.RANKS:
            cards.append(Card(rank,suit))
    self.cards = cards
def shuffle(self):    running time is O(n)
    n = self.size()
    cards = self.cards
    for i,card in enumerate(cards):  n operations, n iterations
        pos = randrange(i,n) 1 operation
        cards[i] = cards[pos] 1 operation
        cards[pos] = card 1 operation
```

Implementing the Deck Class

Full Implementation of Deck class

See the file Deck.py

Specifying the Hand Class

Bridge

Now we will talk about the Bridge card game.

You can find information about its rules online, for example here:
http://www.acbl.org/learn_page/how-to-play-bridge/

And you can 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

We want to write a program to play the card game bridge. We will use the `Card` and `Deck` abstractions, but we need a new class to represent a legal hand for bridge.

We 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.

  We understand that other methods will be defined in implementing these basic ones.

Note that Card.py was modified to have ranks from 2 to 14.

Specifying the Hand Class

Specification for Hand class

```
class Hand(object):
 """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(object):
 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 - will add this code to 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: $\Theta(n \log n)$

Working with the Deck, Hand, Card Classes

use Deck.py, Card.py and Hand.py to:

Objective 1:

(this is not a homework assignment)

Write a program that will generate 4 hands and show(print) all the cards in each hand.

Working with the Deck, Hand, Card Classes

Objective 2:
Implement the card game described below (this is a homework assignment).
The game is played by two players.
A deck of cards is put face down and a random card is drawn from the deck, both players see the **trump suit** and the card is put back into a random place into the deck.
At each turn: players grab one card each from the top of the deck and put them face up on the table.
If both cards have the same suit, then the player with the highest ranked card wins this turn, takes this pair of cards and places them into his/her pile.
If cards have different suits, then the person with a trump card wins his turn, takes this pair of cards and places them into his/her pile.
If the cards have different suits and none of them is a trump, then it is a tie, the cards are discarded.

go to the next slide...

Working with the Deck, Hand, Card Classes

Objective 2 (continues):

The game ends when there is no more cards in the deck. The person with more cards in his/her pile wins.
Your program must print each turn, by showing the both players cards and telling who won this turn.
At the end, it should say who won the game and how many cards each player has.

comments: note that at this level you should define classes (for example, for player, for scoring and so forth) where appropriate, also don't forget documentation (both inner comments that start with # and docstrings)!